Semantics of Metamodels in UML

Lijun Shan
Dept of Comp. Sci., National Univ. of Defence Tech., Changsha, 410073, China
Email: lijunshancn@yahoo.com

Hong Zhu
Dept of Computing, Oxford Brookes University, Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk

Abstract

A modelling language can be defined by a metamodel in UML class diagram. This paper defines the semantics of such metamodels through two mappings: a signature mapping from metamodels to signatures of first order languages and an axiom mapping from metamodels to sets of axioms over the signature. Valid models, i.e. instances of the metamodel, are therefore mathematical structures in the signature that satisfies the axioms. This semantics definition enables us to analyse the logical consistency and completeness of metamodels. A software tool called LAMBDES is implemented to translate metamodels into first order logic systems and analyse them by employing the theorem prover SPASS. Case studies with the tool detected inconsistency and incompleteness in the metamodel of UML 2.0 and an AspectJ profile.

1. Introduction

A metamodel defines a modelling language in the form of a model, e.g. UML class diagram. In particular, metaclasses define a classification of model elements, and associations between metaclasses define the relationships between model elements. However, the semantics of UML is informally defined. Thus, formalisation of the semantics of metamodels is essential for the study of modelling languages. In particular, through formalisation of metamodels, we aim to achieve two goals. The first is a clear definition of the logical relationship between models and metamodels, thus lay the foundation for logically proving whether a model is a valid instance of a metamodel. The second is to facilitate the analysis of the properties of language definitions, such as their consistency and completeness.

In our previous work [2], an approach called *descriptive semantics* to formalising diagrammatic models in first order logic was proposed. It defines a mapping from models to first order logic systems so that a system is an instance of a model if and only if it satisfies the logic description of the model. The formalisation of models enables to analyse properties of models through logical inference. This paper will further present the formal semantics of metamodels. We will define two mappings from metamodels to first order logic systems: a *signature mapping* from metamodels to signatures of first order languages, and an *axiom mapping* from metamodels to sets of sentences over the signatures. Given a metamodel, the signature derived from the metamodel specifies a type of mathematical structures of valid models, and the sentences derived are axioms to be satisfied by all valid models. Properties of a language definition, e.g. consistency and completeness, can be inferred from the formalised semantics of the metamodel. Note that, our approach is applicable to any metamodel depicted in UML class diagrams, no matter what modelling language it defines. The metamodel of UML 2.0 [3] is taken as a running example to illustrate the approach throughout the paper.

The remainder of the paper is organised as follows. Section 2 discusses related work. Section 3 reviews our previous work on the formalisation of UML. Section 4 presents our formalisation of the semantics of metamodels. Section 5 defines the notion of well-defined metamodels in the framework of formal semantics. Section 6 describes a software tool called LAMBDES that supports the logic analysis of metamodels based on the formal semantics. Section 7 reports two case studies. Section 8 concludes the paper with a discussion of future work.

2. Related work

The formal definition of modelling language BON in [5] is similar to our approach. In [5], the metamodel of BON is depicted in BON notation and then specified in formal specification language PVS. Modelling concepts of BON, including *abstractions* such as Class and Feature and *relationships* such as Aggregation and Association, are specified as types in PVS. Inheritance hierarchy in the metamodel are mimicked by subtype relations. The semantic relations between the modelling concepts are defined as functions in PVS. The signature of a PVS system is manually defined according to the metamodel. Then, well-formedness constraints on BON models are specified as axioms in PVS. When BON models are formalised in PVS, their well-formedness with respect to the metamodel can be checked using PVS theorem prover. It is reported that the BON metamodel was analysed and debugged through the formalisation. In comparison, we view a metamodel as more than the definition of the signature of the modelling language. For example, from an inheritance hierarchy in a metamodel, not only types of model elements and subtype relations can be generated, but also axioms on the classification of model elements. Moreover, our method is applicable to all metamodels. In other words, the domain of the semantics mapping is the set of metamodels in UML class diagrams rather than a specific metamodel for a specific language.

Various researches have been conducted in the formalisation of class diagrams in first order logic or description logic. Our rules of formalising metamodels look similar to them. For example, in [4] classes and attributes in a class diagram are translated into unary and binary predicates respectively, and a generalisation between two classes is translated into a formula with an implication between two predicates. Such formalisation enables logical reasoning about UML class diagrams as model of OO systems. In contrast, here we exploit the formalisation of UML class diagrams for the formal analysis of metamodel in the definition of modelling languages.

3. Descriptive Semantics of Models

Seidewitz pointed out that a model is '*a set of statements about some system under study*', and the meaning of a model is the set of systems that satisfy the statements [6]. In our previous work [2], we proposed to formally define the semantics of a model as a set of first order logic (FOL) sentences, taking UML as an example of modelling languages. A model is satisfied by a system if all the FOL sentences derived from the model are true in the system.

Our approach to formalising semantics of models separates *descriptive* semantics from *functional semantics*. The former determines whether a system satisfies a model, while the latter interprets basic concepts of the modelling language in the domain of modelled systems. The descriptive semantics of a modelling language is defined as a mapping from models to first order sentences, which are constructed from a set of predicates and functions via logic connectives and quantifiers. The predicates and functions represent the basic concepts of the modelling language. Satisfaction of a model by a system is the truth of the sentences representing the model's descriptive semantics with respect to the system, provided that how to evaluate the predicates and functions is known.

Formally, the descriptive semantics for a modelling language has the following structure.

Definition 1 (Semantic definition of modelling language) A *semantic definition* of a modelling language consists of the following elements.

- A signature Σ that defines a formal logic language;
- A set $\boldsymbol{Axm}_D$ of axioms about the descriptive semantics, which are first order sentences in Σ;
- A set $\boldsymbol{Axm}_F$ of axioms about the functional semantics, which are also sentences in Σ;
- A translation mapping $\boldsymbol{F}_\Sigma$ from models to a set of formulas in Σ that describes the model;
- A hypothesis mapping $\boldsymbol{H}_\Sigma$ from models to a set of formulas in Σ that represent the context in which the model is used.

Definition 2 (Semantics of a model) Given a semantic definition of a modelling language, the semantics of a model m under a hypothesis H, written $Sem_H(m)$, is defined as follows.

$$Sem_{\boldsymbol{H}}(m) = \boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}_\Sigma(m), \cup \boldsymbol{H}_\Sigma(m),$$

where $\boldsymbol{F}_\Sigma(m)$ and $\boldsymbol{H}_\Sigma(m)$ are the sets of statements obtained by applying the semantic mappings $\boldsymbol{F}_\Sigma$ and $\boldsymbol{H}_\Sigma$, on model m, respectively. The descriptive semantics of a model m under the hypothesis $\boldsymbol{H}_\Sigma$, written $DesSem_{\boldsymbol{H}}(m)$, is defined as follows.

$$DesSem_{\boldsymbol{H}}(m) = \boldsymbol{Axm}_D \cup \boldsymbol{F}_\Sigma(m), \cup \boldsymbol{H}_\Sigma(m),$$

The translation rules and hypothesis rules have been implemented in a software tool, which translates UML models into first order logic systems in the input format of the theorem prover SPASS. Case studies were conducted to check models' consistency using SPASS.

4. Semantics of Metamodels

When a modeling language is defined by a metamodel, the semantics of the metamodel is therefore a logic system in the structure given in Section 3. The formalisation of metamodels is defined by a mapping from metamodels in UML class diagrams to the semantic domain of first order logic systems.

Note that a *metamodel in UML* means a metamodel depicted in UML class diagrams, while the *metamodel for UML* is just one of such metamodels. Examples of metamodels in UML include those profiles for platform-specific models (PSM), which defines an extension to the UML language. In our semantics definition, a metamodel is an input variable to the semantic mapping rather than a constant.

4.1. The framework in institution theory

Regarding a modelling language as a specification language, our approach to the formal semantics of a modelling language outlined in Section 3 can be generalised to define the semantics of metamodels as shown in Fig. 1.

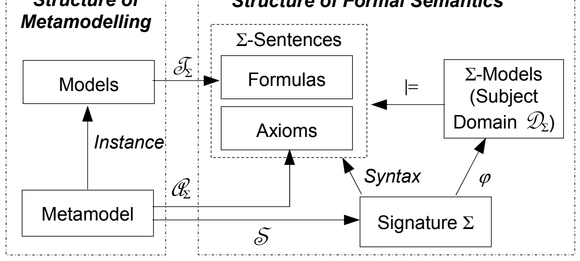

Structure of Metamodelling
Structure of Formal Semantics
Σ-Sentences
Models
Formulas
Axioms
Σ-Models (Subject Domain 𝒟Σ)
|=
Instance
Syntax
φ
Metamodel
Signature Σ
𝒯Σ
𝒜Σ
𝒮

Fig. 1 Framework of the formal semantics

From a metamodel, we derive a signature Σ that defines a first order language (FOL) in which statements about systems can be written. The mapping from metamodels to signatures is called *signature mapping* $\mathcal{S}$. As briefly described in Section 3, models can be translated into Σ-sentences by systematically applying a set of rules, called *translation mapping* $\mathcal{T}_\Sigma$. The Σ-sentences can be evaluated on systems in a subject domain $\mathcal{D}$, which corresponds to the 'collection of models' in the institution framework. The satisfaction |= of a system to a model is defined as the truth of all sentences derived from the model evaluated on the system. The set of rules that derive axioms about models from a metamodel is called *axiom mapping* $\mathcal{A}$.

The following subsections define the mappings $\mathcal{S}$, and $\mathcal{A}$, and the satisfaction relation |=.

4.2. Signature mapping

Metamodelling in UML class diagrams uses the concepts of metaclass, meta-association and meta-attribute. Metaclasses are classifications of model elements. We use a unary predicate to represent a metaclass, and hence the following signature rule.

SR_1 (*Unary predicates*). For each metaclass named *MC* in a metamodel, we define a unary atomic predicate *MC*(*x*).

A unary predicate *MC*(*x*) means that *x* is of type *MC*.

A meta-association and a meta-attribute defines a relationship between model elements. We use a binary predicate to represent a meta-association or meta-attribute, hence the following rule.

SR_2 (*Binary predicates*). For each meta-attribute *MA* of metaclass *X* with *Y* as the data type, or each meta-association from metaclass *X* to metaclass *Y* with *MA* as the association end name on *Y*, a binary predicate *MA*(*x*, *y*) is defined to represent the relation between elements of type *X* and *Y*.

A binary predicate *MA*(*x*, *y*) means that *x* and *y* are in the relation *MA*.

For example, the upper part of Fig. 2 shows a fragment of the UML 2.0 metamodel, and the lower part is an instance of the metamodel. The dashed arrows indicate instance-of relations between elements in the model and elements in the metamodel. From the metamodel, unary predicates *Class*(*x*), *Classifier*(*x*) and *Generalisation*(*x*) can be derived by applying the signature rule *SR_1*, and binary predicates *general*(*x*, *y*) and *specific*(*x*, *y*) can be derived by applying the *SR_2*. These predicates are used to construct sentences that represent descriptive semantics of instances of the metamodel. For example, by applying the translation rules given in [2] on the model shown in the lower part of Fig. 2, a set of sentences can be generated, e.g. *Class*(*Woman*), *Class*(*Person*), *Generalisation*(*wp*), *specific*(*wp*, *Woman*), *general* (*wp*, *Person*), where *wp* represents the unnamed generalisation relation from class *Woman* to class *Person*.

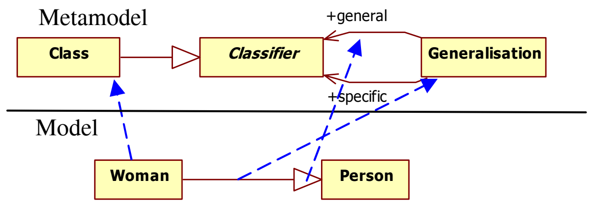


Fig. 2 Example 1 of metamodel and model

Each enumeration metaclass defines a data type, whose values are enumeration literals. We use a constant to represent an enumeration value, hence the following signature rule.

SR_3 (*Constants*). For each enumeration value *EV* given in an enumeration metaclass *ME* in a metamodel, a constant *EV* is defined.

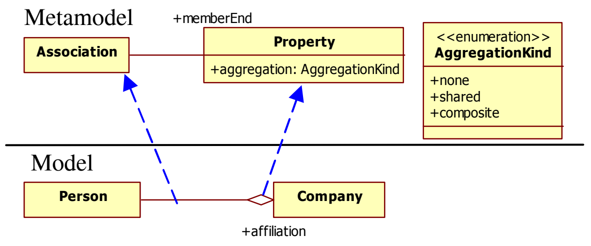


Fig. 3 Example 2 of metamodel and model

In metamodels, enumeration metaclasses are used as data types of meta-attributes. For example, in the metamodel given in the upper part of Fig. 3, enumeration metaclass *AggregationKind* is the data type of meta-attribute *aggregation* of metaclass *Property*. Constants *none*, *shared* and *composite* are derived from the enumeration values by applying *SR_3*. These constants are used in sentences which represent descriptive semantics of instances of the metamodel. For example, by applying the translation rules on the model in the lower part of Fig. 3, we can generate a set of sentences including *memberEnd*(*pc*, *affiliation*), *aggregation*(*affiliation*, *shared*), where *pc* represents the unnamed association between *Person* and *Company*.

4.3. Axiom mapping

Besides defining types of model elements through a set of metaclasses, a metamodel also specifies semantic relationships between the model elements through meta-attributes, meta-associations and generalisations. These relationships are satisfied by all valid instances of the metamodel. They can be represented as formulas in the derived signature. They must be satisfied by all valid models, hence called axioms on models. We will define a set of axiom mapping rules that systematically derive such axioms from a metamodel.

A. Classification of model elements.

The UML specification states that "*the main language constructs are reified into metaclasses in the metamodel*" [7]. Two kinds of metaclasses may be contained in a metamodel: concrete metaclasses and abstract metaclasses. The former classifies model elements. Such a classification must be complete in the sense that every model element must be an instance of a concrete metaclass. This constraint on the relationship between metaclasses and elements in a model is made explicit by the following axiom rule.

> AR_1(*Completeness of classification*). Let MC_1, MC_2, …, MC_n be the set of concrete metaclasses in a metamodel. We have an axiom in the form of
> $$\forall x.\ MC_1(x) \vee MC_2(x) \dots \vee\ MC_n(x)$$

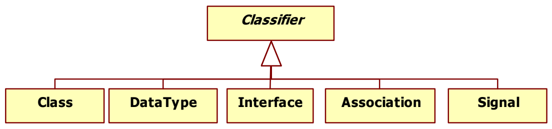


Fig. 4 Example 3 of metamodel

For example, given a metamodel in Fig. 4, the following axiom is generated by applying AR_1.

$$\forall x.\ (Class(x) \vee DataType(x) \vee Interface(x) \vee Association(x) \vee Signal(x))$$

AR_1 is necessary because, due to this axiom, any model containing an element of a undefined type is invalid. It is easy to see that the above axiom is satisfied by the model in Fig. 3, because all three elements are typed by *Class* and *Association*, respectively. But the axiom is not satisfied by the model in Fig. 2, because the arrow *wp* does not belong to either *Class*, *DataType*, *Interface*, *Association* or *Signal*, hence meaningless in the context of the metamodel in Fig. 4.

The classification of model elements to concrete metaclasses must be disjoint, because allowing an element belonging to more than one type leads to ambiguous interpretation of the element. Therefore, we have the following axiom rule.

> AR_2(*Disjointness of classification*) Let MC_1, MC_2, …, MC_n be the set of concrete metaclasses in a metamodel. For each pair of different concrete metaclasses MC_i and MC_j, $i \neq j$, we have an axiom $\forall x.\ MC_i(\mathrm{x}) \rightarrow \neg\ MC_j(x)$.

For example, the following axioms can be derived from the metamodel in Fig. 4 by applying AR_2.

$$\forall x.\ (Class(x) \rightarrow \neg\ DataType(x)),$$
$$\forall x.\ (Class(x) \rightarrow \neg\ Interface(x)), \dots$$

The above two axioms rules make it explicit that a model must have its elements completely and uniquely classified by metaclasses.

B. Inheritance hierarchy on metaclasses.

Inheritance hierarchy of metaclasses represent the taxonomy of modelling concepts. "*Each instance of the specific classifier is also an indirect instance of the general classifier*" [8]. This can be expressed as implication between the predicates, thus, the following rule.

> AR_3(*Logical implication of inheritance*) For a generalisation relation from metaclass MA to MB in a metamodel, we have an axiom in the form of
> $$\forall x.\ MA(x) \rightarrow MB(x).$$

For example, by applying AR_3 to the metamodel shown in Fig. 4, the following axioms can be derived, stating that if a model element has the type *Class* or *DataType* or *Interface* or *Association* or *Signal*, it also belongs to the type *Classifier*.

$$\forall x.\ Class(x) \rightarrow Classifier(x),$$
$$\forall x.\ DataType(x) \rightarrow Classifier(x),$$
$$\forall x.\ Interface(x) \rightarrow Classifier(x),$$
$$\forall x.\ Association(x) \rightarrow Classifier(x),$$
$$\forall x.\ Signal(x) \rightarrow Classifier(x).$$

In current practice of metamodelling, all inheritance relations between metaclasses are explicitly specified in the metamodel, thus the following axiom rule.

> AR_4(*Completeness of specialisations*) Let MA be a metaclass in a metamodel and MB_1, MB_2, …, MB_k be the set of metaclasses specialising MA. We have an axiom in the form of
> $$\forall x.\ MA(x) \text{ -> } MB_1(x) \vee MB_2(x) \vee \dots \vee MB_k(x).$$

For example, by applying AR_4 to the metamodel in Fig. 4, the following axiom can be derived, stating that if a model element is an instance of Classifier, it must belong to one of the types: *Class*, *DataType*, *Interface*, *Association* or *Signal*.

$$\forall x.(Classifier(x) \rightarrow Class(x) \vee DataType(x) \vee Interface(x) \vee Association(x) \vee Signal(x))$$

C. Type constraints

Let A be a meta-association from metaclass MC_1 to MC_2, MA be the association end on the MC_2 side. For the binary predicate $MA(x, y)$ derived from the association, if the first parameter is an element of type MC_1, the second must be of type MC_2. Thus, we have the following axiom rule.

> AR_5(*Types of parameters of predicates*) For each binary predicate $MA(x, y)$ derived from an association from metaclass MC_1 to MC_2 in a metamodel, we have an axiom in the form of
> $$\forall x, y.\ MA(x, y) \wedge MC_1(x) \rightarrow MC_2(y).$$

For each function *MAttr*, we also have an axiom to specify its domain and range.

> AR_6(*Domain and range of functions*) For each function $MAttr(x)$ derived from a meta-attribute *MAttr* of type *MT* in a metaclass *MC*, we have an axiom in the form of

$$\forall x,y.\ MC(x) \wedge (MAttr(x) = y) \to MT(y).$$

D. Multiplicity

Meta-association ends and meta-attributes are constrained by multiplicity. They "*constrains the size of the collection […] of instances at the other end*" [8]. Thus, we have the following axiom rule.

AR_7(*Multiplicity of binary predicate*) For each binary predicate $MA(x, y)$ derived from an association from metaclass MC_1 to MC_2 in a metamodel, let *Mul* be the multicity value specified on the association end *MA*, we have axioms in the form of

If $Mul = 0..1$:
$\forall x, y, z.\ (MC_1(x) \wedge MA(x, y) \wedge MA(x, z) \ \to (y = z))$
If $Mul = 1$ or unspecified:
$\forall x.\ (MC_1(x) \ \to \exists\ y.\ MA(x, y))$ and
$\forall x, y, z.\ (MC_1(x) \wedge MA(x, y) \wedge MA(x, z) \ \to (y = z))$
If $Mul = 1..*$: $\quad \forall x.(MC_1(x) \ \to \exists\ y.\ MA(x, y))$
If $Mul = 2..*$:
$\forall x.(MC_1(x) \ \to \exists y,z.\ MA(x, y) \wedge MA(x, z) \wedge (y \neq z))$
If $Mul = 0..2$: $\quad \forall x, y, z, u.(\ MC_1(x) \wedge MA(x, y) \wedge MA(x, z) \wedge MA(x, u) \to (y = z) \vee (y = u) \vee (u = z))$

Similarly, for each function $MAttr(x)$ derived from a metaattribute *MAttr* of type *MT* in a metaclass MC_1, we have the following axiom rule.

AR_8(*Multiplicity of function*) For each function $MAttr(x)$ derived from a metaattribute *MAttr* of type *MT* in a metaclass *MC*, let *Mul* be the multicity value of the metaattribute *MAttr*, we have axioms in the following form.

If $Mul = 0..1$: $\forall x, y, z.\ (MC(x) \wedge (MAttr(x) = y) \wedge (MAttr(x) = z) \to (y = z))$
If $Mul = 1$: $\forall x.(MC(x) \ \to \exists\ y.\ (MAttr(x) = y))$ and
$\forall x, y, z.\ (MC(x) \wedge (MAttr(x) = y) \wedge (MAttr(x) = z) \to (y = z))$
If $Mul = 1..*$: $\quad \forall x.(MC(x) \ \to \exists\ y.\ (MAttr(x) = y))$

E. Properties of enumeration values

We identified three axiom rules to characterise the information contained in each enumeration metaclass.

AR_9(*Distinguishability of the literal constants*) For each pair of different literal values *a* and *b* of an enumeration type, we have an axiom $a \neq b$.

AR_{10}(*Type of the literal constants*) For each enumeration value *a* defined in an enumeration metaclass *ME*, we have an axiom in the form of $ME(a)$ stating that the type of *a* is *ME.*

AR_{11}(*Completeness of the enumeration*) An enumeration type only contains the listed literal constants as its values, hence for each enumeration metaclass *ME* with literal values $a_1, a_2, \ldots, a_k$, we have an axiom in the form of $\forall x.\ ME(x) \to (x = a_1) \vee (x = a_2) \vee \ldots \vee (x = a_k)$.

F. Deriving axioms from WFR

UML class diagram is insufficient for fully defining the abstract syntax of UML. In complementary, well-formedness constraints are specified in the UML documentation for restricting valid use of the language. Some of these well-formedness rules (WFR) are formally defined in OCL, which can also be specified as axioms.

AR_{12}(*Well-formedness rules*) For each WFR formally specified in OCL, we have a corresponding axiom in the first order language.

4.4. Satisfaction Relation

From the perspective of mathematic logics and the institution theory, a graphic model can also be reviewed as a mathematical structure, when a metamodel is viewed as a formal logic system. The model is a valid instance of the metamodel, if the mathematical structure satisfies the formal logic system. Let *M* be a metamodel, $\Sigma = \mathbb{S}(M)$ be the signature obtained by applying the signature mapping $\mathbb{S}$ to *M*, $\mathscr{A}_\Sigma = \mathfrak{A}_\Sigma(M)$ be the set of axioms by applying the axiom mapping to *M*. We define the satisfaction relation $\models$ as follows.

Definition 3 (Satisfaction relation) Given a metamodel *M*, we say a model *m* satisfies the metamodel *M*, or equivalently *m* is an instance of *M*, write $m \models M$, if

(1) *m* is an interpretation of the signature Σ with the following structure:
   a) a set $E^{(m)}$ that consists of model elements in *m*;
   b) for each constant symbol *c* in Σ, its interpretation $c^{(m)}$ is an element in $E^{(m)}$, i.e. $c^{(m)} \in E^{(m)}$;
   c) for each unary predicate *P* in Σ, its interpretation $P^{(m)}$ is a subset of $E^{(m)}$, i.e. $P^{(m)} \subseteq E^{(m)}$;
   d) for each binary predicate *R* in Σ, its interpretation $R^{(m)}$ is a binary relation on $E^{(m)}$, i.e. $R^{(m)} \subseteq E^{(m)} \times E^{(m)}$;

(2) *m* is a Σ-structure of $\mathscr{A}_\Sigma$, i.e. $\forall \varphi \in \mathscr{A}_\Sigma.\ m \models \varphi$.

For example, the following is the mathematical structure equivalent to the model depicted in Fig. 2.

$E = \{Woman, Person, wp\}$,
$Class = \{Woman, Person\}$, $Generalisation = \{wp\}$,
$Classifier = \{Woman, Person\}$,
$general = \{(wp, Person)\}$, $specific = \{(wp, Woman)\}$.

For the sake of space, we omit the definition of the translation of graphic representation of models to their equivalent representation in mathematical structures. The satisfaction relation $\models$ between mathematical structures and first order logic formulas is defined as usual and details are also omitted for the sake of space.

5. Well-Definedness of Metamodels

Having defined satisfaction relation between models and metamodels, one would define the consistency of a metamodel as the satiability of the metamodel. However, this does not work because the satisfaction relation does not require a model to contain instances for every metaclass. For example, Fig. 5 shows an example of a metamodel M satisfied by a model m, where $AssociationClass \in \mathbb{S}(M)$ has no interpretation in m.

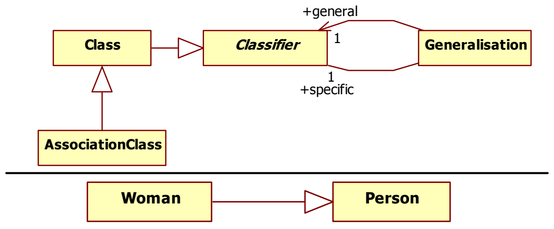


Fig. 5. Example 4 of metamodel and model

Thus, we have the following definition.

Definition 4 (Well-defined Metamodel) A metamodel M is well-defined, if there is a non-trivial model m that satisfies M, where a model is non-trivial if for every concrete metaclass C in M, m contains at least one element of C.

Whether a metamodel is well-defined can be determined according to the consistency of the logical system derived from it. The concept of metamodel's consistency is defined as follows.

Let $\mathscr{A}$ be the set of axioms obtained by applying the rules in section 4 and the following.

> AR_{11}(existence of instance): For each concrete metaclass MC in metamodel M, an sentence is derived as follows: $\exists x.\ MC(x)$.

Definition 5 (Consistency of metamodel) A metamodel M is said to be *consistent*, if the set of axioms $\mathscr{A}$ is logically consistent in the first order logic; otherwise, we say that M an *inconsistent metamodel*, i.e. when $\mathscr{A} \vdash false$.

Theorem 1. A metamodel is well-formed iff it is a consistent metamodel.

6. The prototype tool LAMBEDS

We have developed a prototype software tool called LAMBDES, which stands for a *L*ogic *A*nalyser of *M*odels/*M*etamodels *B*ased on *DE*scriptive *S*emantics. One of the tool's main functions is to automatically generate signature and axioms from a metamodel in UML according to the theory presented in Section 4. The tool is integrated with a UML modelling tool StarUML [11] and a theorem prover SPASS [12].

As shown in Fig. 6, the input to the signature generator and axiom generator components of LAMBDES is a metamodel in XMI format generated by StarUML. The output is a text file readable by SPASS. LAMBDES also generates the proof goal for the theorem prover to perform the required logical analysis of the model or metamodel. Details about other components and uses of LAMBDES for logic analysis of UML models can be found in [2].

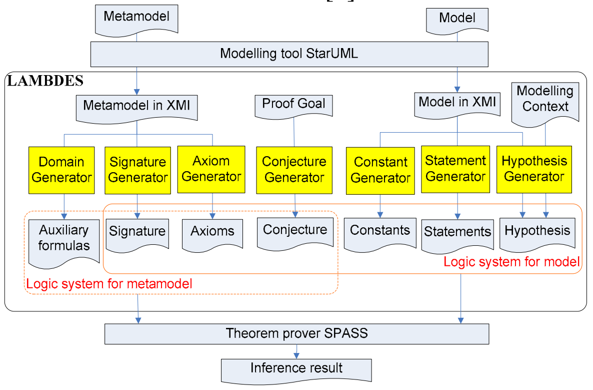
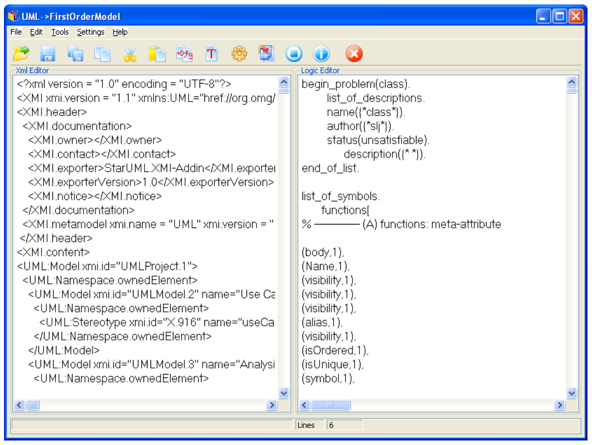


Fig. 6 Overall structure of LAMBDES



```
<?xml version = "1.0" encoding = "UTF-8"?>
<XMI xmi.version = "1.1" xmlns:UML="href://org.omg/
<XMI.header>
 <XMI.documentation>
  <XMI.owner></XMI.owner>
  <XMI.contact></XMI.contact>
  <XMI.exporter>StarUML.XMI-Addin</XMI.exporte
  <XMI.exporterVersion>1.0</XMI.exporterVersion>
  <XMI.notice></XMI.notice>
 </XMI.documentation>
 <XMI.metamodel xmi.name = "UML" xmi.version = "
</XMI.header>
<XMI.content>
<UML:Model xmi.id="UMLProject.1">
 <UML:Namespace.ownedElement>
  <UML:Model xmi.id="UMLModel.2" name="Use Ca
   <UML:Namespace.ownedElement>
    <UML:Stereotype xmi.id="X.916" name="useCa
   </UML:Namespace.ownedElement>
  </UML:Model>
  <UML:Model xmi.id="UMLModel.3" name="Analysi
   <UML:Namespace.ownedElement>
```

```
begin_problem(class).
        list_of_descriptions.
        name({*class*}).
        author({*slj*}).
        status(unsatisfiable).
            description({* *}).
end_of_list.

list_of_symbols.
      functions[
% ———— (A) functions: meta-attribute

(body,1),
(Name,1),
(visibility,1),
(visibility,1),
(visibility,1),
(alias,1),
(visibility,1),
(isOrdered,1),
(isUnique,1),
(symbol,1),
```

Fig. 7 Screen snapshot of LAMBDES

Fig. 7 shows a snapshot of the tool's interface, where *XMI editor* on the left displays the input XMI file and *Logic editor* on the right displays the generated first order system in SPASS input format.

7. Case studies

Using the prototype tool LAMBDES, we applied the proposed method to analyse two metamodels: the UML 2.0 metamodel [8] and an AspectJ profile [13]. Table 2 summarises the scales of the metamodels in terms of the numbers of various types of symbols in the signature and the numbers of various types of axioms.

7.1. UML 2.0 metamodel

Evidently, our method can be applied to the full-fledged UML. In the case study, we investigated the UML 2.0 metamodel defined in the *Classes*, *Common Behaviours*, *Interactions* and *State Machines* packages. These four packages are selected because they define the commonly used UML diagrams, covering both the

Table 2. Summary of UML 2.0 and AspectJ metamodels

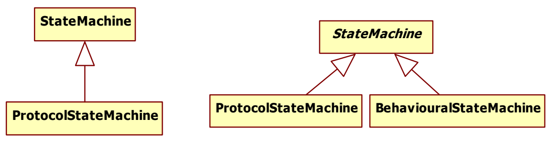
	Types	UML	AspectJ
Signature	Unary predicates	126	31
	Binary predicate	255	12
	Functions	58	11
	Constants	46	7
	Total	485	61
Axioms	Completeness of classification	1	1
	Disjointness of classification	4851	300
	Implication of inheritance	133	26
	Completeness of specialisations	42	7
	Types of binary predicates	255	12
	Domain and range of functions	57	11
	Multiplicity	222	18
	Enumeration metaclasses	196	18
	Total	*6119*	*461*

static and dynamic language facilities.

In the case study, we analysed the consistency and completeness of the metamodel. Two types of problems were identified, which are incompleteness and inconsistency problems. Each problem is resolved by modifications to the metamodel and finally we obtained a consistent and complete metamodel.

An example of incompleteness in UML 2.0 metamodel is the missing definition of enumeration metaclasses. In the metamodel, data types of meta-attributes are either enumeration types e.g. *VisibilityKind*, or primitive types e.g. *String* and *Boolean*. The enumeration types are defined in the metamodel, while the primitive types are used in the metamodel without definition. This is contradicted to the statement in [8] that "*each metaclass is completely described*". This problem is resolved by adding the primitive types as metaclasses to the metamodel.

The inconsistency of a metamodel can be detected in the first order logic by proving *false* from the generated axioms.



(A) Original metamodel (B) Modified metamodel

Fig. 8 A fragment of the metamodel for State Machine

For example, Fig. 8(A) depicts is a part of Figure 15.5 in UML 2.0 specification [8] of state machine models, where StateMachine is defined as a concrete metaclass specialised by a concrete metaclass ProtocolStateMachine. A contradiction is inferred from the axioms (1) (2) (3) below, which are derived from Fig. 8(A) by applying AR 2, AR 3 and AR 4, respectively.

$$\forall x.\ ProtocolStateMachine(x) \rightarrow \neg\, StateMachine(x) \quad (1)$$

$$\forall x.\ ProtocolStateMachine(x) \rightarrow StateMachine(x) \quad (2)$$

$$\forall x.\ StateMachine(x) \rightarrow ProtocolStateMachine(x) \quad (3)$$

This inconsistency occurs because a concrete metaclass is specialised by other metaclasses, thus violates the 'strict metamodelling' principle.

Table 1. Detected inconsistencies in UML 2.0 metamodel

Package	Super-metaclasses	Sub-metaclasses
Classes	InstanceSpecification	EnumerationLiteral
	Class	AssociationClass
	Association	AssociationClass
	DataType	PrimitiveType
	Abstraction	Realisation
	Realisation	Substitution
	Dependency	Usage
Common behaviours	OpaqueBehaviour	FunctionBehaviour
	Constraint	IntervalConstraint
	IntervalConstraint	TimeConstraint
	Class	Behaviour
Interactions	CombinedFragment	ConsiderIgnoreFragment
	InteractionUse	PartDecomposition
State machines	Transition	ProtocolTransition
	State	FinalState
	StateMachine	ProtocolStateMachine

Sixteen such inconsistencies were discovered in our case study as summarised in Table 1. To resolve this type of problems, non-leaf nodes in the inheritance hierarchy must be changed into abstract metaclasses. For example, *StateMachine* is modified as an abstract metaclass specialised by *BehaviouralStateMachine* and *ProtocolStateMachine*, as shown in Fig 8(B).

Another error found in our case study is that *OccurenceSpecification* is specified inconsistently as abstract and concrete metaclass in different diagrams. This error has been corrected in UML 2.1 [3].

7.2. AspectJ profile

UML provides extension mechanism so that a metamodel can be tailored for different platforms or domains. Stereotypes that extend metaclasses comprise a profile, which becomes a new package of metamodel. In the paradigm of model-driven architecture, one can use the extension mechanism to define profiles as metamodels of PSM. However, as pointed out in [14], question remains if the different metamodels as UML profiles are consistent or not. Our method can be applied on profile metamodels to derive axioms and verify their consistency. A case study was made on an AspectJ metamodel proposed in [13]. It is designed to provide aspect-oriented modelling facilities.

The profile enables to specify a PSM, i.e. a model specific to the Java and AspectJ platform. Same as in the case study of UML 2.0, we detected both incompleteness and inconsistency in the metamodel.

As a metamodel, the profile is found incomplete in the sense that the metaclasses *Property*, *Type*, *Operation*, *Boolean*, *InterfaceRealisation*, *Generalisation* and *String* are used as data types of metaattributes without declaration. This incompleteness is acceptable because profiles are supposed to be used together with the original metamodel rather than self-contained.

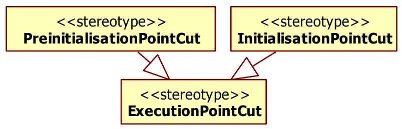


Fig. 9 Segment of AspectJ profile [13]

Two inconsistencies are detected in the AspectJ metamodel. One is because the concrete stereotype *ExecutionPointCut* is specialised by two other metaclasses, as shown in Fig. 9. Another contradiction is inferred from the axioms (4) (5) (6) below, where (4) is derived by applying AR 2, and (5) and (6) by AR 6.

$$\forall x.PointCutConjunction(x) \rightarrow \neg PointCutDisjunction(x) \quad (4)$$

$$\forall x.PointCut(x) \wedge composee(x, y) \rightarrow PointCutConjunction(x) \quad (5)$$

$$\forall x.PointCut(x) \wedge composee(x, y) \rightarrow PointCutDisjunction(x) \quad (6)$$

This inconsistency occurs because both associations from *PointCut* to *PointCutConjunction* and from *PointCut* to *PointCutDisjunction* have an end named as *compose*. An end name enables navigation between elements. Thus, the identical end name from the same node but leading to different ends causes ambiguity in the direction of navigation. To resolve this problem, one of the association ends is renamed.

8. Conclusion

Based on our previous work that formalises the semantics of UML models in FOL, in this paper we proposed a method for formalising the semantics of metamodels in UML by formally defining the signature and axiom mappings. A prototype tool has been implemented to automatically translate metamodels into first order logic systems in SPASS format. Case studies on UML 2.0 metamodel and the AspectJ profile. Demonstrate that, through the formalisation of the semantics of metamodels, the logic properties of metamodels such as consistency and completeness can be verified. Our method is effective in detecting inconsistency and incompleteness errors in metamodels.

Using the tool LAMBDES that implements the signature rules and axiom rules, we have obtained the formal semantics of UML 2.0 metamodel defined in four packages, which comprises the main part of UML. As shown in [2], this enables us to proof that a UML model is an instance of a metamodel by demonstrate that the model satisfies the axioms generated from the metamodel and also additional consistency constraints, if any.

In future work, we will incorporate the full set of well-formedness rules defined in the UML documentation into the formal metamodel, which needs manual work to translate OCL formulas to the format of SPASS. The combination of the formal metamodel and well-formedness rules will provide a complete formal language specification for UML. We also plan to apply the method of deriving descriptive semantics from M_1 level and axioms from M_2 level to the four-level metamodelling framework of UML language.

The language extension mechanisms used in the UML metamodel, especially subset and union, are investigated in [15]. The authors concluded that 'property redefinitions are not safe while package merge does not influence the relationship between model elements. It is interesting to see how such conclusions can be formally derived from formal semantics of metamodel in our framework. It is worth noting that our framework is consistent with Goguen and Burstall's theory of institution [1], thus the institution theory about the integration of multiple logics could be applied.

References

1. Goguen, J.A. and R.M. Burstall, *Institutions: Abstract Model Theory for Specification and Programming.* Journal of ACM, 1992. **39**: p. 95-146.
2. Shan, L. and H. Zhu, *A Formal Descriptive Semantics of UML*, in *Proc. of 10th Int'l Conf. on Formal Engineering Methods (ICFEM 2008)*, Springer. p. 375-396.
3. OMG, *Unified Modeling Language: Superstructure version 2.1.1.* 2007, Object Management Group.
4. Berardi, D., A. Cal, and D. Calvanese, *Reasoning on UML class diagrams.* Artificial Intelligence, 2005. **168**(1): p. 70 - 118.
5. Paige, R.F. and J.S. Ostroff, *Metamodelling and Conformance Checking with PVS*, in *Proc. of 4th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*. Springer. p. 2 - 16.
6. Seidewitz, E., *What models mean.* IEEE Software, 2003. **20**(5): p. 26 - 31.
7. OMG, *Unified Modeling Language: Infrastructure version 2.0.* 2006, Object Management Group.
8. OMG, *Unified Modeling Language: Superstructure version 2.0.* 2005, Object Management Group.
9. Atkinson, C., *Meta-Modeling for Distributed Object Environments*, in *Proc. of 1st Int'l Conf. on Enterprise Distributed Object Computing*. 1997. p. 90-101.
10. Atkinson, C. and T. Kuhne, *Rearchitecting the UML Infrastructure.* ACM Transactions on Modeling and Computer Simulation, 2002. **12**(4): p. 290 – 321.
11. *StarUML.* http://staruml.sourceforge.net/en/ [accessed on Sept. 15, 2008].
12. *SPASS.* http://www.spass-prover.org 2008 [accessed on Sept. 15, 2008].
13. Evermann, J., *A Meta-Level Specification and Profile for AspectJ in UML.* Journal of Object Technology, 2007 **6**(7): p. 27 – 49.
14. Thomas, D., *MDA: Revenge of the Modelers or UML Utopia?* IEEE Software, 2004: p. 15 - 17.
15. Alanen, M. and I. Porres, *A metamodeling language supporting subset and union properties.* Software and System Modeling, 2008. **7**(1): p. 103-124.