HANDLING MUTUAL EXCLUSION IN UML CLASS DIAGRAMS

João Araújo and Ana Moreira
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2825-114 Caparica, PORTUGAL
Tel: +351-21-294 8536; Fax: +351-21-294 8541
E-mail: {ja|amm}@di.fct.unl.pt

Keywords: UML, Object-Z, mutually excluding classes.

Abstract. UML is a standard modelling language that is able to specify a wide range of object-oriented concepts. However, there are some aspects that UML does not fully discuss. For example, UML has no mechanism to prevent the specification, for semantic reasons, of undesirable relationships. With the fast evolution of the requirements of our nowadays applications we cannot simply rely on omitting what we do not want to happen. We explicitly have to specify unwanted concepts. We are referring to the concept of mutually excluding classes. Moreover, the lack of formalisation compromises the precision of the specification of the concepts. By using formal description techniques, such as Object-Z, we can reason about the requirements and identify ambiguities and inconsistencies earlier in the development process. Particularly, the formal specification can be used through the software evolution. In general, we can say that formalising helps obtaining a more reliable system. Our aim is to specify precisely mutually excluding classes.

1. INTRODUCTION

UML (Unified Modeling Language) has a very rich notation (Booch et al. 1998). However, some modelling features like mutually excluding classes are neglected compromising significantly the specification of several applications. Two classes are mutually excluding if it is not desirable to have any kind of relationship between them. In general we could say that relationships omitted, i.e., not represented, in a class diagram are due to two different factors:

?? they are of no interest to the requirements being analysed, but in a different context they could be related;

?? we want to specify explicitly that they must not be related.

The situation we discuss here is the latter, i.e., two classes should not be related, independently of the context. Mutual exclusion will be discussed first informally and then formally with defined rules to mirror the transition between the informal and the formal representations.

Building a formal specification of a system requires a rigorous set of rules to transform a set of informal requirements into a formal specification, bridging the gap between the two representations. Formalisation is useful so that we can identify ambiguities, inconsistencies and incompleteness earlier in the software development process. Our goal is to describe mutually excluding classes formally.

Our proposal is based on the integration of UML models and formal specification languages. Many formal specification languages can be used in the definition of these rules. Here, we have chosen Object-Z (Duke et al. 1991) because it is a well-known object-oriented formal language.

The advantages of our proposal are two folds. Firstly, the high degree of volatility of most application requirements makes this concept an important one. With the fast evolution of the requirements in nowadays applications, we must make sure that an unwanted relationship should not be possible to define later during changes. Secondly, its formalisation contributes to a more complete semantics to the UML class diagram. Therefore, if a new association is required, the process must check if there is conflict, i.e., if the classes are mutually excluding. If this is the case, the conflict must be resolved either by forbidding the request or changing the initial constraint to allow the new association.
This paper is organised as follows. Section 2 presents the related work. Section 3 describes mutually excluding classes. Section 4 presents a case study and shows its class diagram. Section 5 summarises the Object-Z specification language. Section 6 specifies mutually excluding classes using Object-Z and applies the results to our case study. Finally, Section 7 draws some conclusions.

2. RELATED WORK

There has been relatively little work on integrating object-oriented methods with formal description techniques. Lano (Lano 1995) uses OMT (Rumbaugh et al. 1991) and Booch (Booch 1994) combined with the formal languages VDM++ (Durr et al. 1994) and Z++ (Lano 1990). ROOA (Rigorous Object-Oriented Analysis) (Clark and Moreira 1999, Moreira 1994, Moreira and Clark 1996) builds a formal and executable object-oriented specification from informal requirements using SDL (Z.100 1994) or LOTOS (ISO 1998). Metamorphosis (Araújo 1996) combines an object-oriented model with Object-Z (Duke et al. 1991). However, none of them considers formalising all UML models (Booch et al. 1998).

There is some work on formalising UML model components. The precise UML (pUML) group has been undertaking collaborative work to use formal techniques to explore and define appropriate semantic foundations for object-oriented concepts and UML notations. More information can be found on the website http://www.cs.york.ac.uk/puml/. OBLOG proposes a development environment where some of the object-oriented concepts are formalised (see http://www.oblog.com). France discusses the formalisation of the UML static requirements modelling concepts (France 1999). Overgaard gives a formal definition of the collaboration construct in the UML (Overgaard 1999). Knapp presents a formal semantics for UML interactions (Knapp 1999). In (Moreira and Araújo 1999) we describe how use cases and sequence diagrams are mapped into Object-Z class schemas. Additionally, Kim and Carrington show the formalisation of the UML class diagram using Object-Z (Kim and Carrington 1999).

Mutually excluding classes are discussed in (Araújo 1996). This approach uses Object-Z to formalise mutual exclusion, but it does not consider the UML notation. Moreover, that approach is limited to generalisation and specialisation relationships. Here, we will extend this feature to association, aggregation and composition relationships in a UML class diagram, and also use Object-Z to formalise mutually excluding classes.

3. MUTUALLY EXCLUDING CLASSES

Mutual exclusion happens when we do not want that at least two classes be related to each other in a particular way. Each relationship has its own semantics; therefore, the definition of mutual exclusion varies according to the kind of the relationship. The main types of relationship in UML are (Rumbaugh et al. 1999):

?? Generalisation: This is a relationship between a more general element (parent) and a more specific element (child);

?? Specialisation: This is the opposite of generalisation;

?? Association: This is a relationship among two or more specified classifiers that describes connections among their instances. An association has an optional name, and a list of association ends, each of which describes (through, for example, multiplicity and role names) the participation of the objects of a class in the association;

?? Aggregation: This is a form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part;

?? Composition: This is a form of aggregation association with strong ownership and coincident lifetime of parts by the whole. A part object may belong to only one composite. When a composite object is copied or deleted, its component objects are copied or deleted with it.

UML has no mechanism to control undesirable relationships. Nevertheless, for semantic reasons, we may not want to allow mutual exclusion between classes related by any of the above relationships.

Generalisation and specialisation. There are two types of generalisation and specialisation hierarchies: when one class (subclass) inherits from only one parent class (superclass); when a class inherits from one or more superclasses. In the first case, we want to prevent two classes from being related via single inheritance. In the second case we do not want to have two or more classes to be the parents of a certain class (multiple inheritance). This provides a constraint on the inheritance lattice to
proscribe dangerous or nonsensical specialisations, and to help anticipating possible name clashes, for example.

**Associations.** We also want to control mutual exclusion between classes related via a static association. Once again, for semantic reasons, we may not want two given classes to have any kind of association between them, i.e. their respective objects may never be linked to each other.

**Aggregations.** An aggregation relationship allows us to form a complex object from other simpler objects. In this case, we may not want one object to be part of another one, i.e. two objects must not act one as an aggregate object and the other as the component object to form an aggregation.

**Compositions.** A composition relationship is a form of aggregation association with strong ownership and coincident lifetime of parts by the whole. In this case, we want to guarantee that two objects must not act one as a composite object and the other as the component object to form a composition.

4. **CASE STUDY**

The properties described above will be discussed and formalised using a simplified version of a banking system. The requirements are as follows:

“The system has two types of accounts: cheque accounts and savings accounts. Provided a correct account number, we can access an account to withdraw some cash, make a deposit and get the balance. Savings accounts have interest rate. Periodically, the interest is calculated and credited to its balance. A savings account can only exist if its balance is at least 500 euros.

Finally, an account can never be a cheque and savings account at the same time, a cheque account is not a savings account (i.e. a cheque account does not credit interest), and, in general, a cheque account and a savings account are not linked to each other.”

Analysing the requirements described above, we can build the class diagram depicted in Figure 1. As we have two types of account we can define a superclass `Account` with two subclasses `ChequeAccount` and `SavingsAccount`. `Account` has the protected attributes `accNumber` and `balance` and the public operations `Withdraw`, `Deposit` and `GetBalance`. All these properties are inherited by the two subclasses `SavingsAccount` and `ChequeAccount`.

`ChequeAccount` adds the public operation `printMiniStatement`. `SavingsAccount` adds the private attributes `period` and `interestRate`, the constant `minBalance`, the public operation `CreditInterest` and the private operation `CalculateInterest`.

Pre and postconditions in operations, and invariants in classes, can either be defined in OCL (Warmer and Kleppe 1999) or rendered in notes stereotyped as `<<precondition>>`, `<<postcondition>>` and `<<invariant>>` attached to the operation or class by dependency relationships. As a matter of simplicity for the users, we have chosen to use notes with stereotypes, as shown in Figure 1. The invariant `balance >= minBalance` associated with `SavingsAccount` is due to the restriction described in our case study. It determines that the `balance` value is at least equal to the `minBalance` value. The precondition `Account is valid` and the postconditions `Interest is calculated` and `Balance is updated` are associated to the operation `CreditInterest`. These conditions are specified in the note attached to `SavingsAccount`.

According to our requirements we have three constraints to satisfy:

?? an account can never have the properties of cheque and savings account at the same time;

?? a cheque account is not a savings account (i.e. a cheque account does not credit interest);

?? a cheque account and a savings account are not linked to each other.
The first constraint is automatically satisfied, as there is no subclass that has cheque and savings account as its parents. The remaining two constraints, saying that a cheque account and a savings account are mutually excluding, were added to the UML class diagram by using a note attached to both subclasses. Their formal specification is defined using Object-Z.

5. AN OVERVIEW OF OBJECT-Z

Object-Z (Duke et al. 1991) is an object-oriented extension of Z (Spivey 1992); it has been used in many real applications, including real-time systems in the telecommunications area. It is a model-based language that has its roots, like Z, in typed set theory; its most important feature is the class schema. A class schema takes the form of a named box, optionally with generic parameters (see Figure 2). It extends the graphical component of Z (boxes) to define its classes, providing an immediate visual indication of the scope of the definition.

---

**Fig. 1.** Class diagram of the banking system

**Fig. 2.** Object-Z class schema
substituted by a new one, and className is the name of the inherited class;

- a list of type definitions, that includes given sets (types that must be refined later) and free types (enumerated types), which are defined by the user;

- a list of constant definitions, that can be realised using axiomatic descriptions that are composed of a declaration part and a predicate part;

- a state schema which defines the class invariant and its state attributes. It is composed of a declaration part and a predicate part;

- an initial state schema that specifies the initial state of the objects of the class;

- a list of operation schemas that specifies the input and output parameters, and pre and postconditions of the operations of the class;

- a history invariant that constrains the order of the operations based on temporal logic.

History invariants will not be necessary in this paper; therefore, they will not be mentioned again.

6. SPECIFYING MUTUAL EXCLUSION USING OBJECT-Z

Mutual exclusion is modelled in UML as a constraint in a note, whereas, in Object-Z, this should be specified using class schemas that define the appropriate predicates. For each type of relationship, the approach used consists of:

1. identifying a set of rules that generally describes the formalisation of the constraint;
2. formalising that constraint by defining a generic class schema whose parameters are the classes involved in the relationship;
3. putting together all the specified generic classes, i.e., all the mutual exclusion constraints, in the class schema that represents the whole system. This class schema inherits those generic class schemas and renames the generic parameters.

The following subsections use this approach to formalise mutual exclusion for each of the relationships described in Section 3.

6.1 Mutual exclusion in generalisation and specialisation relationships

Concerning generalisation and specialisation relationships there are two situations that must be considered separately: single inheritance and multiple inheritance. Both cases of inheritance are semantic variations of generalisation.

Single inheritance. The rules for single inheritance are:

1. name the generic class schema for mutual exclusion in single inheritance as MECSingleInherit(X, Y);
2. suppose that X is the candidate superclass and Y is the candidate subclass:
   - the predicate part specifies that the set of objects of the (sub)class Y is not included in the set of objects of the (super)class X, i.e., the class Y cannot be a subset of class X.

Figure 3 shows the class schema obtained by applying the rules above.

```
<table>
<thead>
<tr>
<th>MECSingleInherit[X, Y]</th>
</tr>
</thead>
</table>
```

Fig. 3. Class schema of MECSingleInherit(X, Y)

If we were to generalise this case to have several candidate subclasses, the predicate would be built of a conjunction of terms, wherein each term would be the non-inclusion of each different subclass in the superclass.

Multiple inheritance. The formalisation of the constraint of classes that cannot be parents of a given subclass is defined using the following rules:

1. name the generic class schema for mutual exclusion in multiple inheritance as MECMultiInherit(X, Y, Z);
2. consider that X and Y are the candidate superclasses and that Z is the candidate subclass:
   - the predicate part specifies that the set of objects of the subclass are not included in the intersection of the set of objects of the superclass. This specifies that it is not possible do define a subclass that has the two classes as parent classes.
The resulting class schema is depicted in Figure 4.

\[
\text{MECMultiInherit}[X, Y, Z] \quad \downarrow \quad ? \ x: X; \ y: Y; \ z: Z \quad ? \ (x ? y)
\]

**Fig. 4.** Class schema of MECMultiInherit(X, Y, Z)

If we were to generalise this case to have several candidate superclasses, the predicate would include the intersections of all the superclasses.

### 6.2 Mutual exclusion in associations

In the case of the association relationships, the formalisation of the constraint of classes that cannot be linked through a static relationship is defined using the following rules:

1. name the generic class schema for mutual exclusion in association as MECAssoc(X, Y);
2. consider that two classes X and Y cannot have an association; as an association is bi-directional, the predicate is composed of a conjunction formed by two parts:
   - the first part says that there does not exist a link from an object of X to an object of Y;
   - the second part defines the inverse relation, i.e., it says that there does not exist a link from an object of Y to an object of X.

Figure 5 shows the class schema obtained by applying these rules.

\[
\text{MECAssoc}[X, Y] \quad \downarrow \quad \downarrow \quad ? \ x: X; \ y: Y \quad ? \ (x ? y)
\]

**Fig. 5.** Class schema of MECAssoc(X, Y)

This is the case for binary associations. If we had a n-ary association, the idea would be the same, and we only had to add similar terms to the conjunction.

### 6.3 Mutual exclusion in aggregation relationships

Concerning aggregation relationships, the formalisation of the constraint of a class that cannot be an aggregate and another class that cannot be its component part, is defined using the following rules:

1. name the generic class schema for mutual exclusion in aggregation as MECAggreg(X, Y);
2. suppose that X is the candidate class to be the aggregate and Y is the candidate class to be the component; in this case, we would only need to specify the relation in one direction, from the whole to the part:
   - the predicate part specifies that there is no link \( r \) involving the objects of the two classes, where the elements of the domain belong to X (the aggregate) and the elements of the image belong to Y (the component part).

Figure 6 shows this class schema.

\[
\text{MECAggreg}[X, Y] \quad \downarrow \quad ? \ x: X; \ y: Y \quad ? \ (x ? y) \quad ? \ r: X \quad ? \ Y
\]

**Fig. 6.** Class schema of MECAggreg(X, Y)

If we were to generalise to the case where we have several candidate class components, we would use a conjunction of the predicate above with new similar predicates, all of them considering the same domain X.

### 6.4 Mutual exclusion in composition relationships

In the case of composition relationships, the formalisation of the constraint of a class that cannot be a composite and another class that cannot be its component part is defined using the following rules:

1. name the generic class schema for mutual exclusion in composition as MECComp(X, Y);
2. as with in aggregation, suppose that X is the candidate class to be the composite and Y is the candidate class to be the component; in this case, the predicate part has to specify that:
there is no link \( r \) where each element of the image (component part) is related to only one element of the domain (the composite).

Figure 7 shows this class schema.

\[
\text{MECComp}[X, Y]
\]

\[ ? \ y: Y ? \ (? \ r: X \ Y \ (? \ x:X \ (x, y) ? \ r)) \]

Fig. 7. Class schema of \( \text{MECComp}(X, Y) \)

If we were to generalise to the case where we have several candidates to be class components, the predicate would be a conjunction of terms built using a similar idea to the aggregation case.

6.5 Integrating the generic class schemas in the application

The complete specification of our banking system includes:

\[
\text{a class schema for each class showed in the class diagram;}
\]

\[
\text{the class schema } \text{MECSingleInherit} \text{ to render the constraint “a cheque account does not credit interest”;}
\]

\[
\text{the class schema } \text{MECAssoc} \text{ to render the constraint “cheque account and savings account are not linked to each other”;}
\]

\[
\text{the class schema } \text{BankSystem} \text{ that represents the whole system.}
\]

The two class schemas \( \text{MECSingleInherit} \) and \( \text{MECAssoc} \) are inherited by \( \text{BankSystem} \). Figure 8 represents the whole system.

\[
\text{BankSystem}
\]

\[
\text{MECSingleInherit}[\text{savingsAccountSet}/X, \text{chequeAccountSet}/Y]
\]

\[
\text{MECAssoc}[\text{savingsAccountSet}/X, \text{chequeAccountSet}/Y]
\]

\[
\ldots
\]

\[
\text{savingsAccountSet} : \uparrow \text{SavingsAccount}
\]

\[
\text{chequeAccountSet} : \uparrow \text{ChequeAccount}
\]

\[
\ldots
\]

Fig. 8. Class schema of \( \text{BankSystem} \)

\( \text{SavingsAccount} \) and \( \text{ChequeAccount} \) rename the generic parameters \( X \) and \( Y \) defined in the two class schemas that specify mutual exclusion. Note that the complete state schema includes the predicates of those inherited classes.

7. CONCLUSIONS

This paper discusses mutual exclusion among classes, a feature that is not directly incorporated in UML. This constrains the class diagram by not allowing undesired relationships among classes. The idea is to have the power to control situations where it is important to specify what should not happen, instead of just adopting a view of specifying by omission, i.e., if a certain property is not specified then it should not be used. In general, we could say that relationships omitted, or not represented, in a class diagram is due to two different factors:

\[
\text{they are of no interest to the requirements being analysed, but in a different context they could be related;}
\]

\[
\text{we want explicitly to specify that they must not be related.}
\]

The situation we have discussed is the latter, i.e. two classes should not be related, independently of the context. First, we presented the ideas informally, then we came up with a set of rules to mirror the transition between the informal and the formal representations, and, finally, we used Object-Z as a technique to specify the ideas in discussion. In particular, we have studied how to control mutual
exclusion between classes related by four kinds of relationships: generalisation and specialisation, association, aggregation and composition. In what concerns the generalisation and specialisation relationship, we have paid special attention to both single and multiple inheritance.

In summary, from a UML class diagram of an application model, we used Object-Z to formally specify the mutual exclusion constraint. There are two main advantages of having that constraint formally specified. On the one hand, we can guarantee that undesirable relationships will never be allowed, even during the maintenance of a system to deal with evolving requirements. On the other hand, the formalisation contributes to a more complete semantics to the UML class diagram. Moreover, the concise Object-Z syntax provides an object-oriented structure that facilitates the traceability to the UML specification. For example, a class schema is directly related to a classifier in a class diagram. More specifically, in our case a class schema is related to a constraint between classifiers.

REFERENCES


