CHAPTER 1

A FORMAL DESCRIPTIVE SEMANTICS OF UML AND ITS APPLICATIONS

Hong Zhu[(1)], Lijun Shan[(2)], Ian Bayley[(1)] and Richard Amphlett[(1)]

[(1)] Department of Computing and Electronics, School of Technology, Oxford Brookes University, Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk

[(2)] Department of Computer Science, National University of Defense Technology, Changsha, China

1.1 INTRODUCTION

What is the meaning of a UML diagram? Consider the simple class model of a library system, shown in Fig. 1.1. One may interpret its meaning as follows.

> *The system has two classes called* Member *and* Book. *There is an association between them, which is called* Borrows. *The multiplicity upper bound of the* Borrows *association at the* Book end *is 10, and the multiplicity upper bound of Borrows at the* Member end *is 1.*

An alternative interpretation of the model is:

UML 2 Semantics and Applications. By Kevin Lano (Eds.)

Copyright © 2008 John Wiley & Sons, Inc.

There are two types of objects in the system called members and books. Members can borrow books. Each member can only borrow up to 10 books at any time, and each book can only be borrowed by at most 1 member at any time.

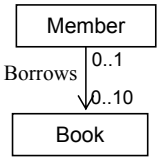

Figure 1.1 Library Systems

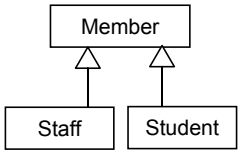

Figure 1.2 Classification of Members

As shown in the above example, in general '*a model is a set of statements about some system under study*', as Seidewitz pointed out [22]. However, the statements themselves differ according to which formalisation of UML is being used, and comparing the two interpretations above, we can identify two types.

- *Descriptive statements* describe the system based on a set of basic concepts, such as class, association, multiplicity upper bound, etc. Such statements can be used to determine which system in a given subject domain is an instance of a model. For example, consider the statement above that '*the system contains two classes Member and Book*'. This is a description of the system based on the concept of class without further information about what a class is, but by making an assertion about its construction.
- *Functional statements* define how the system functions at runtime. An example is the statement above that '*there are two types of objects in the system called member and book.*' This makes an assertion about system's runtime behaviour, i.e. the existence of two types of runtime entities.

The differences between these two types of statements become clearer when they are formalised in predicate logic. The statement '*the system contains two classes Member and Book*' can be formalised as follows,

$$Class(Member) \wedge Class(Book),$$

where $Class(x)$ is a predicate that asserts that an element x is a class. The formal representation of the statement '*there are two types of objects in the system called member and book*' in predicate logic would be

$$\exists x \cdot Member(x) \wedge \exists y \cdot Book(y),$$

where predicates $Member(x)$ and $Book(x)$ mean that the element x is of type $Member$ and $Book$, respectively. Obviously, the difference between these two statements lies in the domain of the predicates.

These two types of statements reflects two aspects of the semantics of UML: the *functional semantics* defines how an instance of a model behaves while *descriptive semantics* describes what an instance of a model 'looks like', i.e. it determines which system in a given subject domain is an instance of a model.

As far as we know, all existing work hitherto on the formalisation of UML semantics has focused on using functional statements in various formalisms to define the functions of modelled systems. As discussed briefly in Section 1.5, such works are interesting and important for the definition of UML's semantics, especially since they significantly deepen our understanding object-oriented concepts. However, a number of issues connected with the semantics of UML are neglected, and they are best addressed by descriptive semantics.

For example, consider the Java-like programs depicted in Fig. 1.3. Which one can be regarded as an instance of the model in Fig 1.2? Unfortunately, the documentation of UML does not answer this question.

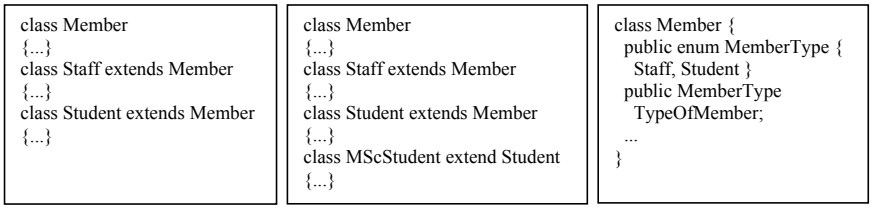

Figure 1.3 Java-like Programs

To actually answer questions like this, we proposed, in [23], an approach to formally specifying the semantics of UML in first order predicate logic (FOPL) and reported a preliminary version of an automated software tool called LAMBDES for the logic analysis of UML models. The theory and the tool focus on the descriptive semantics of UML and address the following open problems in the formalisation of UML semantics.

First, UML models are not limited to modelling computer software systems, and each UML model can be interpreted in many different subject domains. For example, the class diagram of Fig. 1.1 can be regarded as a model of libraries both in the physical world and in a computer information system as well. So, the definition of the semantics of UML must be flexible enough to be interpreted in all subject domains.

Secondly, UML is intended to provide a holistic modelling approach to object-oriented software development. It is designed for use at all stages of software development and support all software development and maintenance activities. This imposes further flexibility requirements on the formal definition of its semantics. For example, if the model in Fig. 1.2 is used as a requirements specification, all three programs in Fig. 1.3 should be considered as correct implementation of the model. If the same model is regarded as a

design of a software system, program P3 would be regarded as not following the design faithfully, so it would be an incorrect implementation. But, both program P1 and P2 should be regarded as correct instances of the model. If the diagram is the result of reverse engineering through source code analysis, it is a correct model only for program P1. So, a good definition of UML's semantics should be flexible enough to cover all these situations and many more.

Finally, UML is designed to be extensible through the use of profile definitions and new stereotypes in the metamodel. The definition of UML semantics must cover these extension mechanisms too.

In this chapter, we present the theory behind and a method for the formal definition of UML's descriptive semantics using FOPL to demonstrate how the above difficulties are overcome in our approach. We will report the current state in the development of the tool LAMBDES, which translates graphic models into descriptive semantics in FOPL and enables the formal analysis of models in FOPL by integration with a theorem prover. We will also demonstrate how the semantics and the tool support both formal analysis of models and metamodels in FOPL.

The remainder of this chapter is organised as follows. Section 1.2 presents the descriptive semantics of UML class diagrams, interaction diagrams and state machine diagrams. Section 1.3 describes the tool LAMBDES. Section 1.4 demonstrates the applications of the semantics and the tool by some examples. Section 1.5 conclude the chapter with a discussion of related work and future work.

1.2 DEFINITION OF DESCRIPTIVE SEMANTICS IN FOPL

In this section, we first outline our approach to the formal definition of UML's semantics, and then present the mappings from models and metamodels to their descriptive semantics. Then we discuss how to deal with the semantics of models in different development contexts and the extension mechanisms.

1.2.1 The Framework

As in all existing approaches to the formalisation of UML in FOPL, we define the descriptive semantics of UML through a mapping from UML models to a set of FOPL statements, which are constructed from a set of predicate and constant symbols via logic connectives and quantifiers. However, in our approach, these symbols represent the basic concepts of the modelling language rather than the concepts in the system to be modelled. For example, a predicate $Class(x)$ is defined to represent the concept *class* in UML. Moreover, our approach differs from existing works in the way that the atomic predicate symbols are derived. Instead of manually determining the signature of the FOPL system, we derive the atomic predicate and constant symbols from

the metamodels because the concepts of OO modelling are specified in UML metamodels. The collection of rules that are used to derive signature from metamodel is called the signature mapping.

A metamodel defines not only a collection of concepts but also their interrelationships. The interrelationships between the concepts are properties that all models must satisfy, and thus are the axioms of models. We also derive these axioms from the metamodel systematically with a set of rules called *axiom rules* and we represent them in the FOPL using the atomic predicates and constants in the derived signature. These axioms are called *axioms of descriptive semantics* to distinguish them from the *axioms of functional semantics*, which define the functional semantics using the runtime properties of the basic concepts. A typical axiom of descriptive semantics is

$$\forall x \cdot (Class(x) \rightarrow Classifier(x)),$$

which means that if x is a class, it is also a classifier. In contrast, here is an example of an axiom of functional semantics.

$$\forall A, B \cdot (Class(A) \wedge Class(B) \wedge Inherits(A, B) \rightarrow \forall x \cdot (A(x) \rightarrow B(x))),$$

which means if class A inherits class B, then every instance of A is also an instance of B. A full treatment of the functional semantics is beyond the scope of this chapter and it will be reported elsewhere.

The descriptive semantics of a UML model is a set of formulae in FOPL that can be systematically derived by applying a set of rules, called *translation rules*. In addition, we also specify the context in which the model is used by a set of formulae in the formal logic using the same signature. These formulae can also be derived from the model by a set of rules, so they are mappings from the model to the formulae and are called *hypothesis mappings*. In different contexts, different rules are applied.

Fig. 1.4 illustrates the overall structure of our approach to the definition of UML semantics.

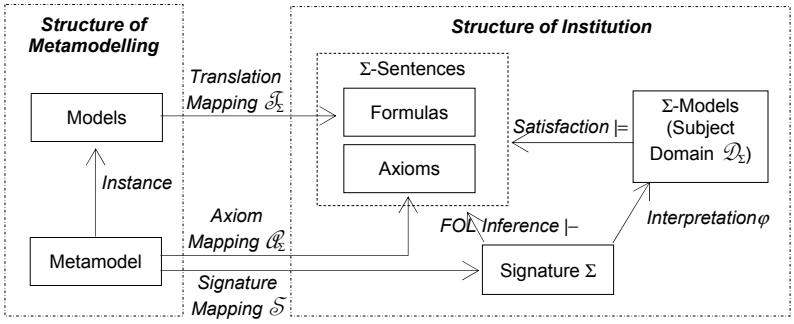

Figure 1.4 Overview of the approach to formalising UML semantics

Notation: in the sequel, we will use Σ and Axm_D to denote the signature and axioms of the descriptive semantics, derived from a given metamodel that a model M is considered as its instance. We will also use $\mathcal{T}(M)$ to denote the translation mapping from models to Σ-sentences and $\mathcal{H}(M)$ denote a hypothesis mapping from models to Σ-sentences that represent the context in which model M is to be used.

Given a formal definition of UML's semantics in the above framework, the semantics of a model is defined as follows.

Definition 1.1 *(Descriptive semantics of a model)*
The descriptive semantics of a model M *under the hypothesis* $\mathcal{H}$ *is* $[\![M]\!]_H = Axm_D \cup \mathcal{T}(M) \cup \mathcal{H}(M)$. ■

A key concept of the semantics of modelling languages is the satisfaction of a model by a system. This is defined in terms of the evaluation of the truth value of the statements in the context of the system. Given a domain of systems, the evaluation of atomic predicates is based on their interpretation in a given subject domain and provides a means of determining the value of an application of an atomic predicate. The evaluation of compound formulae constructed from atomic predicates and constants using logic connectives and equality is defined as usual in the FOPL. The details are omitted for the sake of space. Formally, the notion of subject domain and the interpretation of a formal logic in a subject domain are defined as follows.

Definition 1.2 *(Subject domain)*
A subject domain *Dom is a triple* $\langle D, \Sigma, Eva \rangle$*, where* D *is a collection of systems;* Σ *is a signature;* Eva *is an evaluation rule, i.e. a mapping from systems* s *in* D *and* Σ*-formulae to the truth value* $True$ *or* $False$*. Given a* Σ*-formula* f *and system* s *in* D*,* $Eva(f, s)$ *is called* the interpretation of the formula f in s*. We write* $s \models_{Eva} f$ *if* $Eva(f, s) = true$. ■

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

Definition 1.3 *(Satisfaction of a model)*
Let Σ *be a given signature and Dom a subject domain of* Σ*. A system* s *in* D satisfies a model M *under hypothesis* H *according to a semantic definition* $[\![M]\!]_H$ *if* $s \models [\![M]\!]_H$*, i.e. for all formulae* f *in* $[\![M]\!]_H$*,* $s \models f$*. We will also say that* s *is an* instance of model M*, and write* $s \models M$. ■

1.2.2 Semantics Mappings

We will now elaborate the approach by defining the semantics mappings. We demonstrate that the descriptive semantics of different types of diagrams can be defined using the same set of semantics mappings.

Signature Rules

S1: For each metaclass named C in the metamodel, a unary atomic predicate symbol $C(x)$ is defined to represent that the model element x is an instance of metaclass C.
S2: For each meta-attribute A of metaclass X with Y as its type, and each meta-association from metaclass X to metaclass Y with A as the association end name on Y, a binary predicate $A(x, y)$ is defined to represent the relation between model elements of type X and the elements of type Y.
S3: For each enumeration value V in the metamodel, a constant symbol V is defined.

Figure 1.5 Signature mapping rules

1.2.2.1 Signature mapping Given a metamodel, the signature of a formal logic system can be derived by applying the rules given in Fig. 1.5.

For example, consider the simplified metamodel of UML class diagrams shown in Fig. 1.6. The unary predicate $Class(x)$ represents the metaclass $Class$. The binary predicate $specific(x, y)$ represents that the association named $specific$ connects metaclass x to metaclass y in Fig. 1.6. Table 1.1 lists all the unary and binary predicates derived from the metamodel of class diagram shown in Fig. 1.6.

Table 1.1 Signature of Simplified Class Diagram Metamodel

Unary predicates	Concrete metaclasses	Generalisation, Parameter, Operation, Class, Property, Association, DataType, Signal, Interface, ParameterDirectionKind, AggregationKind, Boolean, VisibilityKind, String, Dependency, InterfaceRealisation
	Abstract metaclasses	MultiplicityElement, TypedElement, Type, Classifier, DirectedRelationship, Feature, Relationship, StructuralFeature, BehaviouralFeature, NamedElement, Element, RedefinableElement
Binary predicates	Meta-attributes	isAbstract, direction, aggregation, visibility, Name, isLeaf, isStatic
	Meta-associations	type, general, specific, supplier, client, contract, ownedParameter, ownedAttribute(2), ownedOperation(2), memberEnd, implementingClassifier
Constants	Enumeration values	in, out, inout, return, none, shared, composite, bTrue, bFalse, public, private, protected, package

Constant symbols in the signature are also derived from the metamodel. For example, two enumeration values t and f are defined in the enumeration metaclass $Boolean$ in Fig. 1.6, so two constant symbols t and f are derived.

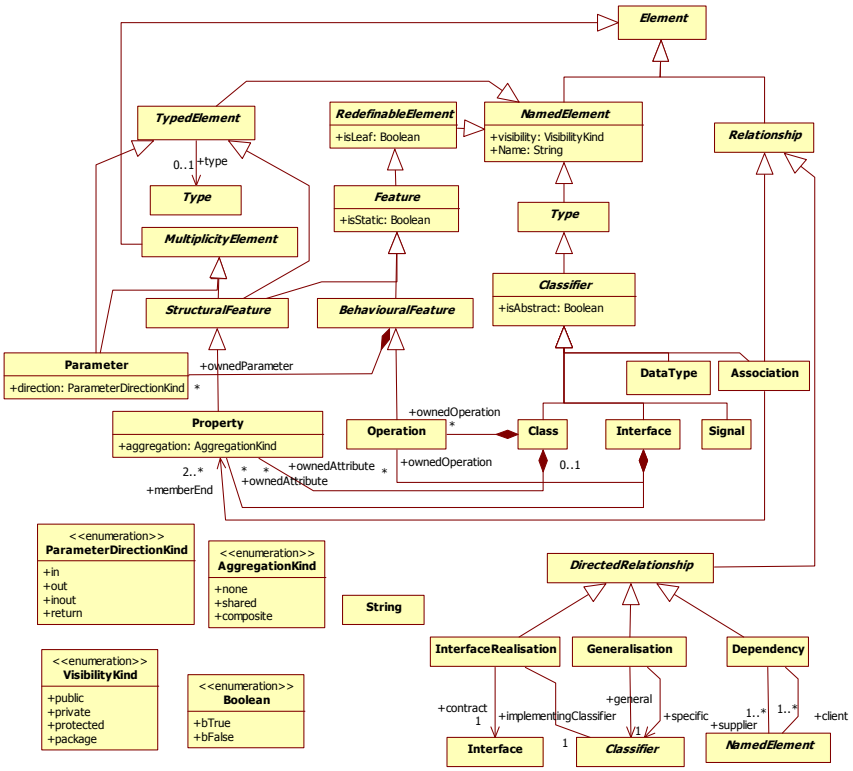

Figure 1.6 A simplified metamodel of UML class diagrams

The interpretation of the constant and predicate symbols must be defined in the context of a subject domain. Taking the set of C++ programs as an example, the predicate $Class(User)$ is true if $User$ is a class in the program. The statement $isAbstract(User, t)$ is true when the class $User$ in the program is declared to be abstract. It is worth noting that the formal definition of descriptive semantics is independent of the subject domain and its interpretation. So, we leave the definition of the interpretation open so that a model can be interpreted in different subject domains.

1.2.2.2 Translation mapping The translation mapping is a set of rules that, when applied to a model, generate a set of descriptive statements in the Σ-sentences.

For example, consider the class diagram in Fig. 1.8. The following formulae are among the statements generated by applying the translation rules.

$Class(User)$, $Class(Bank)$, $Class(BoxOffice)$, $isAbstract(Clerk, f)$.

1.2.2.3 Axiom mapping The axiom mapping for deriving axioms can be defined by a set of rules, which is given in Fig.1.9.

Translation Rules

T1. For each element e in model M as an instance of metaclass C, formula $C(e)$ is in $\mathcal{T}(M)$.
T2. For each element e in model M as an instance of metaclass C, if $Attr$ is a metaattribute of C and v is e's value on the metaattribute $Attr$, formula $Attr(e, v)$ is in $\mathcal{T}(M)$.
T3. For each pair e_1 and e_2 of elements in model M, formula $R(e_1, e_2)$ is in $\mathcal{T}(M)$, if there is an instance of metaassociation R from e_1 to e_2 in M.

Figure 1.7 Translation mapping rules

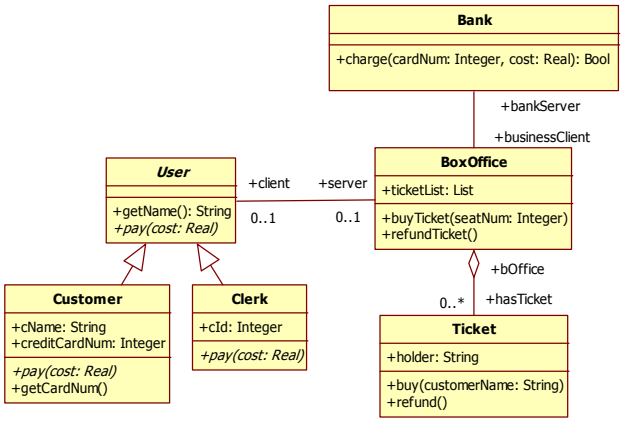

Figure 1.8 Ticket Office System: class model

For example, from the inheritance relation from $Class$ to $Classifier$ in the metamodel shown in Fig. 1.6, by applying rule A3 we can derive the axiom

$$\forall x \cdot (Class(x) \rightarrow Classifier(x)).$$

The following axiom can be obtained by applying rule A2.

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

1.2.3 Context of Modelling

As discussed in section 1.1, a UML model can be understood differently in different contexts of software development. We argue that, this variety of meanings can be represented by additional formulae, known as the *hypothesis on the model*. (Meanwhile, the core meanings of a model is still captured in the formulae generated by the translation mapping plus the axioms that all

Axiom Rules

A1. If $\{C_1, C_2, \cdots, C_n\}$ is the set of concrete metaclasses in the metamodel, the formula $\forall x \cdot (C_1(x) \vee C_2(x) \vee \cdots \vee C_n(x))$ is an axiom.
A2. For each pair of different concrete metaclasses $C \neq C'$, the formula $\forall x \cdot (C(x) \rightarrow \neg C'(x))$ is an axiom.
A3. For each generalisation relation from metaclass A to B, the formula $\forall x \cdot (A(x) \rightarrow B(x))$ is an axiom.
A4. If A is an abstract metaclass and $\{B_1, B_2, \cdots, B_k\}$ is the set of metaclasses specialising A, the following formula is an axiom.

$$\forall x \cdot (A(x) \rightarrow (B_1(x) \vee B_2(x) \vee \cdots \vee B_k(x))).$$

A5. For each an association A from metaclass C_1 to C_2, the formula $\forall x, y \cdot (A(x, y) \wedge C_1(x) \rightarrow C_2(y))$ is an axiom.
A6. For each metaattribute $Attr$ of type T in a metaclass C, the formula $\forall x, y \cdot (C(x) \wedge (Attr(x, y) \rightarrow T(y))$ is an axiom.
A7. For each association A from metaclass C_1 to C_2, if '$e_1 \cdot \cdot e_2'$ is its multiplicity value, the following formula is an axiom.

$$\forall x \cdot (C_1(x) \rightarrow (e_1 \leq ||\{y|A(x, y)\}|| \leq e_2)).$$

A8. For each metaattribute $Attr$ of type MT in a metaclass C, if '$e_1 \cdot \cdot e_2'$ is its multiplicity value, the following formula is an axiom.

$$\forall x \cdot (C(x) \rightarrow (e_1 \leq ||\{y|(Attr(x) = y)\}|| \leq e_2)).$$

A9. For each pair of different literal values a and b of an enumeration metaclass, the formula $a \neq b$ is an axiom.
A10. For each enumeration value a defined in an enumeration metaclass E, the formula $E(a)$ is an axiom.
A11. For each enumeration metaclass E with literal values $a_1, a_2, \cdots, a_k$, the following formula is an axiom.

$$\forall x \cdot (E(x) \rightarrow ((x = a_1) \vee (x = a_2) \vee \cdots \vee (x = a_k))).$$

A12. For each well-formedness rule formally specified in OCL, its corresponding formula is an axiom.

Figure 1.9 Axiom mapping rules

models must satisfy.) Hypothesis mappings can be designed and applied to models on a case-by-case basis to generate the formulae that represent the contexts in which a model is used.

For example, when a model was obtained by reverse engineering all the classes in the source code, we understand that the model is complete as a description of classes in the system. We also assume that each class in the model represents a different class in the source code. Such assumptions can be represented by the following formulae.

$$\forall c \cdot (Class(c) \rightarrow c \in \{c_1, c_2, \cdots, c_k\})$$
$$\forall c, c' \cdot (Class(c) \wedge Class(c') \wedge (Name(c) \neq Name(c')) \rightarrow (c \neq c'))$$

where $\{c_1, c_2, \cdots, c_k\}$ is the set of classes in the model M. Such formulae can be generated by transformation rules called hypothesis rules. Some examples of hypothesis rules are given in Fig. 1.10.

Hypothesis Rules

H1: Distinguishability. If $e_1, e_2, \cdots, e_k$ is the set of instances of a concrete metaclass C in the model, to assume that these elements in the model are all different the following set of formulae are generated as hypotheses. $\{e_i \neq e_j | i \neq j \in \{1, 2, \cdots, k\}\}$.

H2: Completeness of elements. If $e_1, e_2, \cdots, e_k$ is the set of instances of a concrete metaclass C in the model, to assume that these type of elements in the model is complete, the following formula is generated as a hypothesis.

$$\forall x \cdot (C(x) \rightarrow ((x = e_1) \vee (x = e_2) \vee \cdots \vee (x = e_k)))$$

H3: Completeness of relations. If $\{(e_1, e'_1), \cdots, (e_n, e'_n)\}$ is the set of instances of a relation R contained in the model, to assume the completeness of relation R in the model the following formula is generated as a hypothesis.

$$\forall x, y \cdot (R(x, y) \rightarrow (((x = e_1) \wedge (y = e'_1)) \vee \cdots \vee ((x = e_n) \wedge (y = e'_n))))$$

Figure 1.10 Hypothesis mapping rules

Now we give examples of each of these rules in turn. First, in Fig. 1.8, if we assume that class $Clerk$ is different from class $Customer$, then the formula $Clerk \neq Customer$ can be generated by applying rule *H1*. This hypothesis is applicable if the model is considered as a design, as it forces the programmer to implement the two classes $Clerk$ and $Customer$ separately, but not if it is a requirements specification instead, as then a program would satisfy the model with only one class implementing both.

Secondly, the assumption that the model in Fig. 1.8 contains all classes in the system can be specified as follows, and generated by applying rule H2.

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

Thirdly, for the model in Fig. 1.8, if we believe that all the inheritance relations in the modelled system are depicted in the diagram, then we can generate the following hypothesis by applying rule *H3*.

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

It is worth noting that the above hypothesis rules are just examples, and are by no means to be considered as complete. The point here is that the

flexibility of UML for different uses can be explicitly revealed through a set of optional hypothesis mappings. The manner in which the hypothesis rules are related to the use of the modelling language will be an interesting problem for further research.

1.2.4 Extendability and integration of multipleviews

There are two extension mechanisms in UML: meta-modelling and profiles. The former allows the language engineers to use UML class diagrams to define metamodels as far as it can be consistent with the OMG Meta Object Facility (MOF). The latter enables limited extensions of a reference metamodel by introducing new metaclasses in the form of stereotypes, for the purposes of using models in various different platforms or domains. To demonstrate that our approach to formal descriptive semantics is applicable to all metamodels, we apply the semantics mappings defined previously, to the metamodels of UML interaction diagrams and state machine diagrams, shown in Fig. 1.11 and 1.12.

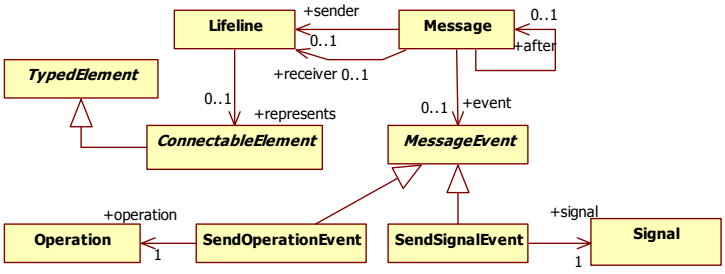

Figure 1.11 Simplified metamodel of interaction diagrams

It is worth noting that for multiple-view modelling languages like UML, each view is often defined by using one metamodel that is linked to other metamodel(s) by references to external metaclasses. For example, the metamodel for interaction diagrams refers to the kernel, which is the metamodel of class diagrams. Also, the metamodel of state machine diagrams refers to the metamodel of interaction diagrams.

The references to an existing metaclass in another metamodel may occur in one of two forms: through an association and via inheritance. In the association case, the axioms can be generated by applying exactly the same axiom rules as in the same metamodel. However, caution must be paid when implementing the axiom rules because the occurrences of a metaclass in two metamodel class diagrams may be assigned with two different internal identifiers. To ensure that the new occurrences are treated as identical to its original occurrence, the original identifier must be used.

If a metaclass is referred to via inheritance, on the other hand, new concrete metaclass(es) are introduced. Consequently, the axiom about completeness of

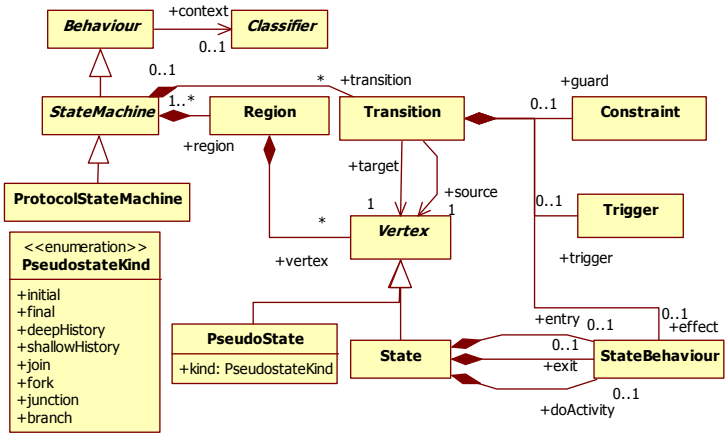

Figure 1.12 Simplified metamodel of state machine diagrams

the classification of the modelling elements must be modified. In this case, the following axiom rule must be applied instead.

Axiom Rule for Cross Metamodel References

A2'. Let A be a metaclass depicted in two metamodels M_1 and M_2. If $\{B_1, B_2, \cdots, B_k\}$ is the set of metaclasses that specialise A in metamodel M_1, and $\{C_1, C_2, \cdots, C_p\}$ is the set of metaclasses that specialise A in metamodel M_2, we have the following axiom for models defined by M_1 and M_2.

$$\forall x \cdot (A(x) \rightarrow (B_1(x) \vee \cdots \vee B_k(x) \vee C_1(x) \vee \cdots \vee C_p(x)))$$

Figure 1.13 Axiom mapping rule for Cross Metamodel References

The semantics mapping, as defined by the rules given above, was successfully applied to these metamodels to generate the signatures and axioms. Table 1.2 summarises the results of applying the rules.

The same translation rules are applicable to interaction diagrams and state machines to generate descriptive semantics of their corresponding models. For example, Fig. 1.14 depicts a simple sequence diagram and state machine for the ticket office system. The following formulae are among those generated from the sequence diagram.

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

Whereas the following formulae are among those generated from the state machine.

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

Table 1.2 Summary of the logic system for UML diagrams

	Type of element		Class Diag.	Inter. Diag.	State Mach.
Signature	Unary Predicate	Abstract metaclass	12	3	4
		Concrete metaclass	16	6	9
	Binary Predicate	Meta-attribute	7	0	2
		Meta-association	13	7	12
	Constant symbol		13	0	8
Axioms	Implication of specialisation		26	3	4
	Completeness of specialisation		12	2	3
	Disjointness of classification		120	15	36
	Domain of binary predicate		21	7	14
	Enumeration constants		33	0	37
	Multiplicity of meta-associations		14	9	12
	Completeness of classification		1	1	1

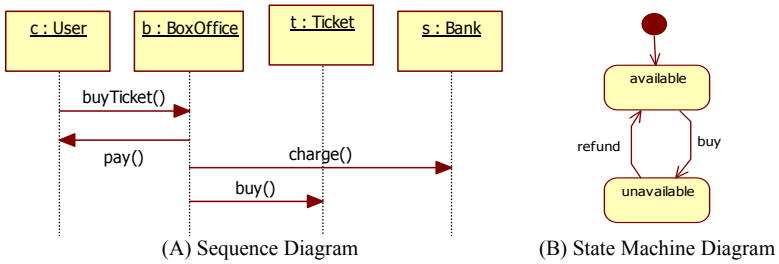

Figure 1.14 Example of sequence diagram and state machine

1.3 AN AUTOMATED MODEL ANALYSIS TOOL: LAMBDES

The descriptive semantics of UML class diagrams, interaction diagrams and state machine diagrams have been implemented in an automated software tool called LAMBDES, which stands for a Logic Analyser of Models and Metamodels Based on Descriptive Semantics. Fig. 1.15 shows its overall structure and main functions.

The current version of the LAMBDES toolkit consists of a GUI interface, a number of generators and a repository of design pattern specifications. It is integrated with a graphic modelling tool StarUML[1] and a theorem prover SPASS [2]. It takes the model or metamodel's XMI representation produced by StarUML as input to generate a logic system in the format of SPASS' input and invokes SPASS to perform logical analysis of the model and/or metamodel.

[1]Available online at URL: http://staruml.sourceforge.net/en/
[2]Available online at URL: http://www.spass-prover.org/tutorial.html.

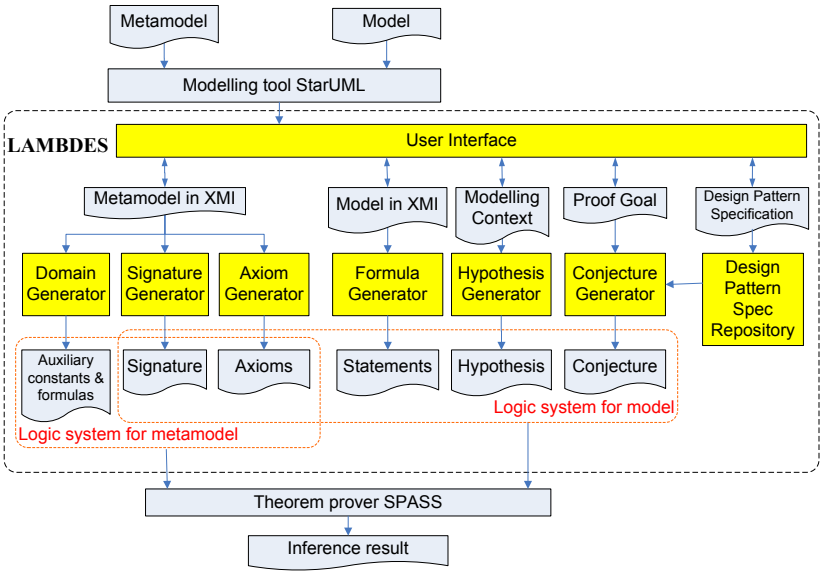

Figure 1.15 Overall structure of LAMBDES toolkit

SPASS is a general purpose theorem prover for FOPL with equality. Its input is a text file that represents a logic system with the following parts.

1. *Description*: background information not used in logic inference by SPASS;

2. *Signature*: declarations of the predicates and constant symbols of the logic system;

3. *Premises*: a list of formulae as the premises of logic inference;

4. *Conjectures*: a list of formulae to be proved.

Given an input, the execution of SPASS may terminate with a proof of the conjecture from the premises, terminate with a failure to prove, or else run forever without producing any results, because inference in FOPL is NP-hard. SPASS is refutationally complete [28], which means when it terminates with a failure to prove, the conjecture cannot be proved from the premises in FOPL.

Fig. 1.16 shows a snapshot of the tool's interface, where the input XMI file of the model is displayed on the left and the generated FOPL system in SPASS input format is displayed on the right. The analysis tool can be invoked either from the tools menu or by pressing buttons.

The main functions of the key components of LAMBDES are as follows:

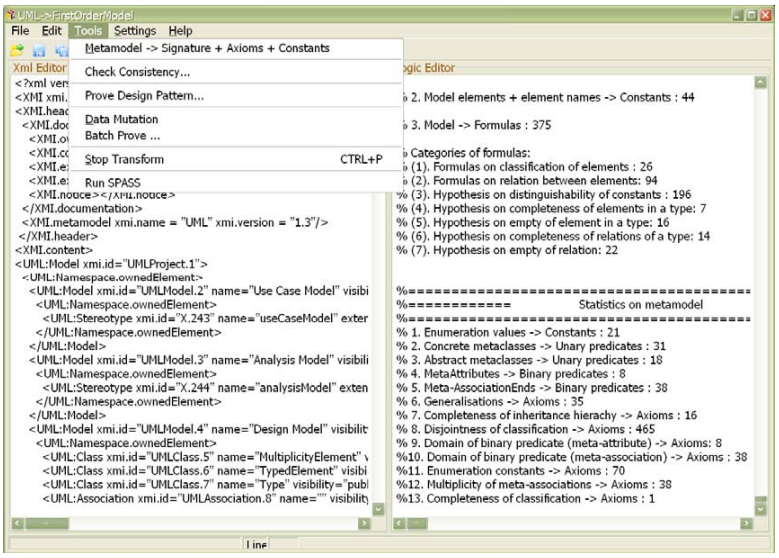

Figure 1.16 Screen snapshot of LAMBDES toolkit

- *Signature generator* implements the signature mapping rules. When a metamodel in a UML class diagram is provided by the user, this produces a signature in the form of SPASS symbol declarations.

- *Axiom generator* implements the axiom mapping rules. When a metamodel in UML class diagram is provided, this generates a set of axioms in the form of formulae in SPASS format using the symbols declared in the signature generated by the signature generator.

- *Formula generator* implements the translation rules. When a model is provided, it analyses the model and generates a set of formulae in the format of SPASS input.

- *Hypothesis generator* takes user's input about the context of modelling to generate the hypothesis formulae. Fig. 1.17 shows the GUI interface through which the user inputs the information about the context of modelling.

- *Conjecture generator* takes the user's indication of what the analysis goal is to generate the conjecture to be proved and merges the signature, axiom and formulae generated by other generators to form a complete input file to SPASS.

- *Design Pattern Specification Repository* stores a set of formal specifications of design patterns in FOPL in the form of SPASS formulae. Currently, it contains the specification of all 23 design patterns of the

GoF book [11], based on the work reported in [5]. It supports proofs that a design model conforms to a given design pattern.

- *Domain generator* takes a metamodel as input and generates a set of constant symbols of various types of model elements and instances of various relations to populate the domain when the metamodel is analysed.

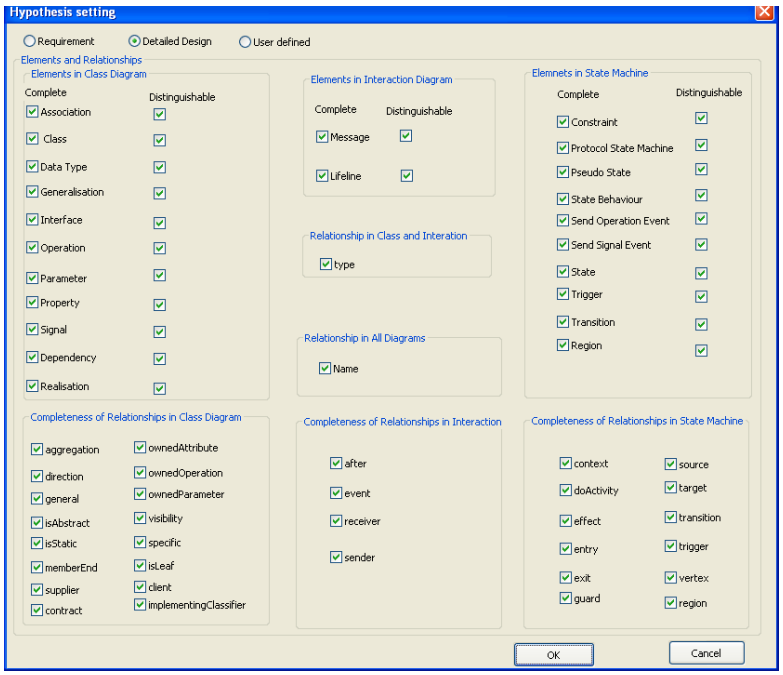

Figure 1.17 Setting modelling context in LAMBDES toolkit

1.4 APPLICATIONS IN MODEL AND METAMODEL ANALYSIS

In this section, we demonstrate some applications of descriptive semantics in the logic analysis of models and metamodels.

1.4.1 Consistency check of models

Let F be a set of formulae in a signature Σ. As in FOPL, if we can deduce that if $F \vdash false$, then F is inconsistent. Thus, we can check if a model is logically consistent or not.

Definition 1.4 *(Logical consistency)*
Model M is said to be logically inconsistent in the descriptive semantics *if $[\![M]\!]_H \vdash false$; otherwise, we say that the model is logically consistent in the descriptive semantics.* ■

In [23], it is proved that a logically inconsistent model is not satisfiable in a subject domain, where a consistent interpretation of formulae is applied.

Definition 1.5 *(Consistent interpretation of formulae in a subject domain)*
Let $Dom = \langle D, Sig, Eva \rangle$ be a subject domain. The interpretation of Σ-formulae in Dom is consistent w.r.t. FOPL if and only if for all formulae q and $p_1, p_2, \cdots, p_k$ that $p_1, p_2, \cdots, p_k \vdash q$, and for all systems s in D that $Eva(p_i, s) = true$ for $i = 1, 2, \cdots, k$, we always have $Eva(q, s) = true$. ■

Theorem 1.1 *(Unsatisfiability of inconsistent model)*
A model M that is logically inconsistent in descriptive semantics is not satisfiable on any subject domain whose interpretation of formulae is consistent w.r.t. FOPL. ■

For example, using the LAMBDES tool, we generated the descriptive semantics of the model of the Ticket Office shown in Fig. 1.2 and Fig. 1.6 and invoked the SPASS theorem prover to prove that each set of formulae generated from the three diagrams in the model are logically consistent. Their union is also consistent. Therefore, the model is consistent.

We have also made various minor changes to the diagrams in the model Ticket Office. Some changes led to logically inconsistent sets of formulae, and these were detected by theorem prover SPASS. It is, therefore, possible to check the consistency of models through logic inferences based on descriptive semantics.

It is worth noting in general though, that logical consistency does not guarantee that the model is satisfiable in a subject domain.

In additional to logical consistency, many other quality attributes of models can also be expressed in first order logic and checked through logic inference. For example, in [7], Cheng *et al.* studied 25 quality problems in software models using the tool DesignAdvisor. As show in Table 1.5, among these quality problems, 20 attributes can be represented in FOPL as indicated in the *Repr* column and 17 attributes are implemented in the LAMBDES tool as indicated in the *Impl* column. Those quality attributes that cannot be checked by LAMBDES tool include: (A) 5 quality issues defined on the bases of metrics, which cannot be represented in FOPL without arithmetics; (B) 1 quality issues related to stereo types of dependence relation, which the current version of LAMBDES does not deal with; and (C) 2 quality issues about the missing pre/post conditions of methods, which is not dealt with in the current implementation of the LAMBDES tool. Note that, in the UML metamodel, the order of parameters in a signature cannot be represented. Thus, two signatures are regarded as same if the orders are ignored.

Table 1.3 Summary of Using LAMBDES for Model Quality Checking

Error Description	Repr.	Impl.
Severe Errors:		
Abstract class not inherited	Yes	Yes
Circular association	Yes	Yes
Circular dependency	Yes	Yes
Abstract class inherits from concrete class	Yes	Yes
Class inherits from one or more non-base classes	Yes	Yes
Interface to class expected but defined improperly	Yes	Yes
Two methods exist in the model with the same signature	Yes	Yes
Two objects exist in the model with the same name	Yes	Yes
Parent accessing attributes/operations of child class	Yes	Yes
Moderate Errors:		
Number of associations above user-defined threshold	No	No
Number of attributes above user-defined threshold	No	No
Number of methods above user-defined threshold	No	No
Base artifact in an inheritance tree is concrete	Yes	Yes
Number of messages passed to a class above user-defined threshold	No	No
Multiple inheritance	Yes	Yes
Operation has more arguments than user-defined threshold	No	No
Base class in inheritance tree has publicly accessible attributes	Yes	Yes
Low Severity Errors:		
A dependency has no declared stereotype	Yes	No
Interface not used	Yes	Yes
Missing Associations	Yes	Yes
Missing Dependencies	Yes	Yes
No classes are dependent on this class	Yes	Yes
Operation missing post-conditions	Yes	No
Operation missing pre-conditions	Yes	No
A class's methods or attributes are unused by other classes	Yes	Yes

1.4.2 Validation of consistency constraints

It is often desirable to check models against consistency constraints. Fig. 1.18 gives some examples of these consistency constraints and show how such constraints can be formally specified as Σ-formulae. They cannot be derived from the axioms, and are not required for logical consistency so we clearly do need a separate notion of consistency w.r.t. a set of constraints, as follows.

Definition 1.6 *(Consistency w.r.t. consistency constraints)*
Given a set of consistency constraints $C = \{c_1, c_2, \cdots, c_n\}$, the consistency of a model M w.r.t. the constraints C in descriptive semantics is the consistency of the set $U = [\![M]\!]_H \cup C$ of Σ-formulae. In particular, we say that a model M fails on a specific constraint c_k, if $[\![M]\!]_H$ is consistent, but $[\![M]\!]_H \cup \{c_k\}$ is not. ■

Examples of consistency constraints

(1) A life line must represent an instance of a class [8, 25].

$$\forall x, y, z \cdot (Lifeline(x) \wedge represent(x, y) \wedge type(y, z) \rightarrow Class(z))$$

(2) A message must represent an operation call of its receiver [8].

$$\forall x, y, z, u \cdot (Message(x) \wedge event(x, y) \wedge SendOperationCall(y) \wedge receiver(x, z) \wedge type(z, u) \rightarrow ownedOperation(u, y))$$

(3) The classifier of a message's sender must be associated to the classifier of its receiver [8].

$$\forall x, y, z, u, v \cdot (Message(x) \wedge sender(x, y) \wedge type(y, u) \wedge receiver(x, z) \wedge type(z, v) \rightarrow \exists w, m, n \cdot (Association(w) \wedge memberEnd(w, m) \wedge AssociateTo(m, u) \wedge memberEnd(w, n) \wedge AssociateTo(n, v)))$$

(4) A protocol state transition must refer to an operation, and that operation must apply to the context classifier of the state machine.

$$\forall x, y, z \cdot (ProtocolStateMachine(x) \wedge transition(x, y) \wedge trigger(y, z) \wedge context(x, u) \rightarrow Operation(z) \wedge ownedOperation(u, z))$$

(5) The order of messages in an interaction diagram must be consistent with the order of triggers on transitions in the state machine [8, 15].

$$\forall x, y, z, u \cdot (Message(x) \wedge event(x, z) \wedge Message(y) \wedge event(y, u) \wedge after(x, y) \rightarrow Trigs(z, u)).$$

Figure 1.18 Examples of consistency constraints

It is important to know if a consistency constraint is valid and effective. Such formal analysis becomes possible now that the descriptive semantics are formally defined. First, for a consistency constraint to be valid, it must be consistent with the semantics of the modelling language.

Definition 1.7 *(Validity of consistency constraints)*
Let Axm_D be the set of axioms of descriptive semantics. A set $C = \{c_1, c_2, \cdots, c_n\}$ of consistency constraints is valid *if $Axm_D \cup C$ is logically consistent.* ■

Secondly, a consistency constraint is not effective if it does not impose any additional restriction on models. This is true if the constraint can be deduced from the axioms in FOPL. Thus, we have the following definition.

Definition 1.8 *(Effectiveness of consistency constraints)*
Let Axm be a set of axioms. A set $C = \{c_1, c_2, \cdots, c_n\}$ of consistency constraints is ineffective w.r.t. the set Axm of axioms *if $Axm \vdash C$.* ■

So a formal analysis of consistency constraints can be performed through logic inference. For example, we have used the LAMBDES tool to prove that the constraints given in Fig. 1.18 are all valid. We have also proven that they are effective by detecting models that are consistent w.r.t. the axioms but inconsistent w.r.t. the constraints.

1.4.3 Consistency check of metamodels

The LAMBDES tool can also be used to analyse metamodels by proving or disproving the consistency of the axioms generated from the metamodel. If the derived axioms are inconsistent, then the metamodel is not well-defined.

We have conducted a case study with two metamodels. The first is the UML 2.0 metamodel defined in the *Classes*, *Common Behaviours*, *Interactions* and *State Machines* packages. The second is the profile of AspectJ proposed in [10] for aspect-oriented modelling. This case study was intended to demonstrate the applicability of descriptive semantics in the analysis of proper uses of profiles as extension mechanisms. Table 1.4 summarises the logic system generated from the metamodels.

Table 1.4 Summary of the Logic Systems

	Type of element		UML 2.0 Metamodel	AspectJ Profile
Signature	Unary Predicate	Abstract metaclass	27	6
		Concrete metaclass	99	25
	Binary Predicate	Meta-attribute	58	11
		Meta-association	255	12
	Constant symbol		46	7
	Total		**485**	**61**
Axioms	Implication of specialisation		133	26
	Completeness of specialisation		25	6
	Disjointness of classification		4851	300
	Domain of binary predicate		321	23
	Enumeration constants		196	18
	Multiplicity of meta-associations		222	18
	Completeness of classification		1	1
	Total		**5740**	**392**

Two types of errors in the metamodels were detected: incompleteness errors and inconsistency errors. For an example of incompleteness, in the UML 2.0 metamodel, the data types of metaattributes are either enumeration types, e.g. *VisibilityKind*, or primitive types, e.g. *String*. The enumeration types are defined in the metamodel, while the primitive types are used in the metamodel without definition. This contradicts the statement in the *Classes Package* that "each metaclass is completely described" [18]. Incompleteness errors were detected by the SPASS theorem prover with error reports where symbol declarations were missing.

For an example of inconsistency, in the UML 2.0 metamodel, *Occurence-Specification* is specified as an abstract metaclass in one diagram and as a concrete metaclass in another. This error has been corrected in UML 2.1 [19]. A more subtle inconsistency detected, this time within the AspectJ metamodel, is that there are two association ends both named *composee*: one on the association from *PointCut* to *PointCutConjunction* and the other on the association from *PointCut* to *PointCutDisjunction*. Since an association end represents a directed relation that enables navigation between elements, two association ends of the same name from the same metaclass cause ambiguity in the direction of the navigation. This problem is detected by the theorem prover SPASS when checking the consistency of the axioms generated from AspectJ metamodel, which include the following formulae.

$$\forall x \cdot (PointCutConjunction(x) \rightarrow \neg PointCutDisjunction(x))$$
$$\forall x \cdot (PointCut(x) \wedge composee(x, y) \rightarrow PointCutConjunction(x))$$
$$\forall x \cdot (PointCut(x) \wedge composee(x, y) \rightarrow PointCutDisjunction(x))$$

Another form of inconsistency in metamodels is the violation of the *principle of strict modelling*, which states that

> *In an n-level modelling architecture $M_0, M_1, \cdots, M_n$, every element of an M_m-level model must be an* instance-of *exactly one element of an M_{m+1}-level model, for all $0 \leq m < n-1$, and any relationship other than the instance-of relationship between two elements X and Y implies that $level(X) = level(Y)$.* [2]

According to this principle, each model element must belong to one and only one concrete metaclass in the metamodel, hence the axiom mapping rules *A1* and *A2*. However, both UML 2.0 and AspectJ metamodels violate this principle. In particular, they contain concrete metaclasses as subclasses of concrete metaclasses. Therefore, a model element can belong to two concrete metaclasses, and the meanings of the model element is ambiguous. Table 1.5 lists such ambiguities in the UML 2.0 metamodel.

1.4.4 Conformance of design to design patterns

Software design patterns are frequently used to share design expertise. They document solutions to commonly occurring design problems. Tool support for patterns has been much reported at the code level [17] but not at the modelling and design stages, and the latter is increasingly important with the advent of model-driven software development methodologies. Here, we demonstrate that the descriptive semantics of UML and the LAMBDES tool can be applied to formally prove the conformance of a design represented in a UML model to a pattern formally specified in the FOPL. More details about a case study on this topic will be reported separately.

In [3, 5], Bayley and Zhu advanced an approach to the formal specification of design patterns using FOPL on UML models. Here a design pattern P is

Table 1.5 Summary of ambiguity in UML 2.0 metamodel

Package	Concrete super-metaclasses	Concrete 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

specified as a predicate $p = Spec(P)$ such that a design model M conforms to a pattern P, if the evaluation of the predicate p on model M is true. For example, the following is the specification of the *Template Method* pattern taken from [5].

Components

- $AbstractClass \in classes$
- $templateMethod \in AbstractClass.opers$
- $others \subseteq AbstractClass.opers$

Static Conditions

- $templateMethod.isLeaf$
- $templateMethod \notin others$
- $\forall o \in others \; . \; \neg o.isLeaf$

Dynamic Conditions

- The template method calls the non-leaf operations.

$$\forall o \in others \; . \; callsHook(templateMethod, o)$$

The static conditions relate to the class diagram and the dynamic conditions relate to the sequence diagram. Here, $classes$ denotes the set of classes in the class diagram. If C is a class then $C.opers$ denotes the set of operations of class C. If o is an operation then $o.isLeaf$ is true when o is not redefined in a subclass. So the static conditions state that there must be a class $AbstractClass$ with a non-redefined operation $templateMethod$ that calls a set $others$ of separate redefined operations.

In the dynamic conditions, the predicate $callsHook(op, op')$ used above is defined as $\exists C \in subs(C') \cdot calls(op, C.op')$, where $calls(op, op')$ denotes that in the sequence diagram, there exists messages m and m' in $messages$, the

set of messages, such that m, labeled with operation op, calls m', labeled with operation op'.

The mix of maths and text forming the specification above is meant to be read as a single (commented) predicate in which the variables $AbstractClass$, $templateMethod$ and $others$ are existentially quantified and the four conditions are conjoined together into a single predicate on those three variables. The general form for the predicate is

$$\exists v_1 : T_1 \exists v_2 : T_2 \cdots \exists v_n : T_n \cdot (Pr_s \wedge Pr_d)$$

where Pr_s and Pr_d are the static and dynamic conditions as predicates and the $v_i : T_i$ are the variables free in Pr_s and Pr_d.

An assignment α is a mapping from free variables in p to elements in model M. The evaluation of a predicate p on a model M in the context of an assignment α, written $Eva_\alpha(M, p)$, is the truth value of p when the free occurrences of each variable x in p are replaced by $\alpha(x)$. If $Eva_\alpha(M, p) = true$, we say that model M *satisfies* predicate p under the assignment α, and write $M \models_\alpha p$. When there is no free variable in the predicate p, its truth value is independent of the assignment so the subscript α can be omitted.

From the above discussion it is apparent that although FOPL is used both in the descriptive semantics of UML and in the formal specification of design patterns in [3, 4, 5], the universes of discourses are different. To bridge the semantic gap, the formal specification of design patterns given in [5] must be translated into Σ-sentences, i.e. in the syntax of LAMBDES tool. The translation is fairly straightforward because both languages use the same basic concepts of object-orientation. For *Template Method* pattern, we get the following.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Template Method Pattern Specification          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
formula(exists([
%Components:
  xAbstractClass, xTemplateMethod, xOthers ],
 and(
%Static conditions:
  Class(xAbstractClass),
  ownedOperation(xAbstractClass,xTemplateMethod),
  ownedOperation(xAbstractClass,xOthers),
  isLeaf(xTemplateMethod,bTrue),
  not(equal(xTemplateMethod,xOthers)),
  isLeaf(xOthers,bFalse)
%Dynamic conditions:
  callsHook(xTemplateMethod,xOthers)
))).
```

The translation mentioned above must meet the following correctness requirement.

Definition 1.9 *(Correctness of translation)*
Let p be a predicate on models, and p' be a predicate on systems. The predicate p' is a correct translation *of p, if for all models M, we have $M \models p \leftrightarrow \forall s \in \mathcal{D} \cdot (s \models ([\![M]\!] \rightarrow p'))$, where D is a subject domain.* ■

Once a specification $Spec(P)$ of pattern P is correctly translated into $Spec'(P)$, then, given a design model M represented in UML diagrams, we can decide whether the design M conforms to pattern P by proving or disproving the logic statement $[\![M]\!] \rightarrow Spec'(P)$ in FOL. For example, the translated specification of *Template Method* pattern can be deduced from the formulae generated from the class diagram in Figure 1.19.

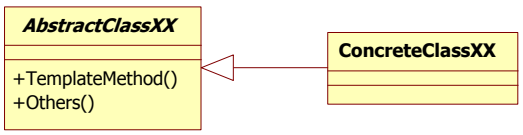

Figure 1.19 Example design instance in template method pattern

The following theorem states that if we can prove $[\![M]\!] \rightarrow Spec'(P)$ in FOPL for model M and pattern P, then every system that is an instance of M must conform to pattern P. The proof is omitted for the sake of space.

Theorem 1.2 *Suppose that $Spec'(P)$ is a correct translation of the formal specification $Spec(P)$ of pattern P. For all models M, if $[\![M]\!] \Rightarrow Spec'(P)$ is true in FOPL, then, for all systems $s \in \mathcal{D}$, $s \models M$ and $M \models Spec(P)$ imply $s \models Spec'(P)$.* ■

We have translated the specifications given in [5] for all 23 design patterns in the GoF book into LAMBDES format. They are stored in a pattern specification repository. The conjecture generator of the LAMBDES tool is implemented to enable the proof (or disproof) of the conformance of a UML design model to a pattern. We have also conducted an experiment with the LAMBDES tool on its ability to recognise patterns in design instances. The experiment results show that the false negative error rate (for rejecting a pattern it should accept) is 0% while the false positive error rate (for accepting a pattern it should reject) is below 22%. Details of the experiment are omitted here for the sake of space, and will be reported separately.

1.4.5 Logic analysis of design patterns

It is worth noting that the specification of a design pattern may contain errors. The conditions to satisfy the pattern may be in conflict with the semantics of the modelling language, or they may be in conflict with each other. Such logic errors can be detected by using LAMBDES tool and SPASS theorem prover.

In particular, let $Spec(P)$ be a specification of a pattern P. If $Axm_D \cup Spec(P) \vdash false$, we can conclude that $Spec(P)$ contains such errors.

In the development of the pattern specification repository, using LAMBDES and SPASS, we have proved that for all specifications of design patterns P in the repository, $Axm_D \cup Spec(P) \nvdash false$. So, all the specifications in our repository are consistent with the axioms of descriptive semantics.

Another application of LAMBDES and SPASS in the logic analysis of design patterns is to prove relations between patterns, for example, to prove one pattern is a specialisation of another. In [4], it is argued that the relationship that a design pattern P is a specialisation of pattern Q can be written as $Spec(P) \rightarrow Spec(Q)$. Such a relationship can be formally proved by using LAMBDES and SPASS to infer that $Axm_D \cup Spec(P) \vdash Spec(Q)$. In the context of descriptive semantics, we can now prove the following property of the pattern specialisation relation.

Theorem 1.3 *Let Dom be a subject domain that is consistent with FOPL. If $Axm_D \cup Spec(P) \vdash Spec(Q)$, then, for all systems $x \in Dom$, if x is an instance of P then x is also an instance of pattern Q, i.e. $\forall x \cdot (x \models Spec(P) \rightarrow x \models Spec(Q))$.* ■

1.5 CONCLUSION

In this chapter, we presented a framework for formalisation of UML semantics and defined a formal descriptive semantics of UML in FOPL. We introduce a tool called LAMBDES, which translates UML class diagrams, interaction diagrams and state machine diagrams to FOPL systems and is integrated with the theorem prover SPASS to enable various logic analysis of models and metamodels. A number of applications of the descriptive semantics and the tool LAMBDES are demonstrated.

1.5.1 Related work

Remarkable efforts have been made in the past decade to formalise UML semantics, so as to address the underspecification and ambiguity in UML's semantics.

With regards to the formalisation of class diagrams, often considered to be the most important type of UML diagram, a number of proposals have been advanced. The work by Evans et al. defines classifier, association, generalisation and attribute etc. in Z schemas [9]. Relations between objects and classifiers are specified as axioms. Diagrammatical transformation rules are defined as deduction rules to prove properties of UML models. In [1], a survey of the different approaches to formalising class diagram with Z or Object-Z can be found. FOPL and description logics (DLs) are used to formalise class diagram in [6]. By encoding UML class diagrams in DL knowledge bases, DL reasoning systems can be used to reason about class diagrams.

The formalisation of other types of diagrams has also been investigated, especially on state machine diagrams. In [26], a rule-based operational semantics of state machines is proposed based on transition systems. Other work on operational semantics of state machines is reported in [27]. In [14], a coalgebra framework for defining the formal semantics of sequence diagrams was proposed.

Great efforts have also been made to formalise the different diagrams in one semantic framework. Considering the semantics of a UML model as a set of acceptable structured process, the authors of [21] map class diagrams and state machines into algebraic specifications in Casl-ltl [20]. Another work aiming at integrating the semantics of class diagrams, object diagrams and state machine diagrams is based on graph transformation [13].

To bridge the gap between UML and formal methods, the extensibility mechanism of UML profiles is used to define specialisations of UML. In [24], a profile UML-B is designed so that the semantics of specialised UML entities can be defined via a translation into B. In [16], 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 labeled transition systems, or OO software systems specified in a formal notation such as Z. The properties of OO systems are specified as axioms and are used to reason about UML models. In other words, they mostly address just the functional semantics of UML. Each method focuses on certain properties of OO systems, so only a certain subset of UML is formalised. However, it is hard to see how these approaches could work either alone or together for fully-fledged UML. Most importantly, the ambiguity in descriptive semantics is not addressed in these works. Instead, their formalisation approaches are based on explicit or implicit assumptions about the descriptive semantics. They do not achieve automatic translation of UML models to formal specifications and this is necessary to facilitate formal reasoning.

In comparison with the existing works, our approach separates descriptive semantics from functional semantics so that the overall structure of semantics is much clearer and simpler. It also conforms the theory of institution proposed by Goguen and Burstall [12] for the study of formal specification languages. As shown in the chapter, our approach successfully addressed the problems of the requirements of flexibility in using models in different software development context by introducing hypothesis mappings into the semantics framework. It also successfully addressed the problem of extensibility of the semantics definition by defining semantics mappings from the metamodel to the logic system so that when new stereotype metaclasses are introduced, new atomic predicate and function symbols can be derived from profile definition, or even from a completely new metamodel. The universality of the semantic

mappings are clearly demonstrated by the application to class diagrams, interaction diagrams and state machine diagrams as well as in the case study of AspectJ profile. Our approach is also independent of the interpretation of the logic in any particular subject domain. Therefore, the semantics can be interpreted in the subject domain of computerised information systems, real world objects and physical systems, human societies, etc. as far as the basic concepts of object orientation apply. These are open problems that have not been solved in existing works.

Our approach is scalable as shown in the case study of the main parts of UML 2.0 containing four large packages and the real example of AspectJ profile, all 23 design patterns in the GoF category, etc. Our approach is also highly automated in the sense that a graphical model edited by the modelling tool StarUML can be input into LAMBDES to generate formal semantics of the model, to invoke a theorem prover to check its consistency, its conformance to design patterns, etc. Our approach applies not only to models, but also to metamodels.

1.5.2 Future work

We are investigating how functional semantics can be formally specified and the interplay between descriptive semantics and functional semantics. The static functional semantics has also been developed, and this will be reported separately.

We are also studying the logic properties of the descriptive semantics reported here. It is apparent that the axioms of descriptive semantics are consistent, as proved in the experiment by using SPASS. The particular problems that we are interested in include whether the axioms and various other semantics mappings are complete.

One of the problems that we encounter in the case studies and experiments is the inefficiency of the theorem prover. When the number of formulae in the logic system is more than a thousand, the proof that the formulas are consistent does not terminate, and this would appear to be a bottleneck for the practical uses of the LAMBDES tool.

References

1. N. Amlio and F. Polack. Comparison of formalisation approaches of UML class constructs. In *in Z and Object-Z. In Bert et al*, pages 339–358. Springer, 2003.
2. C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.
3. I. Bayley and H. Zhu. Formalising design patterns in predicate logic. In *Proc. of SEFM'07* , 2007.
4. I. Bayley and H. Zhu. On the composition of design patterns. In *Proc. of QSIC'09*, pages 27–36, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
5. I. Bayley and H. Zhu. Specifying behavioural features of design patterns in first order logic. In *Proc. of COMPSAC'08*, pages 203–210, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
6. D. Berardi, A. Cal, and D. Calvanese. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 2005.
7. B. H. Cheng, R. Stephenson1, and B. Berenbach. Lessons learned from automated analysis of industrial UML class models. In *MoDELS 2005, LNCS Vol. 3713*, pages 324–338, 2005. Springer-Verlag, Berlin Heidelberg.
8. A. Egyed. Instant consistency checking for the UML. In *Proc. of ICSE'06*, pages 381–390, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
9. A. Evans, R. B. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. In *UML'98: Selected papers from the First International Workshop*

Copyright © 2008 John Wiley & Sons, Inc.

on The Unified Modeling Language UML'98, pages 336–348, London, UK, 1999. Springer-Verlag.

10. J. Evermann. A meta-level specification and profile for AspectJ in UML. *Journal of Object Technology*, 6(7):27–49, 2007.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.
13. S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 11–28, London, UK, 2002. Springer-Verlag.
14. S. Meng and L. S. Barbosa. A coalgebraic semantic framework for reasoning about UML sequence diagrams. In *Proc. of QSIC'08*, pages 17–26, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
15. T. Mens, Der, and J. Simmonds. Maintaining consistency between UML models with description logic tools. In *ECOOP Workshop on Object-Oriented Reengineering*, 2003.
16. M. Muller, E. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A case study. In *Integrated Formal Methods, Vol. 2999, Lecture Notes in Computer Science*, pages 267–286. Springer, 2004.
17. N. Nija Shi and R. Olsson. Reverse engineering of design patterns from java source code. In *Proc. of ASE'06, Tokyo, Japan*, pages 123–134, September 2006.
18. OMG. *Unified Modeling Language: Superstructure version 2.0*. Object Management Group, 2005.
19. OMG. *Unified Modeling Language: Superstructure version 2.1.1*. Object Management Group, 2007.
20. G. Reggio, E. Astesiano, and C. Choppy. Casl-ltl : A casl extension for dynamic reactive systems – summary. Technical Report DISI-TR-99-34, DISI – Universit'a di Genova, Italy, 1999.
21. G. Reggio, M. Cerioli, and E. Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 171–186, London, UK, 2001. Springer-Verlag.
22. E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.
23. L. Shan and H. Zhu. A formal descriptive semantics of UML. In *Proc. of ICFEM'08*, pages 375–396. Springer, October 2008.
24. C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.
25. R. Van, D. Straeten, J. Simmonds, and T. Mens. Detecting inconsistencies between UML models using description logic. In *Proc. of DL2003*, 2003.
26. D. Varr. A formal semantics of UML statecharts by model transition systems. In *in Proceedings ICGT 2002: International Conference on Graph Transformation, Lecture Notes in Computer Science*, pages 378–392. Springer-Verlag, 2002.

27. M. von der Beeck. A structured operational semantics for UML-statecharts. *Software Systems Model*, 1:130–141, 2002.

28. C. Weidenbach. Spass - version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, 1997.