A Formal Descriptive Semantics of UML

Lijun Shan[1] and Hong Zhu[2]

[1]Dept of Computer Science, National Univ. of Defence Tech, Changsha, 410073, China
Email: lijunshan@brookes.ac.uk
[2]Department of Computing, Oxford Brookes University, Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk

Abstract. This paper proposes a novel approach to the formal definition of UML semantics. We distinguish descriptive semantics from functional semantics of modelling languages. The former defines which system is an instance of a model while the later defines the basic concepts underlying the models. In this paper, the descriptive semantics of class diagram, interaction diagram and state machine diagram are defined by first order logic formulas. A translation tool is implemented and integrated with the theorem prover SPASS to enable automated reasoning about models. The formalisation and reasoning of models is then applied to model consistency checking.

1. Introduction

With the rapid development of model-driven software development, concerns have been expressed on the semantics of modelling languages such as UML. It is widely recognised that a clear and rigorous semantics of UML is indispensable for rigorous modelling. Unfortunately, in spite of the numerous efforts in the past decade, formal specification of UML has not been satisfactory. This paper proposes a novel approach to defining formal semantics of modelling languages. It is applied to UML class diagram, interaction diagram and state machine diagram. The usefulness of the approach is demonstrated by its implementation in an automated tool and its application to model consistency checking.

The paper is organised as follows. Section 2 describes the proposed approach and discusses the related work. Section 3 elaborates our approach by a formal definition of UML class diagram. Section 4 further discusses how to deal with multiple views defined by separate metamodels and illustrates our approach through defining the semantics of interaction diagram and state machine. Section 5 applies the formal semantics to model consistency checking. Section 6 presents an automated tool, which translates UML models into first order logic and uses a theorem prover SPASS [1] to reason about models. Section 7 concludes the paper and discusses future work.

2. Proposed approach

2.1. Basic concepts

As Seidewitz pointed out [2], a software model, like models in any other scientific disciplines, is '*a set of statements about some system under study*', where statements are expressions that can be evaluated to a truth value with respect to the modelled systems. Further, Seidewitz stated that a model's meaning has two aspects. One is the model's relationship to the things being modelled. This meaning is implied when saying 'this model means that the Java program must contain these classes'. In this sense, a model is mapped to a collection of systems in a subject domain. By subject domain, we mean a set of systems that a modelling language intends to model, e.g. the collection of Java software systems. Another example of subject domain is the collection of real world systems described with OO concepts. In these subject domains, the truth of a statement like 'a system contains these classes' can be judged.

The other aspect of models' meaning is about the functions and properties of systems being modelled. This meaning is indicated when saying 'an inheritance relation means that every instance of the subclass is also an instance of the superclass'. On this aspect, the meaning of a model is concerned with the basic concepts such as *what is a class*, and their properties and behaviours such as *how the instances of a class behave*. Semantics on this aspect determines the functions of the systems that satisfy a model, and hence whether two models are functionally equivalent even if they look different. To distinguish these two aspects of meanings of models, we call the former *descriptive semantics* and the later *functional semantics*.

From this point of view, we can examine the weakness in the definition of UML semantics. In the UML specification [3], the 'semantics' sections explain properties and structure of each metaclass. Little has been said about how a model is mapped to a collection of systems, or equivalently, how to judge whether a system satisfies a model. Take a simple class diagram that contains one and only one class node labelled with identifier *A* as an example. It can be interpreted in any of the following ways.

– *there is **only one** class in the system and it is **named A**,*
– *there is **at least one** class **named A** in the system (which may have other classes),*
– *there is **only one** class in the system and **its name does not matter**,*
– *there is **at least one** class in the system and **its name does not matter**.*

The official UML documentation does not specify which interpretation of this simplest class diagram is correct. As Kent et al pointed out, a UML model 'typically has more than one possible implementation', and such 'underspecification' must be reflected by explicit definitions of the semantics [4]. However, formalisation of UML descriptive semantics is difficult due to the following reasons.

First, UML is not only for modelling software systems, but also for modelling real world systems, organisations and business processes. Any domain described with OO concepts can be a subject domain. This feature enables UML to bridge the gap between problem domains in the real world and the computation domain, and to model different problem fields including software, hardware, business process, etc. A formal definition of UML semantics must enable such multiple interpretations.

Second, when the full-fledged UML is considered, even the mapping from UML models to systems in a fixed subject domain is non-trivial due to the large set of language elements with complicated interrelations. It is also recognised by many researchers that the official definition of UML contains errors, hence, it evolves rapidly.

Third, UML employs the multiple view principle of modelling. A large number of different types of diagrams can be drawn to model a system from different perspectives. Each type of diagram is defined by one metamodel. These metamodels are interrelated through references and inheritances between metaclasses. The connections between metamodels further complicate the semantics of the language and also cause a potential serious problem of model inconsistency. A formal definition of UML semantics must be able to deal with such cross references between metamodels.

Another major cause of difficulty comes from the abstraction and under-specification nature of models [4]. UML is intended to be used in different stages of software engineering to describe systems at different levels of abstraction. For example, a model produced at requirements stage should be more abstract than a model built at design stage. The formalisation of UML semantics must reflect the use of the language at different levels of abstraction.

Finally, one of the most important features of UML language is its flexibility. This is achieved by at least two mechanisms. One is the extension mechanism with which new metaclasses can be introduced through the definition of profiles. The other is the under-definition of language elements.

2.2. Related work

Addressing the underspecification and ambiguity in UML's semantics, remarkable efforts have been made in the past decade to formalise UML semantics. As far as we know, all of them are about the functional semantics or aim at 'a deeper understanding of OO concepts' [5]. The following proposals are among the most well-known.

On the formalisation of class diagram, which is considered the most important type of diagrams in UML, a number of proposals have been advanced. The work by Evans *et al*. defines classifier, association, generalisation and attribute etc. in Z schemas [5]. Relations between objects and classifiers are specified as axioms. Diagrammatical transformation rules are defined as deduction rules to prove properties of UML models. See [6] for a survey of different approaches to formalising class diagram with Z or Object-Z. First order logic (FOL) and description logics (DLs) are used to formalise class diagram [7]. By encoding UML class diagrams in DL knowledge bases, DL reasoning systems can be used to reason about class diagrams. Formalisation of other types of diagrams has also been investigated, especially on state machine diagram. In [8], a rule-based operational semantics of state machine is proposed based on transition systems. Another work on operational semantics of state machine is reported in [9].

Great efforts have been made on formalising different diagrams in one semantic framework. Considering the semantics of a UML model as a set of acceptable structured process, the authors of [10] map class diagrams and state machines into algebraic specifications in Casl-ltl [11]. Another work aiming at integrated semantics of class diagram, object diagram and state machine diagrams is based on graph transformation [12].

To bridge the gap between UML and formal methods, the extensibility mechanism of UML *profile* is used to define specialisations of UML. In [13], a profile UML-B is designed so that the semantics of specialised UML entities is defined via a translation into B. In [14], an integrated formal method combining the process algebra CSP with the specification language Object-Z is used as the intermediate specification language to link UML and Java. A UML profile for CSP-OZ is designed with the aim of generating part of the CSP-OZ specifications from the specialised UML models.

The above existing methods define the semantics of UML by mapping models into a specific semantic domain, such as labelled transition systems, or OO software systems specified in a formal notation such as Z. The properties of OO systems are specified as axioms and used to reason about UML models. In other words, they mostly addressed the functional semantics of UML. Each method focuses on certain properties of OO systems, hence a certain subset of UML is formalised. However, it is hard to see how these approaches could work either alone or together for the full-fledged UML. Most importantly, the ambiguity in descriptive semantics is not addressed in these works. Instead, their semantics formalisations are based on explicit or implicit assumption on the descriptive semantics. Automation of translating UML models to formal specifications to facilitate automated reasoning of UML models has not been achieved in the existing methods.

2.3. Outline of the proposed approach

In this paper, we take a novel approach to formalising the semantics of UML models by explicitly distinguishing descriptive semantics from functional semantics and specifying them separately.

First, the descriptive semantics is defined through a mapping from UML models to a set of first order logic statements, which are constructed from a set of predicates and functions via logic connectives and quantifiers. Predicates and functions represent the basic concepts of the modelling language. For example, predicate *Class*(*x*) is defined to represent the concept class in UML. Interrelations between basic concepts as specified in UML metamodels are characterised by a set of axioms, called *axioms of descriptive semantics* in the sequel. The satisfaction of a model by a system is defined as the evaluation of the truth of the statements in the context of the system, provided that how to evaluate these predicates and functions is known.

Second, the functional semantics of UML is defined for the predicates and functions. The properties and dynamic behaviours of modelled systems can be characterised by a set of axioms called *axioms of functional semantics*. Thus, the functional semantics of a model determines the functions and runtime behaviours of the systems that satisfy a model.

Formally, we have the following structure of semantics for a modelling language.

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

– A signature ***Sig***, which defines a formal logic system;
– A set $\boldsymbol{Axm}_D$ of axioms about the descriptive semantics, which is in the formal logic system defined by ***Sig***;
– A set $\boldsymbol{Axm}_F$ of axioms about the functional semantics, which is also in the formal logic systems defined by ***Sig***;
– A mapping ***F*** from models to a set of formulas in the formal logic system defined by ***Sig***. The formulas are the statements for the descriptive semantics of the model;
– A mapping ***H*** from models to a set of formulas in the formal logic system defined by ***Sig***. The formulas represent the hypothesis about the context in which the descriptive semantics is interpreted. □

In the above definition, the signature defines the symbols that can be used in the formulas and axioms. The evaluation of a first order logic formula is as usual.

Definition 2. (Semantics of a model)

Given a semantics definition of a modelling language as in Definition 1, the semantics of a model M under the hypothesis $\boldsymbol{H}$, written $Sem_{\boldsymbol{H}}(M)$, is defined as follows.

$$Sem_{\boldsymbol{H}}(M) = \boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M)$$

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

$$DesSem_{\boldsymbol{H}}(M) = \boldsymbol{Axm}_D \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M) \qquad \square$$

Given a semantics definition of a modelling language in the above framework, reasoning about the properties of a model can be defined as logical inference as follows.

Definition 3. (Properties of a model)
Let $Sem_{\boldsymbol{H}}(M)$ be the semantics of a model M. M has a property P (represented as a formula in the logic system defined by $\boldsymbol{Sig}$) under the semantics definition $Sem_{\boldsymbol{H}}(M)$ and the hypothesis $\boldsymbol{H}$, if and only if $\boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M) \vdash P$ in the formal logic system. Similarly, we say that M has a property P in descriptive semantics, if and only if $\boldsymbol{Axm}_D \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M) \vdash P$ in the formal logic system. □

A key concept of the semantics of modelling languages is the satisfaction of a model by a system. Before defining this concept, let's first define the notion of subject domain and the interpretation of a formal logic in a subject domain.

Definition 4. (Subject domain)
A subject domain $\boldsymbol{Dom}$ of signature $\boldsymbol{Sig}$ with an interpretation Eva is a triple $<D, \boldsymbol{Sig}, Eva>$, where D is a collection of systems on which the formulas of the logic system defined by $\boldsymbol{Sig}$ can be evaluated according to a specific evaluation rule Eva. The value of a formula f evaluated according to the rule Eva in the context of system $s \in D$, written as $Eva(f, s)$, is called the *interpretation* of the formula f in s. We write $s \models_{Eva} f$, if a formula f is evaluated to true in a system $s \in D$, i.e. $s \models_{Eva} f$ iff $Eva(f, s) = true$. □

When there is no risk of confusion, we will omit the subscript in $s \models_{Eva} f$. For a set F of formulas, we write $s \models F$ to denote that for all f in F, $s \models f$.

Definition 5. (Satisfaction of a model)
Let $\boldsymbol{Sig}$ be a given signature and $\boldsymbol{Dom}$ a subject domain of $\boldsymbol{Sig}$. A system s in D *satisfies* a model M according to a semantic definition $Sem_{\boldsymbol{H}}(M)$ if $s \models Sem_{\boldsymbol{H}}(M)$, i.e. for all formulas f in $Sem_{\boldsymbol{H}}(M)$, $s \models f$. □

In the remainder of the paper, we will elaborate the approach by defining the descriptive semantics of UML class diagram, interaction diagram and state machine diagram. We will also demonstrate the application of the semantic definition to model consistency checking.

3. Descriptive Semantics of Class Diagram

3.1. Metamodel

Fig. 1 shows the simplified metamodel on which our formal definition of the descriptive semantics of UML class diagrams is based.

3.2. Derivation of signature

Given a metamodel, the signature of a formal logic system can be induced by applying the derivation rules defined as follows.

– *Signature Rule* 1: *Unary predicates*. For each metaclass named *MC* in the metamodel, we define a unary atomic predicate *MC*(*x*).

– *Signature Rule* 2: *Binary predicates*. For each association named *MA* between two metaclasses *X* and *Y* in the metamodel, a binary predicates *MA*(*x*, *y*) is defined to represent the relation between elements of type *X* and *Y*.

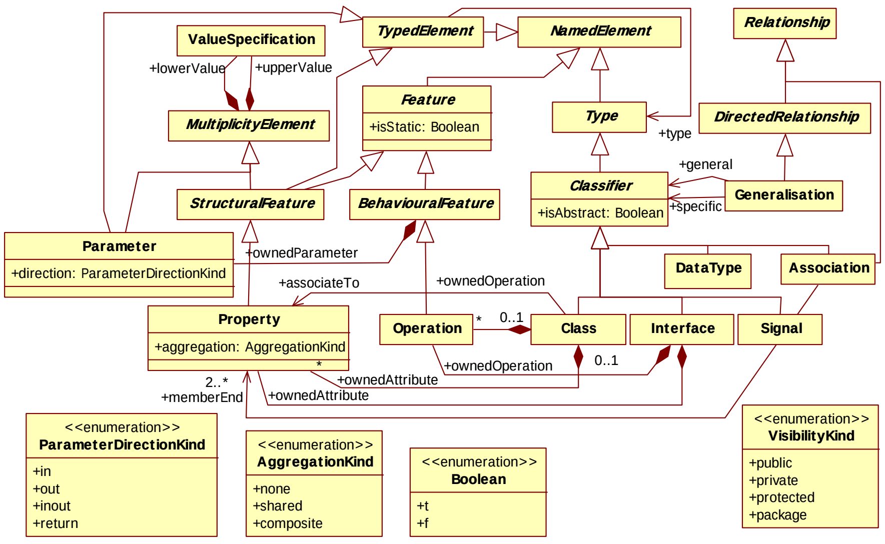


Fig. 1 Metamodel of Class Diagram

A predicate *MC*(*x*) means that element *x* is of type *MC*. For example, a unary predicate *Class*(*x*) is defined to represent the metaclass *Class* in Fig. 1. A binary predicate *MA*(*x*,*y*) means that elements *x* and *y* are related by the relation *MA*. For example, a binary predicate *specific*(*x*, *y*) is defined to represent the association named *specific* from metaclass *Generalisation* to *Classifier* in Fig. 1.

Constants and functions in the signature are also derived from the metamodel.

– *Signature Rule* 3: *Constants*. For each enumeration value *EV* given in an enumeration metaclass *ME* in the metamodel, a constant *EV* is defined.

For example, two enumeration values *t* and *f* are defined in the enumeration metaclass *Boolean* in Fig. 1. Thus, two constants *t* and *f* are defined.

– *Signature Rule* 4: *Functions*. For each meta-attribute *MAttr* of type *MT* in a metaclass *MC*, a function *MAttr* is defined with domain *MC* and range *MT*.

For example, in Fig. 1, metaclass *Classifier* has an attribute *isAbstract* of type *Boolean*. Thus, a function *isAbstract* is defined on domain *Classifier* and range *Boolean*. A statement *isAbstract*(*x*)=*t* means element *x*' property on *isAbstract* is *t*.

Table 2 summarises the constants representing enumeration values and their types, as well as the functions derived from the metamodel in Fig. 1 . These functions are partial, i.e. they can be undefined on some elements in a model.

The interpretation of the functions and predicates must be defined in the context of a subject domain. Take the set of C++ programs as an example of subject domain.

Given a C++ program, the predicate *Class*(*User*) is true if *User* is a class in the program. The statement *isAbstract*(*User*) is true when the class *User* in the program is declared to be abstract. In this paper, we leave the definition of the interpretation open so that a model can be interpreted in different subject domains.

Table 1 Predicates for Class Diagram

Predicate	**Meaning**
ValueSpecification(x)	x has type ValueSpecification
MultiplicityElement(x)	x has type MultiplicityElement
StructuralFeature(x)	x has type StructuralFeature
TypedElement(x)	x has type TypedElement
Feature(x)	x has type Feature
BehaviouralFeature(x)	x has type BehaviouralFeature
NamedElement(x)	x has type NamedElement
Type(x)	x has type Type
Classifier(x)	x has type Classifier
Relationship(x)	x has type Relationship
DirectedRelationship(x)	x has type DirectedRelationship
Parameter(x)	x has type Parameter
Property(x)	x has type Property
Operation(x)	x has type Operation
Class(x)	x has type Class
Interface(x)	x has type Interface
Signal(x)	x has type Signal
Generalisation(x)	x has type Generalisation
Association(x)	x has type Association
DataType(x)	x has type DataType
ParameterDirectionKind(x)	x has type ParameterDirectionKind
AggregationKind(x)	x has type AggregationKind
Boolean(x)	x has type Boolean
VisibilityKind(*x*)	x has type VisibilityKind
upperValue(x, y)	the relation between x and y is upperValue
lowerValue(x, y)	the relation between x and y is lowerValue
type(x, y)	the relation between x and y is type
general(x, y)	the relation between x and y is general
specific(x, y)	the relation between x and y is specific
ownedParameter(x, y)	the relation between x and y is ownedParameter
ownedAttribute(x, y)	the relation between x and y is ownedAttribute
ownedOperation(x, y)	the relation between x and y is ownedOperation
associateTo(x, y)	the relation between x and y is AssociateTo
memberEnd(x, y)	the relation between x and y is memberEnd

Table 2 Functions and Constants for Class Diagrams

Function	**Domain**	**Range**	
		Type	**Values**
isStatic	Feature	Boolean	f, t
visibility	NamedElement	VisibilityKind	public, private, protected, package
isAbstract	Classifier	Boolean	f, t
direction	Parameter	ParameterDirectionKind	in, out, inout, return
aggregation	Property	AggregationKind	shared, composite, none

3.3. Axioms

A UML metamodel is a model that defines the abstract syntax of UML diagrams. It can also be regarded as a collection of statements that evaluate to truth values on UML models. A UML model is syntactically valid only if all these statements are true. Thus, they are axioms on the formal systems representing descriptive semantics of models. We identified the following five groups of axioms.

A. Inheritance hierarchy on metaclasses

In a metamodel, concrete metaclasses define types of model elements, while abstract metaclasses define common features of concrete metaclasses. These common features may be specialised by concrete metaclasses. In the sequel, we call a type defined by a concrete metaclass a *concrete type*, and a type defined by an abstract metaclass an *abstract type*. Each element has exactly one concrete type, but may belong to a number of abstract types.

– *Axiom Rule* 1: *Logical implication of inheritance*. For an inheritance relation from metaclass *MA* to *MB*, we have an axiom in the form of $\forall x.\ MA(x) \rightarrow MB(x)$

For example, from the inheritance relation from *Class* to *Classifier* in Fig. 1, an axiom is derived to state that if an element has *Class* as its type, it also belongs to the type *Classifier*. Formally, $\forall x.\ Class(x) \rightarrow Classifier(x)$.

– *Axiom Rule* 2: *Completeness of specialisations*. Let *MA* be a metaclass in a metamodel and $MB_1, MB_2, \ldots, MB_k$ be the set of metaclasses specialising *MA*. We have an axiom in the form of $\forall x.\ MA(x) \rightarrow MB_1(x) \vee MB_2(x) \vee \ldots \vee MB_k(x)$

For example, the following axiom is derived from the metamodel in Fig. 1. It states that if an element has *Classifier* as its type, it must belong to one of the 5 sub-types: *Association*, *DataType*, *Class*, *Interface* or *Signal*.

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

– *Axiom Rule* 3: *Uniqueness of element classification*. Let $MC_1, MC_2, \ldots, MC_n$ 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 in the following form.

$$\forall x.\ MC_i(x) \rightarrow \neg MC_j(x)$$

For example, the following axiom states that if an element has *Property* as its concrete type, it cannot be an *Operation* at the same time.

$$\forall x.\ Property(x) \rightarrow \neg Operation(x)$$

B. Navigation between element types

Let *MA* be an association from metaclass MC_1 to MC_2 in a metamodel. For the binary predicate $MA(x,y)$ derived from the association *MA*, the two parameters must be elements of type MC_1 and MC_2, respectively. Thus, we have the following axiom rule.

– *Axiom Rule* 4: *Types of parameters of predicates*. For each binary predicate $MA(x,y)$ derived from an association from metaclass MC_1 to MC_2 in the metamodel, we have an axiom in the following form.

$$\forall x,y.\ MA(x,y) \rightarrow MC_1(x) \wedge MC_2(y)$$

For example, the following axiom is derived from the association *general* from metaclass *Generalisation* to *Classifier* in Fig. 1. It states that if predicate $general(x,y)$ is true, *x* must belong to the type *Generalisation* and *y* must belong to *Classifier*.

$$\forall x,y.\ general(x,y) \rightarrow Generalisation(x) \wedge Classifier(y)$$

Similar to binary predicates, for each function *MAttr*, we have an axiom to specify its domain and range.

– *Axiom Rule* 5: *Domain and range of functions*. For each function *MAttr* derived from a meta-attribute *MAttr* of type *MT* in a metaclass *MC*, we have an axiom in the following form.

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

For example, for the meta-attribute *isAbstract* of *Classifier* in Fig. 1, the following axiom is derived. It states that if function *isAbstract* is applied on an element of the type *Classifier*, the value of the function must belong to *Boolean*.

$$\forall x,y.\ Classifier(x) \wedge (isAbstract(x) = y) \rightarrow Boolean(y)$$

C. Well-formedness constraints

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

– *Axiom Rule* 6: *Well-formedness rules*. For each WFR formally specified in OCL, we have a corresponding axiom in the first order language.

For example, a WFR in UML document is "*Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.*" Thus, we have the following axiom.

$$\forall x, y.\ Inherit(x, y) \rightarrow \neg\ Inherit(y, x)$$

where *Inherit*(x,y) is a binary predicate introduced to simplify the specification of the axiom. It is formally defined by the following two formulas.

$$\forall x,y.\ Generalisation(z) \wedge specific(z, x) \wedge general(z, y) \rightarrow Inherit(x, y)$$

$$\forall x, y, z.\ Inherit(x, y) \wedge Inherit(y, z) \rightarrow Inherit(x, z)$$

Some well-formedness rules are informally defined in the UML documentation. They cannot be easily specified in first order logic. For example, a rule for *MultiplicityElement* is '*if a non-literal ValueSpecification is used for the lower or upper bound, then evaluating that specification must not have side effects*'. It cannot be formally specified as an axiom.

D. Definition of enumeration values

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

– *Axiom Rule* 7: *Distinguishability of the literal constants*. For each pair of different literal values a and b defined in an enumeration type, we have an axiom in the form of $a \neq b$.

For example, the metaclass *Boolean* defines two literal values t and f. Thus, we have the axiom $t \neq f$.

– *Axiom Rule* 8: *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*.

For example, for the Boolean values t and f, we have the following two axioms.

$$Boolean(t),\ \ Boolean(f).$$

– *Axiom Rule* 9: *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, …, a_k, we have an axiom in the form of

$$\forall x.\ ME(x) \rightarrow (x = a_1) \vee (x = a_2) \vee \ldots \vee (x = a_k)$$

For example, we have the following axiom for the *Boolean* metaclass.

$$\forall x.\ Boolean(x) \text{ -> } (x = t) \vee (x = f)$$

3.4. Translating models into first order logic formulas

This subsection shows how to translate diagrammatic models to first order logic formulas. We will use the class diagram in Fig. 2 as an example.

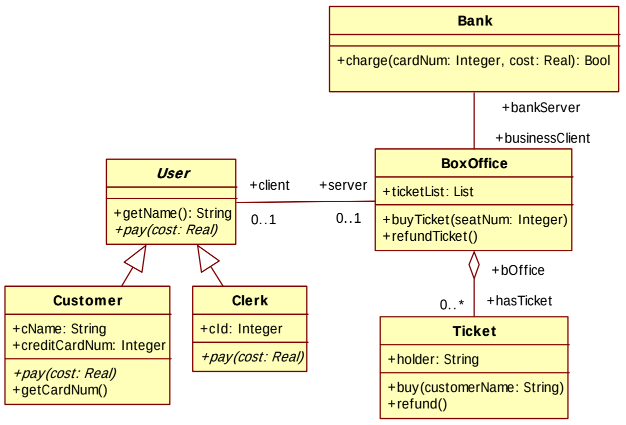


Fig. 2 Class Diagram in the Model Ticketing Office

A. Semantics mapping F_M

For each class diagram, the following rules are applied to generate formulas.

– *Translation Rule* 1: *Classification of elements*. For each identifier *id* of concrete type *MC*, a formula in the form of *MC*(*id*) is generated.

By applying this rule to every element in a diagram, a set of formulas are generated to declare the classification of the identifiers. For example, the following formulas are among those generated from the class nodes in Fig. 2.

$$Class(User),\ Class(Bank),\ Class(BoxOffice).$$

Similarly, formulas are generated by applying other unary predicates that represent concrete types such as *Property*, *Association*, *Operation*, etc.

– *Translation Rule* 2: *Properties of elements*. For each element *a* in the model and every applicable function *MAttr* that represents a meta-attribute, a formula in the form of *MAttr*(*a*)=*v* is generated, where *v* is *a*'s value on the property.

For example, *Clerk* in Fig. 2 is a concrete class. Therefore, the following formula is generated, which states that the value of function *isAbstract* on *Clerk* is false.

$$isAbstract(Clerk) = f$$

Table 3 lists the functions applicable on each concrete type of model elements and the elements contained in the class diagram in Fig. 2. Applicable functions are derived from the metamodel according to the inheritance relation between metaclasses.

– *Translation Rule* 3: *Relationships between elements*. For each related pair (e_1, e_2) of elements in a model, a formula in the form of $R(e_1, e_2)$ is generated to specify the relationship by applying binary predicate $R(x_1, x_2)$.

For example, the class diagram in Fig. 2 depicts a generalisation relation from class

Clerk to *User*. Hence, we have the formula *specific*(*g*, *Clerk*), where *g* denotes the generalisation arrow.

Table 3 Constants representing model elements

Type	Applicable functions	Elements in Fig.2
Class	isAbstract, visibility	Bank, User, BoxOffice, Clerk, Customer, Ticket
Property	isStatic, aggregation, visibility	name, creditCardNum, ID, ticketList, holder, client, server, businessClient, bankServer, hasTicket, bOffice
Operation	isStatic, visibility	GetName, Pay, GetCardNum, Charge, BuyTicket, Refund-Ticket, Buy, Refund
Association	*isAbstract*, visibility	UserBoxoffice, BoxofficeTicket, BoxofficeBank
DataType	*isAbstract*, *visibility*	String, Integer, Bool, List
Generalisation	/	customerUser, clerkUser
Parameter	direction, visibility	cardNum, cost, seatNum
Signal	isAbstract, visibility	/
Interface	isAbstract, visibility	/
ValueSpecification	/	0, 50, 1, 200

B. Hypothesis mapping H_M

In addition to the above translation rules that are applied to all models, hypothesis rules are needed to generate formulas that represent the meanings of models in specific uses of the modelling language. Their application should be determined by users according to the situation in which a model is used. The following are some examples of such hypothesis rules.

Let $e_1, e_2, \ldots, e_k$ be the set of elements of a concrete type *MC* in a model.

– *Hypothesis Rule* 1: *Distinguishability of elements*. The hypothesis that the elements of type *MC* in the model are all different can be generated as formulas in the form of $e_i \neq e_j$, for $i \neq j \in \{1,2,\ldots,k\}$.

For example, if it is assumed that in Fig. 2 class *Clerk* is different from class *Customer*, the formula *Clerk* ≠ *Customer* is generated. This hypothesis is applicable if the model is considered as a design, thus force the programmer to implement two classes *Clerk* and *Customer* separately. However, if the model is used as a requirements specification, this hypothesis may not be necessary because a program with one class implementing both *Clerk* and *Customer* can be considered as satisfying the model.

– *Hypothesis Rule* 2: *Completeness of elements*. The hypothesis on the completeness of elements of type *MC* can be generated as a formula in the following form.

$$\forall x.\ MC(x) \rightarrow (x = e_1) \vee (x = e_2) \vee \ldots \vee (x = e_k)$$

For example, the assumption that the model in Fig. 2 contains all classes in the modelled system can be specified as follows.

$$\forall x.\ Class(x) \rightarrow (x = Ticket) \vee (x = Clerk) \vee (x = Customer) \\ \vee (x = User) \vee (x = Bank) \vee (x = BoxOffice)$$

This hypothesis on the completeness of classes is applicable when a model represents a system in reverse engineering or as a detailed design. However, when a model is used as requirements specification, an implementation of the system may introduce additional classes and still be regarded as satisfying the requirements. In this case, this hypothesis is not applicable.

Similarly, we have the following hypothesis on the completeness of relations. Let $R(x_1, x_2)$ be a binary predicate, $R(e_{1,1}, e_{1,2})$, $R(e_{2,1}, e_{2,2})$, …, $R(e_{n,1}, e_{n,2})$ be the set of *R* relations contained in the model.

– *Hypothesis Rule* 3: *Completeness of relations*. The hypothesis on the completeness of relation R in the model can be generated as a formula in the following form.

$\forall x_1,x_2.R(x_1,x_2)\to((x_1=e_{1,1})\wedge(x_2=e_{1,2}))\vee((x_1= e_{2,1})\wedge (x_2= e_{2,2}))\vee\ldots ((x_1= e_{n,1})\wedge (x_2= e_{n,2}))$

This hypothesis assumes that all relations of a certain type are specified in the model, thus any additional relation in a system will be regarded as not satisfying the model. For example, for the model in Fig. 2, we will specify the following formula, if we believe all inheritance relations in the modelled system are depicted in the model.

$\forall x,y.\ specific(x,y)\to((x=ClerkUser)\wedge(y=Clerk))\vee((x=CustomerUser)\wedge(y= Customer))$

It is worth noting that the above hypothesis rules are just examples. They are by no means considered as complete. The point here is the flexibility of UML for different uses can be explicitly revealed through a set of optional hypothesis mappings. How hypothesis rules are related to the use of the modelling language will be an interesting practical problem for further research.

4. Semantics of Interaction and State Machine

Our approach to defining descriptive semantics is applicable on various types of UML diagrams. This section defines the descriptive semantics of interaction diagram and state machine. The same rules and process described in section 3 are applied. The only difference is that their metamodels are connected to the metamodel of class diagram. This section will focus on how to deal with such connections.

4.1. Integration of Metamodels

Fig. 3 shows a simplified metamodel of interaction diagram.

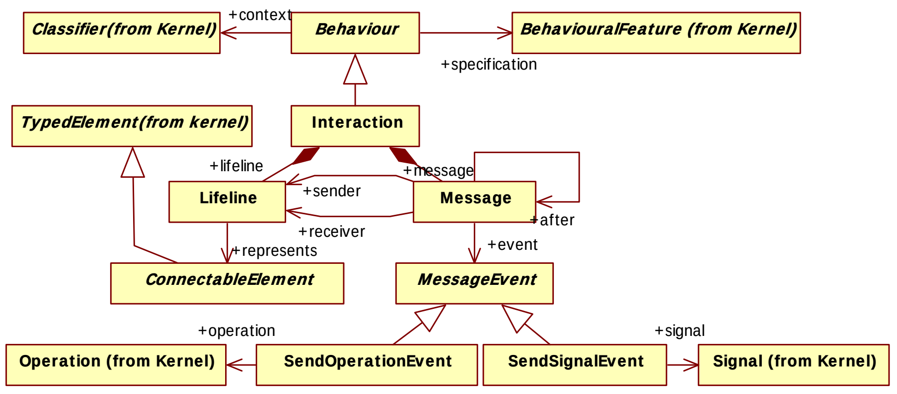


Fig. 3 Metamodel of Interaction Diagram

Metaclasses *Operation*, *Signal*, *TypedElement*, *BehaviouralFeature* and *Classifier* in Fig. 3 were defined in the metamodel of class diagram in Fig. 1 as indicated by ‘from Kernel’ after their names. They are included in this metamodel to specify the connection between the metamodels. For the associations that relate a metamodel to external metaclasses, the rules for defining predicates and axioms differ from the ordinary rules. For example, in Fig. 3, the association *operation* denotes the correspondence between *SendOperationEvent* in interaction diagram and *Operation* in class diagram. Similarly, the association *signal* denotes the correspondence between *SendSignalEvent* in interaction diagram and *Signal* in class diagram. Thus, the Signature Rule 2 is not applied on them. Instead, such correspondences are specified as axioms about the related element types. The following two axioms are derived from associations *operation* and *signal* in Fig. 3, respectively.

$$\forall x.\ SendOperationEvent(x) \rightarrow Operation(x)$$

$$\forall x.\ SendSignalEvent(x) \rightarrow Signal(x)$$

Formally, we have the following general rule for generating axioms from cross metamodel associations.

– *Axiom Rule 10: Cross metamodel association*. For each cross metamodel association from metaclass *MA* to external metaclass *MB*, we have an axiom in the form of $\forall x.\ MA(x) \rightarrow MB(x)$.

Axioms for multiple-view UML models comprise the axioms for different types of diagrams, which are separately derived from the respective metamodels. When the different sets of axioms are integrated, the axioms about '*completeness of specialisations'* have to be modified due to the overlap between the inheritance hierarchies in the different metamodels. Formally,

– *Axiom Rule* 2': *Completeness of specialisations across metamodels*. Let *MA* be a metaclass depicted in two metamodels MM_1 and MM_2. Let metaclasses MB_1, MB_2, …, MB_k be the set of metaclasses that specialise *MA* in metamodel MM_1, and MC_1, MC_2, …, MC_p be the set of metaclasses that specialise *MA* in metamodel MM_2. We have the following axiom when a model is defined by MM_1 and MM_2.

$$\forall x.\ MA(x) \rightarrow MB_1(x) \vee \ldots \vee MB_k(x) \vee MC_1(x) \vee \ldots \vee MC_p(x)$$

Take the specialisations of metaclass *TypedElement* in Fig. 1 and Fig. 3 as an example. Axiom (1) below will be derived from Fig. 1 by applying Axiom Rule 2 for defining the semantics of models that only contains class diagrams. Similarly, when a model only contains sequence diagrams, axiom (2) will be used. When the model contains both class diagrams and sequence diagrams, i.e. the models are defined by the two interrelated metamodels, axiom (3) below will be used.

$$\forall x.\ TypedElement(x) \rightarrow Parameter(x) \vee StructuralFeature(x) \tag{1}$$

$$\forall x.\ TypedElement(x) \rightarrow ConnectableElement(x) \tag{2}$$

$$\forall x.\ TypedElement(x) \rightarrow Parameter(x) \vee StructuralFeature(x) \vee ConnectableElement(x) \tag{3}$$

The signature and axioms of state machine diagrams are derived from the metamodel shown in Fig. 4 by applying the rules given in section 3 and section 4.1. Table 4 summarises the number of generated predicates, functions and axioms.

4.2. Translating diagrams into first order logic formulas

The translation rules given in section 3 are applied to sequence diagrams and state machines to generate first order logic formulas. For example, the following formulas are among those generated from the interaction diagram shown in Fig. 5 (A).

$$Message(buyTicket)\ ,\ sender(buyTicket, c).$$

Below are some of the formulas generated from the state machine in Fig. 5 (B).

$$State(available),\ \ trigger(Transition7,refund),\ source(Transition7,unavailable).$$

Totally 1459 formulas were generated from the three diagrams of the model Ticketing Office.

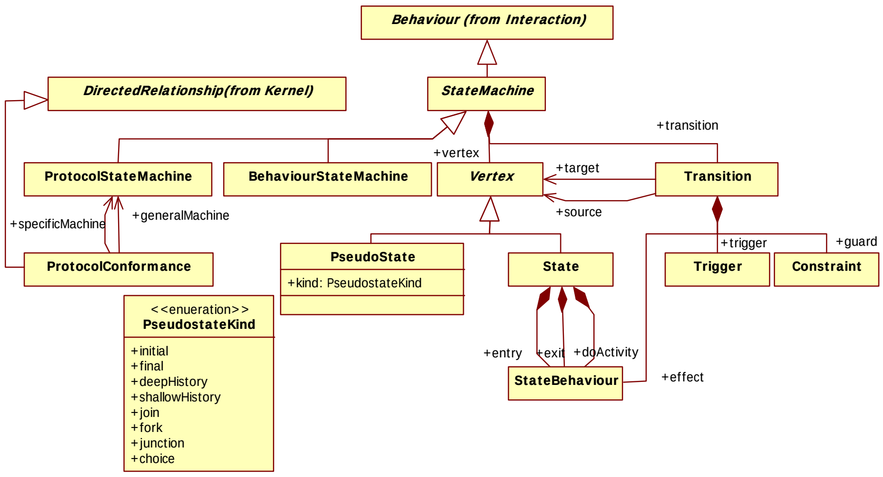
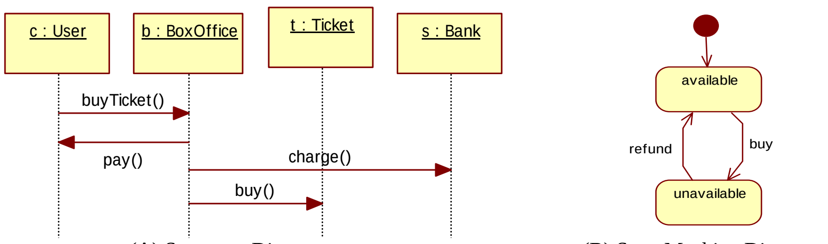


Fig. 4 Metamodel of State Machine Diagram

Table 4 Summary of the signature and axioms defined for three types of diagrams

			Class Diagram	Interaction Diagram	State Machine
Signature	Unary Predicate	Abstract metaclasses	10	3	2
		Concrete metaclasses	10	5	9
		Enumeration metaclasses	4	0	1
	Binary Predicates		10	8	12
	Functions		5	0	1
	Enumeration constants		13	0	8
Axiom	Inheritance relations		20	4	6
	Completeness of specialisation		10	3	4
	Completeness of classification		10	5	9
	Valid application of binary predicates/functions		15	8	13
	Well-formedness rules		7	1	21



(A) Sequence Diagram (B) State Machine Diagram

Fig. 5 Sequence Diagram and State Machine Diagram in the Model Ticketing Office

5. Consistency Check: An Application of Descriptive Semantics

The formal definition of UML semantics in our approach naturally facilitates reasoning about models. This section demonstrates the application of the descriptive semantics in consistency checking of models.

Aiming at rigorous modelling, great efforts have been made to define and check models' consistency [15-18], especially in the context of UML models [19-22]. With the definition of model semantics in first order logic, checking the consistency of a model is to prove that the formulas generated from the model are consistent in the context of the axioms. Moreover, additional stronger consistency constraints can also be specified in first order logic. The validity of such constraints, i.e. their consistency with the axioms, can be formally proved.

5.1. Checking consistency as logical inference

Let ***F*** be a set of formulas in a signature ***Sig***. As in first order logic, if we can deduce that $\boldsymbol{F}|\!-\!\mathit{false}$, then ***F*** is inconsistent. Thus, we have the following definition.

Definition 6. (Logical consistency)

Let $Sem_{\boldsymbol{H}}(M) = \boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M)$ be the semantics of a model *M*. Model *M* is said to be *logically inconsistent* in the semantic definition $Sem_{\boldsymbol{H}}(M)$ if $Sem_{\boldsymbol{H}}(M)|\!-\!\mathit{false}$; otherwise, we say that the model is *logically consistent*. □

It is easy to see that a logically inconsistent model is not satsifiable in a subject domain whose interpretation of formulas is consistent with the logic system.

Definition 7. (Consistent interpretation of formulas in a subject domain)

Let ***Dom***=<*D*, ***Sig***, *Eva*> be a subject domain as defined in Definition 4. The interpretation of formulas in signature ***Sig*** is consistent with first order logic if and only if for all formulas *q* and $p_1, p_2, \ldots, p_k$ that $p_1, p_2, \ldots, p_k \mid\!- q$, and for all systems *s* in *D* that $Eva(p_i, s) = \mathit{true}$ for i=1,2,…, *k*, we always have $Eva(q, s) = \mathit{true}$. □

Theorem 1. (Unsatisfiability of inconsistent model)

A model *M* that is logically inconsistent in the semantic definition $Sem_{\boldsymbol{H}}(M)$ is not satisfiable on any subject domain whose interpretation of formulas is consistent with first order logic.

Proof. We prove by contradiction. Let *M* be a logically inconsistent model, *s* be a system in a subject domain ***Dom*** that satisfies the model according to the semantic definition $Sem_{\boldsymbol{H}}(M)$. By Definition 5, for all formulas *p* in $Sem_{\boldsymbol{H}}(M)$, $s|\!=p$. By Definition 6, *M* is logically inconsistent means that $Sem_{\boldsymbol{H}}(M)|\!-\!\mathit{false}$. By the property that the interpretation of formulas in the subject domain ***Dom*** is consistent with the first order logic, it follows (Definition 8) that $s|\!=\mathit{false}$. Thus, we find a contradiction. Therefore, the theorem is true. □

In the experiment, we used SPASS theorem prover to prove that each set of the formulas generated from the three diagrams in the model Ticketing Office shown in Fig. 2 and Fig. 5 are logically consistent. Their union is also consistent. Moreover, the set of axioms for class diagrams, interaction diagrams and state machines are also proven to be logically consistent. Thus, we have the following theorem.

Theorem 2. (Consistency of the axioms in semantics definition)

The sets of axioms generated from the metamodels for class diagrams, interaction

diagrams and state machines are consistent as they are individually as well as together.
Proof. As stated above. □

We have also made various minor changes to the diagrams in the model Ticketing Office to demonstrate that some changes can lead to logically inconsistent set of formulas, thus proved the existence of inconsistent models in UML according to our semantic definition. Thus, it is feasible to check models' consistency through logic inferences based on descriptive semantics.

It is worth noting that, generally speaking, logical consistency does not guarantee that the model is satsifiable in a subject domain.

5.2. Checking consistency against additional constraints

In addition to checking the consistency of a model as described in the previous subsection, it is often desirable to check models against addition constraints. For example, the following consistency constraint has been studied in the literature [23, 24]. It states that a life line must represent an instance of a class.

$$\forall x, y, z.\ \mathit{Lifeline}(x) \wedge \mathit{represent}(x,y) \wedge \mathit{type}(y, z) \text{ -> } \mathit{Class}(z)$$

If a consistency constraint cannot be derived from the axioms, a model that is logically consistent does not necessarily satisfy the additional constraint. Thus, we have the following notion of consistency with respect to a set of constraints.

Definition 8. (Consistency w.r.t. consistency constraints)
Given a set of consistency constraints $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$, the consistency of a model M with respect to the constraints $\boldsymbol{C}$ under the semantics definition $Sem_{\boldsymbol{H}}(M)$ is the consistency of the set $\boldsymbol{U} = Sem_{\boldsymbol{H}}(M) \cup \boldsymbol{C}$ of formulas. In particular, we say that a model fails on a specific constraint c_k, if $Sem_{\boldsymbol{H}}(M)$ is consistent, but $Sem_{\boldsymbol{H}}(M) \cup \{c_k\}$ is not. □

The following are some commonly used consistency constraints.

– Message represents operation call of the message receiver [23]. Formally,

$$\forall x, y, z, u.\ \mathit{Message}(x) \wedge \mathit{event}(x,y) \wedge \mathit{SendOperationCall}(y) \wedge \mathit{receiver}(x,z) \wedge \mathit{type}(z, u) \text{–>} \mathit{ownedOperation}(u,y)$$

– The classifier of a message's sender must be associated to the classifier of the message's receiver [23]. Formally,

$$\forall x,y,z,u,v.\ \mathit{Message}(x) \wedge \mathit{sender}(x,y) \wedge \mathit{type}(y,u) \wedge \mathit{receiver}(x,z) \wedge \mathit{type}(z,v) \text{–>} \exists\, w,m,n.\ \mathit{Association}(w) \wedge \mathit{memberEnd}(w, m) \wedge \mathit{memberEnd}(w, n) \wedge \mathit{AssociateTo}(m, u) \wedge \mathit{AssociateTo}(n,v)$$

– Protocol transition refers to an operation (i.e., has a call trigger corresponding to an operation), and that operation applies to the context classifier of the state machine of the protocol transition. Formally,

$$\forall x,y,z.\ \mathit{ProtocolStateMachine}(x) \wedge \mathit{transition}(x,y) \wedge \mathit{trigger}(y,z) \wedge \mathit{context}(x,u) \text{–>} \mathit{Operation}(z) \wedge \mathit{ownedOperation}(u,z)$$

– The order of messages in interaction diagram must be consistent with the order of triggers on transitions in state machine diagram [23, 25]

$$\forall x,y,z,u.\mathit{Message}(x)\wedge\mathit{event}(x,z)\wedge\mathit{Message}(y)\wedge\mathit{event}(y,u)\wedge\mathit{after}(x,y)\text{–>}\mathit{Trigs}(z,u)$$

where *Trigs(x,y)* is an auxiliary predicate defined as follows.

$$\forall x,y,z,u,v.\ \mathit{Transition}(x) \wedge \mathit{trigger}(x,u) \wedge \mathit{target}(x,y) \wedge \mathit{Transition}(z) \wedge \mathit{trigger}(z,v) \wedge \mathit{source}(z,y) \text{–>} \mathit{Trigs}(v,u)$$

$$\forall x,y,z.\ Trigs(x,y) \wedge Trigs(y,z) \rightarrow Trigs(x,z)$$

In the above discussion, we have made an implicit assumption about the validity of the constraints. Informally, a constraint is invalid if it conflicts with the semantics axioms of the language and thus cannot be satisfied by any model. Here, we distinguish two types of validity: descriptive validity and functional validity.

Definition 9. (Validity of consistency constraints)

Let $\boldsymbol{A}_D$ and $\boldsymbol{A}_F$ be the sets of axioms for descriptive semantics and functional semantics, respectively. A set $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$ of consistency constraints is *descriptively valid* if $\boldsymbol{A}_D \cup \boldsymbol{C}$ is logically consistent. The set $\boldsymbol{C}$ of consistency constraints is functionally valid $\boldsymbol{A}_D \cup \boldsymbol{A}_F \cup \boldsymbol{C}$ is logically consistent. □

We have conducted an experiment with the validity of consistency constraints using SPASS. It is proved that the constraints given above are all descriptively valid.

A consistency constraint can be ineffective if it does not impose any additional restriction on models. This is true if the constraint can be deduced from the axioms in first order logic. Thus, we have the following definition.

Definition 10. (Effectiveness of consistency constraints)

Let $\boldsymbol{A}$ be a set of semantics axioms. A set $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$ of consistency constraints is *logically ineffective* with respect to the set $\boldsymbol{A}$ of axioms if $\boldsymbol{A} \vdash \boldsymbol{C}$. □

Obviously, if $\boldsymbol{C}$ is logically ineffective, a model logically consistent in the context of axiom $\boldsymbol{A}$ will be consistent with respect to $\boldsymbol{C}$.

The consistency constraints given above are all proven to be not ineffective.

6. Implementation of Semantics Translation Tool

By translating UML models into first order logic statements, reasoning about models can be realised as logical inferences and automated by using a theorem prover. We have designed and implemented a tool Translator to translate UML models to first order logic statements. The tool is integrated with a modelling tool and a theorem prover. Fig. 6 shows the structure and workflow of the tools.

The input to Translator is UML models in XMI formats. StarUML [26], a UML modelling tool, is used to generate XMI representation of UML models. The output of our tool is a text file that is readable by SPASS, which is an automated theorem prover for first order logic with equality. Fig. 7 gives a screen snapshot of Translator, where *XMI editor* on the left displays the input XMI file and *Logic editor* on the right displays the generated formulas in SPASS input format.

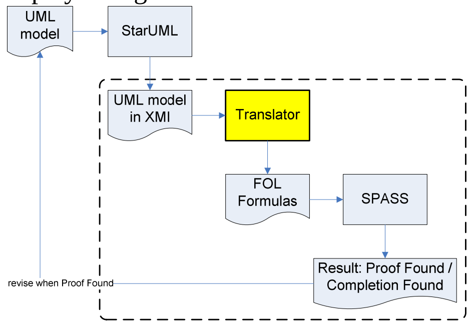
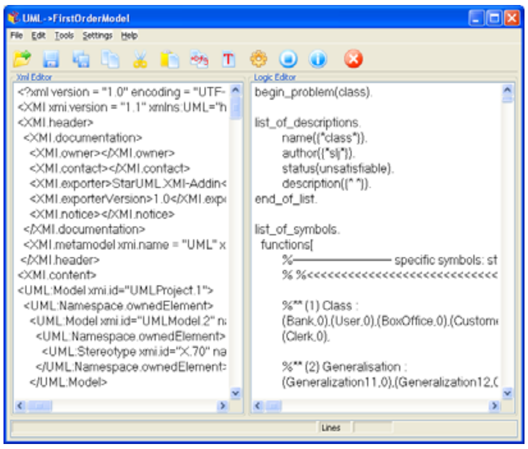


Fig. 6 Process of formalising and reasoning UML models



```
<?xml version = "1.0" encoding = "UTF-
<XMI xmi.version = "1.1" xmlns:UML="h
<XMI.header>
 <XMI.documentation>
  <XMI.owner></XMI.owner>
  <XMI.contact></XMI.contact>
  <XMI.exporter>StarUML.XMI-Addin<
  <XMI.exporterVersion>1.0</XMI.exp
  <XMI.notice></XMI.notice>
 </XMI.documentation>
 <XMI.metamodel xmi.name = "UML" x
</XMI.header>
<XMI.content>
<UML:Model xmi.id="UMLProject.1">
 <UML:Namespace.ownedElement>
  <UML:Model xmi.id="UMLModel.2" n
   <UML:Namespace.ownedElement>
    <UML:Stereotype xmi.id="X.70" na
   </UML:Namespace.ownedElement>
  </UML:Model>
```

```
begin_problem(class).

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

list_of_symbols.
 functions[
      %—————————— specific symbols: st
      % %<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

      %** (1) Class :
      (Bank,0),(User,0),(BoxOffice,0),(Custom
      (Clerk,0),

      %** (2) Generalisation :
      (Generalization11,0),(Generalization12,(
```

Fig. 7 Snapshot of Translator

When SPASS is invoked with an input generated by Translator, the consistency of the statements is inferred. In particular, it infers whether ***S***|-*false* can be proved, where ***S*** is the set of formulas including the axioms, hypothesis and formulas generated from the model and optionally some consistency constraints. Since SPASS is refutationally complete [27], if the set of statements in ***S*** is logically inconsistent, the system terminates with '*proof found*' and outputs a proof of false; otherwise if it terminates with '*completion found*', which means no proof of false can be found, so ***S*** is logically consistent.

We have used the tool to conduct a number of experiments on reasoning about interesting properties of UML diagrams. These experiments include checking the consistency of the axioms, checking model consistency without additional constraints and with various additional constraints, checking consistency constraints' validity and effectiveness, etc. Details of the experiments will be reported separately.

7. Conclusion

The main contribution of this paper is three-fold. First, we introduced the notions of descriptive semantics and functional semantics, and proposed a general framework for separately defining these two aspects of semantics of modelling languages. Second, we proposed a systematic technique to formally specify the descriptive semantics of UML in first order logic, which include the rules for rigorously inducing first order languages from metamodels, the rules for systematically deriving axioms from metamodels, and the rules for automatic translating models into formulas. Third, we successfully applied the technique to UML class diagram, interaction diagram and state machine. We also demonstrated the usefulness of the formal definition of descriptive semantics by applying it to model consistency checking, and thus laid a logic foundation for consistency checking.

Our approach has the following distinctive features in comparison with existing methods, which are in complementary to ours in the sense that they mostly defined the functional aspect of semantics.

First, our approach explicitly separates descriptive semantics from functional semantics of modelling languages. This enables the definition of the descriptive aspect of semantics to be abstract in the sense that it is independent of any subject domain. This reflects the practical uses of UML that a same model describes both real world systems and computer information systems.

Second, by introducing the notion of hypothesis in semantic definition, our approach achieves the flexibility of semantics of UML models, i.e. the same language is used for various purposes in software development.

Third, the approach is practically useful as we demonstrated the successful application of the approach on non-trivial subsets of class diagrams, interaction diagrams and state machines. In particular, our approach provides a natural and nice solution to the problem in defining multiple view modelling languages where each view is defined by one metamodel and these meat-models are interconnected.

Moreover, the translation from UML models to semantics can be rigorously defined. The translation for the subset of class diagram, interaction diagram and state machine has been implemented and tested.

Finally, the semantic definition facilitates formal and automated reasoning about models. We have demonstrated the application of such reasoning to a well-known non-trivial problem of software modelling, i.e. consistency checking. Experiments have shown promising results.

We are further researching on the definition of the functional semantics of UML in a form that can be nicely linked to descriptive semantics reported in this paper. We are also investigating logical properties of the semantic definitions.

In our investigation of UML semantics, we found a number of errors in its metamodel. Some of them were corrected in the simplified metamodel presented in this paper. More details will be reported separately.

Acknowledgement

The work reported in this paper was done during Lijun Shan's visit at the Oxford Brookes University, which is funded by China Scholarship Council. The project is partly funded by the China Ministry of Science and Technology in the National Basic Research Program (973 program) under the grant 2005CB321800 and the Foundation of Postgraduate Innovation at National University of Defined Technology, China. The authors are grateful to the members of the Applied Formal Methods research group at the school of technology of Oxford Brookes University, especially Dr. Ian Bayley and Mr Lamine Ouyahia, for numerous discussions on related topics.

References

1. *SPASS.* http://spass.mpi-inf.mpg.de/.
2. Seidewitz, E., *What models mean.* IEEE Software. **20**(5): p. 26-31. (2003)
3. OMG, *Unified Modeling Language: Superstructure version 2.0.* Object Management Group. (2005)
4. Kent, S., Evans, A., and Rumpe, B., *UML Semantics FAQ*, in *Proceedings of ECOOP'99 Workshops, Panels, and Posters. LNCS 1743.* Springer. p. 33-56 (1999)
5. Evans, A., et al., *The UML as a Formal Modeling Notation*, in *First International Workshop*

on The Unified Modeling Language «UML»'98. Springer. p. 325-334. (1998)

6. Amálio, N. and Polack, F., *Comparison of Formalisation Approaches of UML Class Constructs in Z and Object-Z* in *ZB 2003: Formal Specification and Development in Z and B*. Springer. p. 339-358. (2003)
7. Berardi, D., Cal, A., and Calvanese, D., *Reasoning on UML class diagrams* Artificial Intelligence. **168**(1): p. 70-118. (2005)
8. Varro, D., *A Formal Semantics of UML Statecharts by Model Transition Systems*, in *ICGT 2002. LNCS 2505*. Springer. p. 378-392. (2002)
9. Beeck, M.v.d., *A structured operational semantics for UML-statecharts.* Softw Syst Model, **1**: p. 130-141. (2002)
10. Reggio, G., Cerioli, M., and Astesiano, E., *Towards a Rigorous Semantics of UML Supporting Its Multiview Approach*, in *FASE 2001, LNCS 2029*. Springer. p. 171-186 (2001)
11. Reggio, G., Astesiano, E., and Choppy, C., *Casl-Ltl : A Casl Extension for Dynamic Reactive Systems -- Summary.*, in *Technical Report DISI-TR-99-34*. DISI -- Universit`a di Genova, Italy. (1999)
12. Kuske, S., et al., *An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation*, in *IFM 2002, LNCS 2335*. p. 11–28. (2002)
13. Snook, C. and Butler, M., *UML-B: Formal Modeling and Design Aided by UML.* ACM Transactions on Software Engineering and Methodology. **15**(1): p. 92–122. (2006)
14. Moller, M., et al., *Linking CSP-OZ with UML and Java: A Case Study*, in *Integrated Formal Methods*. Springer Berlin / Heidelberg. p. 267-286. (2004)
15. Nentwich, C., et al., *Flexible consistency checking.* ACM Transactions on Software Engineering and Methodology, **12**(1): p. 28-63. (2003)
16. Nentwich, C., et al., *xlinkit: a consistency checking and smart link generation service.* ACM Trans. Internet Techn, **2** (2): p. 51-185. (2002)
17. Shan, L. and Zhu, H., *Specifying consistency constraints for modelling languages*, in *18th International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*. Knowledge Systems Institute: San Francisco, CA, USA. p. 578-583. (2006)
18. Shan, L. and Zhu, H., *Consistency check in modeling multi-agent systems*, in *26th International Computer Software and Applications Conference (COMPSAC'04)*. IEEE Computer Society: Hong Kong, China. p. 114-121. (2004)
19. Muskens, J., Bril, R.J., and Chaudron, M.R.V., *Generalizing Consistency Checking between Software Views*, in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE Computer Society. p. 169-180. (2005)
20. Rasch, H. and Wehrheim, H., *Cheking Consistency in UML Diagrams: Classes and State Machines*, in *Formal Methods for Open Object-Based Distributed Systems*. Springer p. 229-243. (2003)
21. Simmonds, J. and Bastarrica, M.C., *A Tool for Automatic UML Model Consistency Checking*, in *ASE'05*. ACM: Long Beach, California, USA. p. 431-432. (2005)
22. Straeten, R.V.D., et al., *Using Description Logic to Maintain Consistency between UML Models*, in *UML 2003, LNCS 2863*. Springer. p. 326-340. (2003)
23. Egyed, A., *Instant Consistency Checking for the UML*, in *ICSE'06*. Shanghai, China. p. 381 - 390. (2006)
24. Straeten, R.V.D., Simmonds, J., and Mens, T., *Detecting Inconsistencies between UML Models Using Description Logic*, in *Proceedings of the 2003 International Workshop on Description Logics (DL2003)*: Rome, Italy. (2003)
25. Mens, T., Straeten, R.V.D., and Simmonds, J., *Maintaining Consistency between UML Models with Description Logic Tools*, in *ECOOP Workshop on Object-Oriented Reengineering*. (2003)
26. *StarUML*. http://staruml.sourceforge.net/en/.
27. Weidenbach, C., *SPASS - Version 0.49.* J. Autom. Reasoning, **18**(2): p. 247-252. (1997)