Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp379-384)

Metamodeling UML in Rewriting Logic

JIANFEI YIN ZHONG MING
School of Software
Shenzhen University
No. 2336, Nanhai Ave. Nanshan Dist. Shenzhen 518060
P.R. CHINA

Abstract: - Despite the popular MDA development paradigm, currently there is still some vague explanation of the core concepts of MDA, such as model instantiation, verification, reflection, composition and transformation, etc. To get a better understanding the core concepts, a metamodeling approach is adopted to metamodel the UML, the important member of MDA. Based on rewriting logic and an implement environment Maude, the approach may introduce important benefits to the MDA core concepts, such as formal support, provision for rigorous specifications, and easy access to Maude's toolkit.

Key-Words: - MDA, UML, Rewriting Logic, Metamodeling

1 Introduction

Despite the popular MDA development paradigm, currently there is still some vague explanation of the core concepts, such as model instantiation, verification, composition and transformation, etc. Understanding above concepts clearly is crucial to engage in the Model-Driven Development (MDD). One of the reasons of the vague explanation comes from using the semantic formalism (class diagram + OCL + natural language) to define MDA related concepts. Especially when we use them in cycle, e.g., using MOF to define UML and OCL , using UML to define MOF and OCL, etc.

To get a better understanding the MDA core concepts, we adopt a metamodeling approach to metamodel the UML, the important member in MDA. The metamodeling approach uses a wide-spectrum reflective formalism named rewriting logic and an environment Maude [1] to describe several important aspects of the UML, such as model instantiation, verification, composition and transformation, etc. The result Maude object-oriented algebraic specifications are organized into a framework named OMCR (Object Message Concurrent Rewriting) framework.

Through the initial process of metamodeling UML in Maude, OMCR explores the semantics of model instantiation, verification, reflection, composition and transformation by programming them in rewriting logic. The framework may introduce important benefits to the MDA core concepts, such as formal support, provision for rigorous specifications, and easy access to Maude's toolkit.

2 Related Work

In [2], formally supporting the evolution of the UML metamodel is presented. The metamodel of class diagram is built using Maude *fmod* modules (functional modules). The extended *fmod META-LEVEL* is used as the meta-metamodel for meta-representing the metamodel of class diagram. The formalization provides weak support for the object and relationship semantics of the UML metamodel. For example, modeling *UML*::*Attribute* as sort not class, so inheritance relationships of UML model elements can't be supported; The roles and relationships in the UML metamodel are less concerned (see Sect. 5). Because Maude *fmod* modules can't use rewriting rules, the model transformation will be implemented in equational style and may lead to hard to code and reuse (see Sect. 7). In [3], a rudimentary UML virtual machine (UML-VM) as a Smalltalk extension is presented. Leveraging ST-VM to build UML-VM is a feasible method. But the inter-instantiation of Smalltalk ST-3 and ST-4 and how to relate it to MDA M3-level are not described clearly, which is important for self-description of MOF [4].

3 OMCR concepts

OMCR is built on top of Maude object-oriented algebraic specification. Maude object-oriented model is state (made of objects and messages) concurrent rewriting and asynchronous message communication. The system state is an instance of the sort *Configuration*. An instance of the sort *Configuration* is multiset made up of Maude objects and messages. At runtime, the system state will be rewritten concurrently by the rewriting engine against the user-defined rewriting rules. The general form of rewrite rule is following:

$$crl\ [r]\ :\ < id_0 : C_0 \mid avl_0 > \ldots < id_m : C_m \mid avl_m > M_0\ \ldots\ M_p \Rightarrow\ < id_0^{'} : C_0^{'} \mid avl_0^{'} > \ldots < id_n^{'} : C_n^{'} \mid avl_n^{'} >\ M_0^{'}\ \ldots M_q^{'}\ if\ \ Con\ .$$

Where r is the rule's label. avl and $avl^{'}$ are lists of pairs of attribute identifier and value, M and $M^{'}$ are messages, Con is the condition of the rule. The meaning of the rule is: in the system state, if local state (subset of the system state) matches (modulo associativity and commutativity [1]) the LHS of the rule and Con is satisfied, the local state will be replaced by corresponding instance of the RHS of the rule. See the reference [1] for details of Maude object-oriented algebraic specification.

According to the m and p parameters in the rule r, we classify rewriting rules to describe the coordination aspects of objects as follows:

1) When $p < 0 \wedge m \geq 0$, there are no messages participating the rule, and the coordination among objects is realized through shared objects. We name the rule OC (Object Configuration) rule, otherwise OM (Object Message) rule, in which the coordination among objects is realized through messages passing.

2) When $p = 0 \wedge m = 0 \wedge oid(M_0) = id_0$, where oid is function to extract the object identifier of a message, we name the rule asynchronous OM rule, which defines the rule of rewriting the matched local state when the object identified by id_0 has received the asynchronous message M_0.

3) When $((p > 0 \wedge m = 0) \vee (p \geq 0 \wedge m > 0)) \wedge (\forall i \in [0, p] \rightarrow \exists j \in [0, m] \wedge oid(M_i) = id_j)$, we name the rule synchronous OM rule, which defines the rule of rewriting the matched local state when one object needs more than one asynchronous messages to act, or more than one object need one or more than one message to act synchronously.

Using OM and OC rules as building blocks, we can identify the coordination patterns of objects. As UML models can be described by Maude objects (see Sect. 4), different kinds of rewriting rules will be chosen to programming model operations.

4 OMCR framework for UML

Figure 1 shows main packages of the OMCR framework. We use term "package" to divide metamodel elements and their instance conceptually according to different subjects.

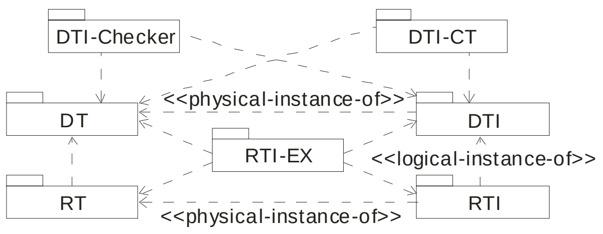


Fig. 1. Main packages of OMCR framework

According to the abstract syntax of models at M1-level and M0-level of the MDA 4-layered metamodel architecture [4], the UML metamodel is divided into segments at design-time (the DT package) and run-time (the RT package). The instance packages DTI and RTI are corresponding to meta-representations [1] of models at M1-level and M0-level. The facility packages DTI-Checker, DTI-CT and RTI-EX work on the DTI/RTI packages according to the DT/RT packages. The packages in Fig.1 can be divided into more sub-packages according to the UML metamodel. The OMCR framework are briefly described as follows:

4.1 DT/RT and DTI/RTI packages

The DT (Design-Time) package is a class library for the UML metamodel at design-time. The DT package is further divided into sub-packages according to the UML Specification [5], such as *Classes*, *Components*, *Actions*, etc. For example, the signature for the metaclass *Class* in the *Kernel* package of the UML metamodel is following:

```
(omod CLASS is inc CLASSIFIER .
class Class | ownedAttributes :
 OidList, ownedOperations : OidList,
 nestedClassifiers : OidList .
subclass Class < Classifier .
msg getOwnedAttributes Oid Oid -> Msg .
msg getOwnedAttributes-r Oid OidList ->
 Msg .
msg apdOwnedAttributes Oid OidList ->
 Msg .
msg rmvOwnedAttributes Oid OidList ->
 Msg .
*** other get/apd/rmv msgs omitted here
endom)
```

Because Maude message is always asynchronous, a code style *messageName-r* is used to code return messages, such as *getOwnedAttributes-r* message. Because Maude message and operation are first-class, non-trivial messages and operations can be coded later in separate modules, which provides a programming model in multi-method style.

The RT (Run-Time) package is class library for the UML metamodel at run-time. The RT package includes model elements, such as

InstanceSpecification, *Slot*, *ValueSpecification*, etc. For example, the signature for the model element *InstanceSpecification* in the *Kernel* package of the UML metamodel is following:

```
(omod INSTANCE-SPECIFICATION is
 inc PACKAGEABLE-ELEMENT .
class InstanceSpecification |
   classifiers  :  OidList,  slots  :
  OidList, specification : Oid .
subclass InstanceSpecification
 < PackageableElement .
 *** get/apd/rmv msgs omitted here
endom)
```

The DTI (Design-Time Instances) package includes objects physical-instantiated [6] from classes in the DT package. An object in the DTI package meta-represents [1] a model element at M1-level. For example, an object of *Class* in the DT package is following (some attributes are omitted for conciseness):

```
< cId : Class | name : “People”,
   visibility : package, isLeaf : true,
  isAbstract : false, ownedAttributes :
  pId >
 < pId : Property | name : “age”, isStatic
  : false, isReadOnly : false, isDrived
  : false, aggregation : none, type : ptId
  >
< ptId : PrimitiveType | name : “Integer”
>
```

The *cId* identified object meta-represents user-class (at M1-level) named “*People*” with attribute named “*age*” whose type name is “*Integer*”.

The RTI (Run-Time Instances) Package includes objects physical-instantiated from classes in the RT package. An object in the RTI package meta-represents a model element at M0-level. For example, an object of *InstanceSpecification* is following (some attributes are omitted for conciseness) :

```
< isId : InstanceSpecification |
 classifiers : cId, slots : sId >
< sId : Slot | values : liId,
 definingFeature : pId >
< liId : LiteralInteger | value : 25,
 type : ptId >
```

The *isId* identified object meta-represents an user-object whose classifier is the “*People*”. The *isId* identified object has slot object whose classifier is the “*age*”. The slot has the value 25 whose classifier is the “*Integer*”. Each object in the RTI package has at least [7] a metaobject in the DTI package. The relationship is so-called “logical-instance-of” relationship [6].

4.2 DTI-Checker, DTI-CT and RTI-EX packages

The DTI-Checker (Design-Time Instances Checker) package is used to check whether the content of the DTI package is well-formed or not. The DTI-checker package implements the constraints defined for the UML metamodel.

The DTI-CT (Design-Time Instances Composition & Transformation) package is used to operate the content of the DTI package. We think model operation can be unified in the view of rewriting logic (see Sect. 7).

The RTI-EX (Run-Time EXecution) package is used to operate the content of the RTI package. Because each object *A* in the RTI package has at least [7] metaobject *B* in the DTI package. *B* defines the structure features and behavior rules of *A*, in theory, we can access *A*’s meta-level *B* when running the meta-representations of M0-level models. Accessing the meta-level will do following reflective activities:
1) Dynamic model checking *A*’s state against the structure features of *B*.
2) Reading behavior rules of in order to run by rewriting *A*’s state.
3) Changing the definitions of *B* to change the future behavior of *A*.

The RTI-EX package will be designed as the place to implement above dynamic model reflection. For reason of space, we leave the design of the RTI-EX package to another paper. In following sections, we discuss designs of main packages of the OMCR framework.

5 Design of DT/RT packages

The design of DT/RT packages is very important for OMCR framework, because they provide type information for other OMCR packages. Using a code generator to generate main contents of the DT/RT packages is possible. So we need to consider : 1) How to map UML metamodel elements to Maude class modules. 2) How to organize the result modules.

To the first question, because the attributes of UML metamodel elements can be mapped directly to corresponding attributes of Maude classes, we emphasize how to map the relationships of UML metamodel elements to Maude codes. For example, Fig. 2 shows the relationships of the metaclass *Behavior* in the UML metamodel.

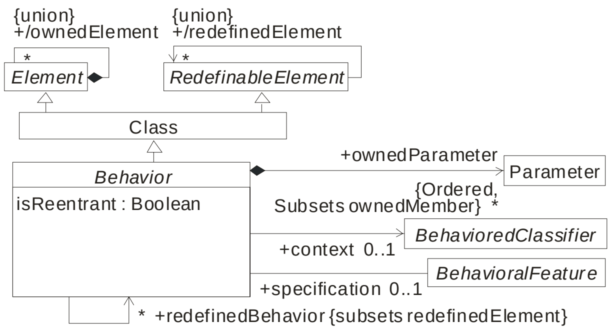


Fig. 2. Relationships of the metaclass Behavior

In Fig. 2, from the view of box (class notation), there are three kinds of relationships, namely, inheritance (vertical dimension), composition and non-composition (horizontal dimension). For using the object-oriented programming style and rewriting logic, composition relationships will be implemented similarly as non-composition relationships besides additional codes for objects creation and deletion. The class module for the metaclass *Behavior* in Fig. 2 is following:

```
(omod BEHAVIOR is inc CLASS .  *** I&P
class Behavior | isReentrant : Bool,
  parameters : OidList,        *** C
  context : Oid, specification : Oid,
  redefinedBehaviors : OidList . *** N
subclass Behavior < Class .   *** I&P
msg getParameters Oid Oid -> Msg .
msg getContext Oid Oid -> Msg . *** C|N
 *** other get/apd/rmv omitted here
 vars O X : Oid.
 vars EIL PIL GL AL NL OL PL : OidList.
 rl  :  getOwnedMembers(O,  X)  <  O  :
 Behavior|
  elementImports : EIL,
  packageImports : PIL,
  generalizations : GL,
  ownedAttributes : AL,
  nestedClassifiers : NL,
  ownedOperations : OL,
  parameters     :     PL     >     =>
 getOwnedMembers-r(X, EIL ; PIL ; GL ; AL
 ; NL ; OL ; PL)
<O : Behavior | > . *** S&I
*** other rules omitted here endom)
```

Comments are added to indicate the contributions from the UML metamodel information, namely, *I* stands for inheritance, *C* for composition, *N* for non-composition, *P* for package, *S* for subsets constraint. The subset constraints are always come with inheritance relationships (marked with *S&I*).

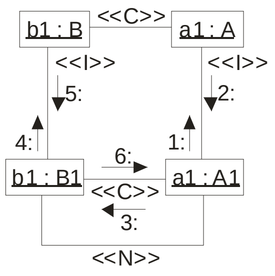
To the second question, we map UML metamodel elements to Maude class modules one by one. For horizontal relationships, we use the sort *OidList* that is independent of type information of the horizontal relationships, so we needn't import corresponding modules. For vertical relationships, the modules of superclasses have to be imported for inheritance declaration, rule overriding, etc. Using this mode, generating a Maude class module for a UML metamodel element will only need to access limited meta-information, which makes the code generation easy to implement.

As the design of the DT/RT packages, we can also generate the DTI/RTI packages through code generators. For reason of space, we omit the design details of DTI/RTI packages.

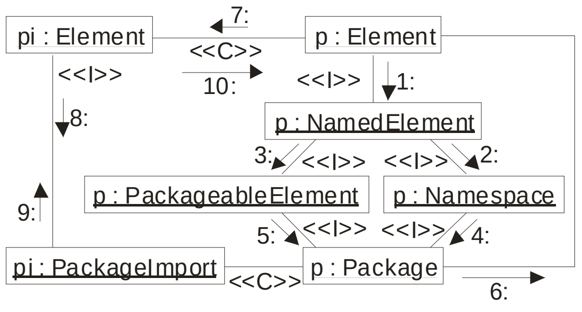
6 Design of DTI-Checker package

In this section, we discuss the design of the general checking flow in the DTI-Checker package. After the flow is designed, concrete constraints checking rules can be inserted into the flow, which is an application of template method pattern. According to the relationship classification of UML metamodel elements (Sect. 5), instances of the UML metamodel elements will be checked in two dimensions (i.e. composition and inheritance relationships). We first show the checking flow for an object in the general object-oriented style, then that in the rewriting logic style. By comparing them, we present the important design principles for the DTI-Checker package.

Figure 3(a) shows the general checking flow for the object *a1 : A1* in the general object-oriented style. Composition relationships are marked with *<<C>>*. Inheritance relationships are marked with *<<I>>*. Classes *A1* and *B1* inherit classes *A*, *B* respectively.



(a)



(b)

Fig. 3. General checking flow in the general object-oriented style (a) and in the rewriting logic style (b)

For lack of constructs for procedure invoking in rewriting logic, the checking flow will be implemented using additional states coding, such as the states coding for invoking methods of a superclass and returning back to the invoking point. To decrease additional states coding, we propose an Control Flow Reversed (CFR) design principle:

Control flows should be reversed as far as possible to be expressed as message flows in rewriting logic.

By this principle, we should move the start point of the checking flow from leaf class to its root superclass. From the UML metamodel, we can elicit an observation named Root Generalized Composition relationship (RGC):

Whenever there is a composition relationship between subclasses, the composition relationship must be generalized to that of corresponding superclasses by subsets constraints. The root of all generalized composition relationships is the association (owner, ownedElements) between the metaclass Element and itself. See Fig. 2 for an example.

Using the CFR principle and RGC observation, the general checking flow for an object in the rewriting logic style can be coded in the metaclass *Element*. Fig. 3(b) shows an example of checking a package metaobject *p* : *Package* in the rewriting logic style. For reason of space, we omit the Maude codes for the general checking flow algorithm.

7 Design of DTI-CT package

In this section, we discuss design principles of model operations. First, a unified form for model composition and transformation in rewriting logic is presented, then the design of model transformation is discussed.

Under Maude object-oriented algebraic specification, when models are described as instances of the sort *Object*, model compositions are natural applications of the mixfix operation (with the empty syntax __) declared in the mod *CONFIGURATION* :

```
  subsort Object Msg < Configuration .
  op __ Configuration Configuration ->
    Configuration [ctor config assoc comm
 id: none]
```

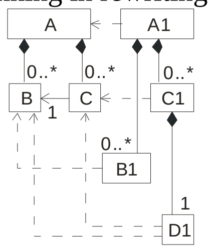
Above definition means model composition is just like object composition in Maude. Putting models of different metamodels together as an instance of the sort *Configuration,* we can combine models without changing the metamodels or designing special metamodels for model composition. Definitions of model transformations are expressed as OM and OC rules and equations. The running of rules and equations on models connected by the mixfix operation gives the meaning to model compositions. The result model is still in the composite form. Using Maude reflection, the rules and equations can be modified at the meta-level and compiled into their object-level [1], which implements a kind of higher-order model transformation. The unified model form for model composition and transformation in rewriting logic is:

$$[[o(\{\langle T_1, M_1\rangle, \langle T_2, M_2\rangle, ..., \langle T_n, M_n\rangle\})]]_R \Rightarrow \{\langle T_1, M_1^{'}\rangle, \langle T_2, M_2^{'}\rangle, ..., \langle T_n, M_n^{'}\rangle\}$$

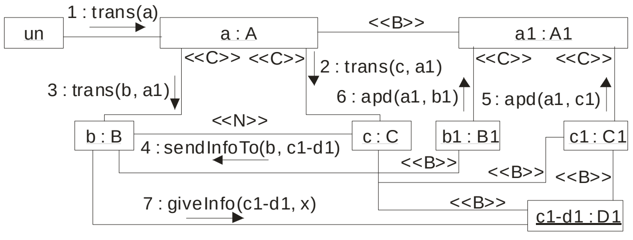
Where $T \in TypeModel$, $M, M \in TokenModel$, $R \in \rho(U)$, $o \in ModelOperationSymbol$.

$TypeModel$ and $TokenModel$ are terms proposed in [8], which correspond to elements in the DT/RT packages and DTI/RTI packages. U is the total set of rewriting rules. $[[]]_R$ means using R to interpret the application of model operation symbol o .

Model transformation is an important application of MDA. A lot of model transformation implementations follow tree-based transformation, because the composition relationships implicated in metamodels are the main contribution of the transformation. In Maude, when using only equations to do model transformation, we can only use the tree transformation model. When some dynamic or shared information (e.g. inherited and synthesized attributes [9]) are not available in current node, but needed to fill target nodes, only the tree transformation model is not enough. As an example, Figure 4 shows a metamodel of LHS and RHS and transformation running in rewriting logic.



(a)



(b)

Fig. 4. (a) Sample metamodel of LHS and RHS. (b) transformation implementation for (a)

In Fig. 4 (a), the dependence associations indicate source nodes when building target nodes. The node

D1 depends more than one nodes (i.e. *B* and *C*). The node *B* is shared between the node *B1* and *D1*, so does the node *C*. Those phenomena are common in model transformation, which indicates an observation :

Through building the target model is in the order of a tree with links between the nodes of the tree, the nodes of source model may not organize in that order.

This is one reason that only the tree transformation model is not easy to deal with model transformation. The coordination facilities (e.g. OM and OC rules) for transferring inherited and synthesized attributes are needed.

Figure 4(b) shows a running of the transformation defined in Fig. 4(a). The stereotype <<*B*>> means creating target node, other stereotypes are same as those in Fig. 3(a). Maude message format is used [1]. The first parameter of a message is the object identifier (of sort *Oid*) of the target object. The transformation running is described briefly as follows:

1) When the message 1 is received by *a : A*, an OM rule is fired to create *a1 : A1*, and send the *Oid a1* with the message 2 and 3 to the contained objects. The message 2 and 3 are for constructing the inherited attribute *Oid* of *c* and *b*.

2) When the message 2 is received by *c : C*, an OM rule is fired to create *c1 : C1*, *c1-d1 : D1* and send the message 4 to *b : B*, message 5 to *a1 : A1*. The message 5 is for constructing synthesized attribute *Oid* of *a1*.

3) When the message 4 is received by *b : B*, an OM rule is fired to send message 7 to *c1-d1 : D1* to give more information to fill it. The OM rules for message 3 and 6 are similar as that for message 2 and 5.

For reason of space, the Maude codes for above OM rules are not presented here. As we can see in above example, it will be much complex to implement the same transformation in only tree transformation model. The coordination in model transformation is another important horizontal dimension in addition to vertical dimension.

8. Acknowledgements

The research program is financially supported by National Social Science Foundation of China (No.05CTQ001), Guangdong Natural Science Foundation (No.04011304), and Shenzhen Science Technology Plan (No.200422).

9 Conclusion

Understanding the core concepts of MDA clearly is crucial to engage in the Model-Driven Development (MDD). The core concepts include model instantiation, verification, reflection, composition, transformation, etc. Based on rewriting logic, an OMCR (Object Message Concurrent Rewriting) framework is presented for metamodeling UML. The framework is divided into seven main packages to explore the semantics of the core concepts. The framework may introduce important benefits to the (usually ambiguous) UML specifications, such as formal support, provision for rigorous specifications, and easy access to Maude's toolkit.

References:

[1] Manuel Clavel, Francisco Durán and Steven Eker, et al, Maude Manual, v2.1, SRI International, 2004.

[2] Ambrosio Toval and José Luis Fernández, Formally Modeling UML and its Evolution: A Holistic Approach, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000*, Kluwer Academic Publishers, 2000, pp. 183–206.

[3] Trygve M. H. Reenskaug, Rudimentary UML Virtual Machine as a Smalltalk Extension, Working paper, University of Oslo, Norway, 2005.

[4] OMG, MOF 2.0 Core Final Adopted Specification, OMG Document ptc/03-10-04, 2003.

[5] OMG, UML Superstructure Specification, v2.0, OMG Document formal/05-07-04, 2005.

[6] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, et al, The Architecture Of Uml Virtual Machine, *Proc. 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01).* ACM Press, 2001, pp. 327–341.

[7] OMG, UML 2.0 Infrastructure Final Adopted Specifcation, OMG Document ptc/03-09-15, 2003.

[8] Thomas Kühne, What is Model?, *Proc. Language Engineering for Model-Driven Software Development. Dagstuhl Seminar Proceedings*, Vol. 04101, Internationales Begegnungs-und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[9] Kenneth Slonneger, Barry L. Kurtz, *Formal Syntax and Semantics of Programming Languages: Laboratory Based Approach*, Addison-Wesley, 1995.