Proc. <<UML>>98 Workshop, Jean Bezivin, Pierre-Alain Muller (Eds.)
Springer, Berlin, LNCS 1618, pp. 92-106 (1998)

Transformation Rules for UML Class Diagrams

Martin Gogolla & Mark Richters
University of Bremen, FB 3, Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
Phone +49-421-218-3495
Fax +49-421-218-3054
{gogolla|mr}@informatik.uni-bremen.de

Abstract. UML is a complex language with many modeling features. In particular, the modeling of static structures with class diagrams is supported by a rich set of description primitives. We show how to transform UML class diagrams involving cardinality constraints, qualifiers, association classes, aggregations, compositions, and generalizations into equivalent UML class diagrams employing only n-ary associations and OCL constraints. This provides a better understanding of UML features. By explaining more complex features in terms of basic ones, we suggest an easy way users can gradually extend the set of UML elements they commonly apply in the modeling process.
Keywords: UML Class Diagram, Model Transformation, Model Transformation, OCL, Constraint.

1 Introduction

UML [BJR97c,BJR97b,BJR97a] is a complex language with many modeling features. But up to now, no overall agreed precise semantic foundation has been developed. So naturally the question arises how to deal with the language element variety, for example, to relate the different language elements. In this paper we study UML class diagrams and compare the language elements available there. The general idea here is to explain advanced UML features by some more basic UML constructs. A similar approach [GPP98] has been taken by us recently for UML state diagrams concerning dynamic modeling aspects.

Although UML provides a rich set of language features, many projects are forced to restrict themselves to utilizing only a small set of the language. This is partly due to a current lack of sufficiently reliable and efficient tools supporting the whole language and partly due to the fact that more time and effort is needed to learn and adapt additional features. Indeed, using only a small set of features can reduce the complexity of a design and facilitate communication. Our approach enables a smooth transition from utilizing a basic set of language features to more sophisticated ones by explaining them in terms of already known features. On the other hand, giving a definition in terms of simple features possibly helps identifying certain repeating patterns in existing designs which may better be represented by more suitable special features, thus emphasizing a particular design decision. Thus, what we do in this paper is to – in a sense – reverse engineer simpler UML concepts from more sophisticated ones.

Our work is related to recent approaches handling formal aspects of UML and other object-oriented methods. Work has been done on the basis of well-established traditional approaches to specification like Z and VDM: [SF97,FBLPS97] focuses on the UML type system, a general integration of Z with object-orientation is discussed in [Ebe97], and in [Lan96,BLM97] an object calculus enhancing the expressibility of object-oriented notations has been proposed. Other approaches treat in detail the UML predecessor OMT [BCV96,WRC97], and in particular class diagrams [BC95] in connection with Larch are discussed. [BHH$^+$97] sketches a general scenario for most UML diagrams without going into technical details, and [Öve98] presents a general framework for relationships in UML. We cannot give a deeper discussion here due to space limitations.

The structure of the rest of this paper is as follows. Sections 2 to 6 point out how to translate n-ary associations with cardinality restrictions, qualifiers, association classes, aggregations and compositions, and generalizations, respectively. The UML subset we use employs general n-ary associations with additional constraints formulated in OCL. These sections all have the same structure. First, we motivate the handled concept by citing respective parts from the "official" UML material: the semantics document [BJR97c], the notation guide [BJR97b], and the OCL description [BJR97a]. Then, a general translation scheme is presented, which is explained afterwards by an example. The paper ends with some concluding remarks.

2 N-ary Associations with Cardinalities

2.1 Statements from UML Material

- *An n-ary association is an association among three or more classes (a single class may appear more than once).* Notation, page 61, line 4.
- *Multiplicity for n-ary associations may be specified but is less obvious than binary multiplicity. The multiplicity of a role represents the potential number of instance tuples in the association when the other N-1 values are fixed.* Notation, page 61, line 7.

2.2 General Translation

As indicated in the original UML statements, the translation of n-ary associations with cardinality restrictions is a bit more involved in comparison to the binary case. The diagram in Fig. 1 keeps on the right hand side the starting situation except the cardinality requirement.[1] The formula fixes the pair of objects on the opposite side of the cardinality specification and requires, that the size of the

[1] We use the notion of transformation here. Indeed, we could have also called this rule an equivalence rule in the following sense: Both sides of the rule have the same state

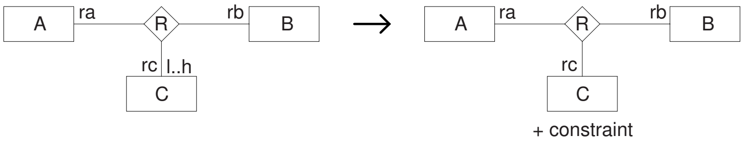

Fig. 1. Transformation Rule for N-ary Association

set of related objects is restricted by the given lower and upper bounds. The formula expresses the connection between three participating objects as `R(a,b,c)`[2], and `A->forAll(...)` is short for `A.allInstances->forAll(...)` (analogously for `exists` and `select`).

```
A->forAll( a |
  B->forAll( b |
    C->select( c | R(a,b,c) )->size>=l
    and
    C->select( c | R(a,b,c) )->size<=h ) )
```

Above, we have shown only the translation for a ternary association. The formula can be generalized to the arbitrary case with n participating classes by introducing a universally quantified variable for each of the $n-1$ classes on the opposite side of the cardinality specification.

2.3 Example

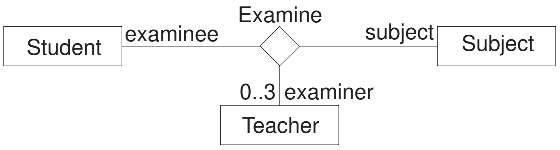

Fig. 2. Example for Cardinality Notation in N-ary Association

The example in Fig. 2 shows a ternary association where students can be examined by teachers on certain subjects. The following cardinality restriction requires that a student can be examined on a single subject at most three times (each time by a different teacher).

space in that for each left hand side state there is a corresponding right hand side state and vice versa. But some rules to follow will introduce more operations on the right hand side.

[2] The current version of OCL does not support this form of syntax. There is no way in OCL to express the existence of links for n-ary associations.

```
Student->forAll( st |
  Subject->forAll( su |
    Teacher->select( t | Examine(st,su,t) )->size>=0
    and
    Teacher->select( t | Examine(st,su,t) )->size<=3 ) )
```

3 Association Classes

3.1 Statements from UML Material

- *An association class is an association that also has class properties (or a class that has association properties).* Notation, page 59, line -12.
- *An association class is an association that is also a class. It not only connects a set of classifiers [classes] but also defines a set of features that belong to the relationship itself and not any of the classifiers.* Semantics, page 18, line 2.

3.2 General Translation

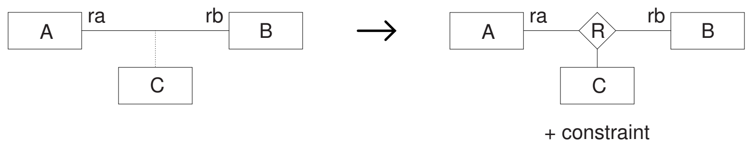

Fig. 3. Transformation Rule for Association Class

In Fig. 3 the translation of association classes into ternary associations is shown. The specified role names induce the operations `ra: C -> Set(A)` and `rb: C -> Set(B)`. The formula expresses the idea that each `C` object "points" to a unique pair of `A` and `B` objects. In other words, `R` constitutes an injective function from `C` into the product of `A` and `B`.

```
C->forAll( c |
  c.ra->size=1 and c.rb->size=1 and
  C->forAll( c' | (c.ra=c'.ra and c.rb=c'.rb) implies c=c' ) )
```

3.3 Example

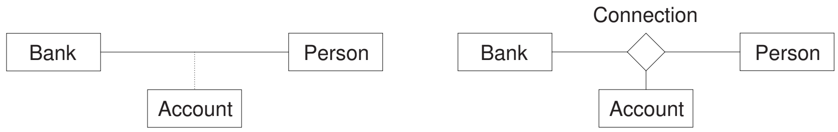

Fig. 4. Example for Association Class

In Fig. 4 a bank example is modeled with a ternary association called `Connection` between `Bank`, `Account`, and `Person`. The constraint demands that (1) an account is related to exactly one bank and to exactly one person, and (2) there cannot be a different account with the same links.

```
Account->forAll( a |
  a.bank->size=1 and a.person->size=1 and
  Account->forAll( a' | ( a.bank=a'.bank and a.person=a'.person )
                        implies
                        a=a' ) )
```

4 Qualifier

4.1 Statements from UML Material

- *A qualifier is an attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association. The qualifiers are attributes of the association.* Notation, page 58, line 4.
- *The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value.* Notation, page 58, line 13.
- *[A] qualifier [is] an association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.* Semantics, page 156, line 9.

4.2 General Translation

Figure 5 shows that a qualifier is translated into an association class (if the association class is already present, the qualifier translates into an additional attribute of the association class). The respective constraint is illustrated in Fig. 6: We fix an `A` object, look for all `AC` objects with a fixed `q` attribute value, and restrict the number of related `B` objects. The formula requires that the size of the set of `B` objects determined by a combination of an `A` object and a `q`

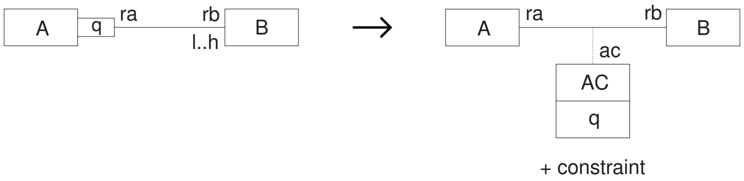

Fig. 5. Transformation Rule for Qualifier

attribute value is restricted by the given lower and upper bound. Regarding the operations used in the defining formula, the role name `rb` implies we have an operation `rb: A -> Set(B)`. Due to the additional association class `AC`, we have further operations `ra: AC -> A`, `rb: AC -> B`, `ac: A -> Set(AC)`, and `ac: B -> Set(AC)`.

```
A->forAll( a |
  a.ac->forAll( ac |
    a.rb->select( b |
      b.ac->exists( ac' | ac'.q=ac.q and ac'.ra=a ) )->size>=l
    and
    a.rb->select( b |
      b.ac->exists( ac' | ac'.q=ac.q and ac'.ra=a ) )->size<=h ) )
```

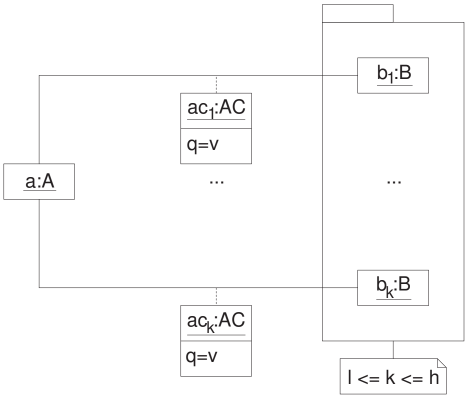

Fig. 6. Explanation for Qualifier Formula

4.3 Example

Fig. 7. Example for Qualifier

The example in Fig. 7 taken from the OCL description [BJR97b] expresses that an account number at a given bank either uniquely determines a person or the account number is not connected to a person at all. The defining formula makes use of implicit role names and induced operations as follows: `person: Bank -> Set(Person)`, `bank: Account -> Bank`, `person: Account -> Person`, `account: Bank -> Set(Account)`, and `account: Person -> Set(Account)`.

The formula expresses the following: (1) For all banks `b` and all accounts `a` at this bank, we select from all persons `p` connected with the bank `b` those who possess an account `a'` at bank `b` having the same account number as account `a`. (2) The size of this set of selected persons is either `0` or `1`.

```
Bank->forAll( b |
  b.account->forAll( a |
    b.person->select( p |
      p.account->exists( a' |
        a'.account#=a.account# and a'.bank=b ) )->size>=0
    and
    b.person->select( p |
      p.account->exists( a' |
        a'.account#=a.account# and a'.bank=b ) )->size<=1 ) )
```

5 Aggregation and Composition

5.1 Statements from UML Material

- *[A filled diamond] signifies the strong form of aggregation known as composition.* Notation, page 54, line 9.
- *Composition is a form of aggregation with strong ownership and coincident lifetimes of part with the whole. The multiplicity of the aggregate may not exceed one (it is unshared).* Notation, page 62, line -7.

– *[AggregationKind="aggregate"] The part may be contained in other aggregates.* Semantics, page 18, line -17.
– *[AggregationKind="composite"] The part is strongly owned by the composite and may not be part of any other composite.* Semantics, page 18, line -15.
– *Aggregation [is] a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.* Semantics, page 148, line -13.
– *Both kinds of aggregations define a transitive, antisymmetric relationship, i.e. the instances form a directed, non-cyclic graph. Composition instances form a strict tree (or rather a forest).* Semantics, page 38, line 18.

5.2 General Translation

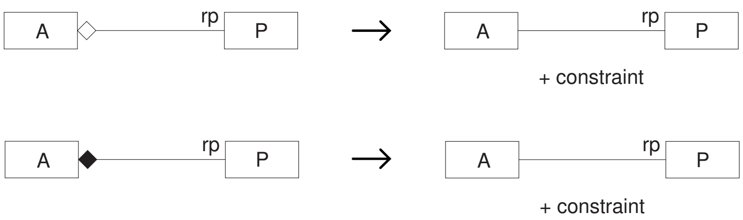

Fig. 8. Transformation Rule for Aggregation and Composition

Aggregate-part (or whole-part) relationships touch three areas to be discussed: Existential dependency, sharing, and instance reflexivity. (1) Existential dependency comes in two facets: The part can existentially depend on the aggregate (a part can only exist inside an aggregate), or, the other way round, the aggregate can existentially depend on the part (an aggregate can only exist if its parts exist). (2) Forbidding sharing means that two aggregates cannot have a part in common. With respect to sharing, one can further distinguish the situation where the two aggregates are objects of the same class from the situation where it is possible that the two aggregates belong to different classes. (3) Forbidding instance reflexivity means that a part cannot be directly or indirectly part of itself.

As indicated in Fig. 8, both UML versions of aggregate-part relationships, namely aggregations and compositions, are translated into binary associations. General associations differ from aggregate-part relationships in that instance reflexivity is forbidden for aggregations and compositions. For compositions we require in addition the part to be existentially dependent from the aggregate and a strong form of forbidding sharing. These requirements are summarized in Fig. 9. Below, we define the facets of aggregate-part relationships independent from these UML requirements as OCL constraints.

UML feature	Requirement in comparison to association
Association	-
Aggregation	Forbidding instance reflexivity
Composition	Forbidding instance reflexivity Existential dependency for the part Strong form of forbidding sharing

Fig. 9. Requirements for Aggregation and Composition

Existential Dependency for the Part: The part is existentially dependent from the aggregate. In technical terms, this means that a `P` object can only exist if it is connected to an `A` object.

```
P->forAll( p |
  A->exists( a | a.rp->includes(p) ) )
```

Existential Dependency for the Aggregate: The aggregate is existentially dependent from the part. In technical terms, this means that an `A` object can only exist if it is related to a `P` object.

```
A->forAll( a |
  P->exists( p | a.rp->includes(p) ) )
```

Weak Form of Forbidding Sharing: A `P` object cannot be shared by two different `A` objects. In technical terms, if two aggregates comprise one common part, then the aggregates coincide.

```
P->forAll( p |
  A->forAll( a, a' |
    ( a.rp->includes(p) and a'.rp->includes(p) )
    implies
    a=a' ) )
```

Strong Form of Forbidding Sharing: A `P` object cannot be shared by two different objects belonging to (potentially) different classes `A1` and `A2`. As shown in Fig. 10, this constraint has to be given for any two classes `A1` and `A2` being potential aggregates for class `P`.

```
P->forAll( p |
  A1->forAll( a1 |
    A2->forAll( a2 |
      ( a1.rp1->includes(p) and a2.rp2->includes(p) )
      implies
      a1=a2 ) ) )
```

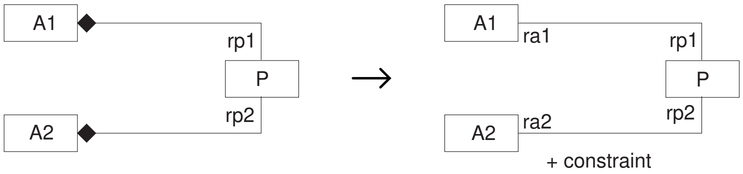

Fig. 10. Transformation Rule for Composition with Strong Form of Forbidding Sharing

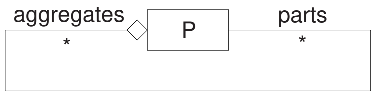

Fig. 11. Reflexive Aggregation

Forbidding Instance Reflexivity for Aggregation: A `P` object cannot be part of itself. In Fig. 11 we have a reflexive aggregation (one class participates twice) where the role names define operations `parts: P -> Set(P)` and `aggregates: P -> Set(P)`. Without any further restriction, a `P` object `p` can be a *direct* part of itself, i.e. `p.parts->includes(p)` is possible. The formula below disallows this. It goes even further by forbidding that p is an *indirect* part of itself by using the transitive closure of the operation `parts`, here called `partsClosure: P -> Set(P)`.[3] The term `partsClosure(p)` yields the parts of `p`, the parts of parts of `p` and so on. The situation in Fig. 11 and the corresponding constraint can be generalized to the case where the reflexivity on the class diagram level is established with some intermediate steps.

```
P.forAll( p | not( p.partsClosure->includes(p) ) )
```

5.3 Example

The example in Fig. 12 shows one ordinary association, one aggregation, and two compositions. As an example for an ordinary association, we see that a paper can be connected to a conference (for instance, submitted to a conference). The components of a paper are described by aggregation and composition: A paper has as parts (1) one or more authors (author names), (2) exactly one abstract, and (3) one or more sections. The association between a paper and an author is classified as an aggregation, the ones connecting a paper with abstracts and sections as compositions. This means two different papers can share an author, but an abstract and the paper's sections exclusively belong to one paper, thus sharing is not possible. However, a paper has coincident lifetime with its strong

[3] We assume here that recursive equations in OCL have a unique least fixpoint solution calculated on the basis of set inclusion.

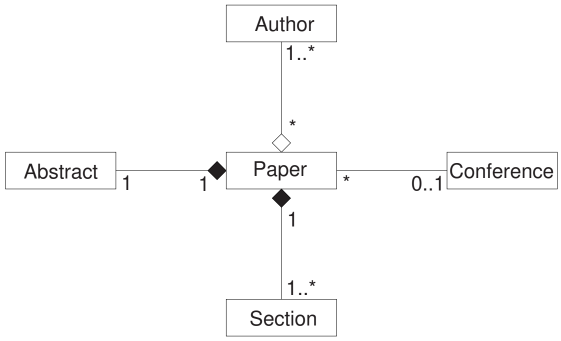

Fig. 12. Example for Ordinary Association, Aggregation, and Composition

components (abstract and Section), but a paper can exist without any connection to a conference (and conferences can exist without being connected to a paper). The difference between modeling a weak or strong component is motivated here by the observation that the connection between a paper and an author seems to be weaker than the one between a paper and an abstract or a section. Instance reflexivity is not applicable in this example because there is no explicit reflexive association.

Existential Dependency for Composition: A paper component classified by composition is existentially dependent from the paper, i.e. an abstract or a section cannot exist without a corresponding paper.

```
Abstract->forAll( a |
  Paper->forAll( p | p.abstract->includes(a) ) )
Section->forAll( s |
  Paper->forAll( p | p.section->includes(s) ) )
```

Forbidding Sharing for Composition: An abstract and a section cannot be shared by two different papers. In this example the weak and the strong form of forbidding sharing coincide because neither `Abstract` nor `Section` participate as parts in other compositions.

```
Abstract->forAll( a |
  Paper->exists( p, p' |
    ( p.abstract->includes(a) and p'.abstract->includes(a) )
    implies
    p=p' ) )

Section->forAll( s |
  Paper->exists( p, p' |
    ( p.section->includes(s) and p'.section->includes(s) )
    implies
    p=p' ) )
```

6 Generalization

6.1 Statements from UML Material

- *Generalization is the taxonomic relationship between a more general element and a more specific element that is fully consistent with the first element and that adds information.* Notation, page 67, line 4; Semantics, page 24, line 10; Semantics, page 152, line -5.
- *An instance of the more specific element may be used where the more general element is allowed.* Semantics, page 152, line 3.
- *The following constraints are predefined. Overlapping: A descendent may be descended from more than one of the subclasses. Disjoint: A descendent may not be descended from more than one of the subclasses. Complete: All subclasses have been specified (...); no additional subclasses are expected. Incomplete: Some subclasses have been specified but the list is known to be incomplete.* Notation, page 68, line 7.

6.2 General Translation

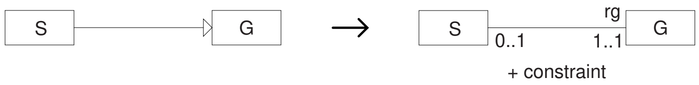

Fig. 13. Transformation Rule for Generalization

As shown in Fig. 13, UML generalizations are transformed to special binary associations. The cardinalities make sure that each specialized object is related with exactly one general object, although not necessarily every general object has a link to a special object. In other words, we have a total mapping from special to general objects. An additional constraint assures the injectivity of the mapping, or, in other words, a special object is associated with a unique general object (no two special objects are associated with the same general object). The defining formula employs `rg` as the role name with an induced operation `rg: S -> G`.

```
S->forAll( s, s' | s<>s' implies s.rg<>s'.rg )
```

This simple translation for generalizations does even handle type substitutability in the context of subtype polymorphism: Wherever an object of the general class `G` is expected (for example as an argument of an operation), we can substitute a specialized object of class `S` after applying the "type cast" `rg`. For example, if `op: boolean` is an operation in class `G`, the expression `s.rg.op` would be allowed for the `S` object `s`.

Generalization relationships can further be classified as disjoint or complete. Due to space limitations we have to refrain from presenting the details, but simply state that equivalent constraints for these cases can be formulated.

Discussion: This translation of generalization into associations seems to be controversial, at least from our referee's point of view. One might argue that Fig. 13 throws *away a useful cognitive tool.* And one may *doubt that the transformation of generalization into an aggregation is appropriate.* However, we had our reasons for taking the explained choice:

1. We do not want to throw away generalization as a concept but we want to indicate how it could be implemented on a lower level. The translation shown is much in the spirit of translating higher level, i.e. semantic, database schemas into lower level data model schemas, for example relational data model schemas.
2. We emphasize that the translation shown does not map generalization into aggregation but into general association.
3. We are not alone in doing such a translation, because one of the standard books on Java [Fla97] mentions a similar translation by taking the viewpoint of "inheritance by delegation":

   We want to implement `GraphicCircle` *so that it can make use of the code we've already written for* `Circle`*. One way to do that is the following:*

   ```
   public class GraphicCircle {
     public Circle c;
     public double area() { return c.area(); }
     public double circumference()
       { return c.circumference(); }
     // new variables and methods
     public Color outline
     public void draw(...) {...}
   }
   ```

   This approach would work, but it is not particularly elegant.
4. We also emphasize that in OCL the expression `rg(s)` with `s` being an arbitrary variable of class `S` can formally be used anywhere where an expression of class `G` is expected. However, this requires that it is always statically known whether we handle an `S` or a `G` object.

6.3 Example

```
Car->forAll( c, c' | c<>c' implies c.vehicle<>c'.vehicle )
```

The example in Fig. 14 shows a specialization of vehicles to cars. The constraint requires, that two given distinct cars are mapped to distinct vehicles by the "type cast" `vehicle` (the operation induced by the association).

Fig. 14. Example for Generalization

7 Conclusion

We have achieved guiding rules for UML designers in order to cope with the UML diagram variety. Our approach can be seen as a way to give semantics to an advanced UML language layer. What remains to be done is to give semantics to the "low-level" UML layer by stating a translation into an abstract model. Such a model together with a semantics for the UML constraint language OCL has already been worked out [RG98].

Our results suggest that some of the UML class diagram concepts do not really increase the modeling power but merely serve as shortcuts for existing techniques. We have translated all discussed UML features into n-ary associations with additional constraints. Due to the modeling power of associations and OCL this seems appropriate, especially because the association concept is a very general one which is able to model many situations. But one can go even further and transform all n-ary associations into an additional class (plus the classes given before) resolving the n-ary association into n binary associations. This is subject to further consideration.

Acknowledgments

The comments of the referees have helped to improve this paper.

References

[BC95] R. Bourdeau and B. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.

[BCV96] E. Bertino, D. Castelli, and F. Vitale. A Formal Representation for State Diagrams in the OMT Methodology. In K.G. Jeffery, J. Kral, and M. Bartosek, editors, *Proc. Seminar Theory and Practice of Informatics (SOFSEM'96)*, pages 327–341. Springer, Berlin, LNCS 1175, 1996.

[BHH+97] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a Formalization of the Unified Modeling Language. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP'97)*, pages 344–366. Springer, Berlin, LNCS 1241, 1997.

[BJR97a] Grady Booch, Ivar Jacobson, and James Rumbaugh, editors. *Object Constraint Language Specification (Version 1.1)*. Rational Corporation, Santa Clara, 1997. `http://www.rational.com`.

[BJR97b] Grady Booch, Ivar Jacobson, and James Rumbaugh, editors. *UML Notation Guide (Version 1.1)*. Rational Corporation, Santa Clara, 1997. `http://www.rational.com`.

[BJR97c] Grady Booch, Ivar Jacobson, and James Rumbaugh, editors. *UML Semantics (Version 1.1)*. Rational Corporation, Santa Clara, 1997. `http://www.rational.com`.

[BLM97] J.C. Bicarregui, Kevin Lano, and Tom S.E. Maibaum. Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP'97)*, pages 324–343. Springer, Berlin, LNCS 1241, 1997.

[Ebe97] Jürgen Ebert. Integration of Z-Based Semantics of OO-Notations. In Haim Kilov and Bernhard Rumpe, editors, *Proc. ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*. Technische Universität München, Informatik-Bericht TUM-I9725, 1997.

[FBLPS97] R.B. France, J.M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML Type Structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 247–260. Chapman and Hall, London, 1997.

[Fla97] D. Flanagan. Java in a Nutshell. O'Reilly, Cambridge, 1997.

[GPP98] Martin Gogolla and Francesco Parisi-Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations. In Bernhard Rumpe, Manfred Broy, Derek Coleman, and Tom S.E. Maibaum, editors, *Proc. ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, 1998. `http://www4.informatik.tu-muenchen.de/~rumpe/icse98-ws`.

[Lan96] Kevin Lano. Enhancing Object-Oriented Methods with Formal Notations. *Theory and Practice of Object Systems*, 2(4):247–268, 1996.

[Öve98] Gunnar Övergaard. A Formal Approach to Relationships in the Unified Modeling Language. In Bernhard Rumpe, Manfred Broy, Derek Coleman, and Tom S.E. Maibaum, editors, *Proc. ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, 1998. `http://www4.informatik.tu-muenchen.de/~rumpe/icse98-ws`.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, editor, *Proc. Int. Conf. Entity-Relationship Approach (ER'98)*, LNCS, Springer, 1998.

[SF97] M. Shroff and R.B. France. Towards a Formalization of UML Class Structures in Z. In *Proc. 21st Annual Int. Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651. IEEE, 1997.

[WRC97] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proc. 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 45–55. ACM Press, 1997.