Mapping UML Models incorporating OCL Constraints into Object-Z

David Roe, Krysia Broda, and Alessandra Russo

Department of Computer Science, Imperial College London
180 Queen's Gate, London SW7 2BZ
{ar3}@doc.ic.ac.uk

Imperial College Technical Report No. 2003/9

Abstract. Focusing on object-oriented designs, this paper proposes a mapping for translating systems modelled in the Unified Modelling Language (UML) incorporating Object Constraint Language (OCL) constraints into formal software specifications in Object-Z. Joint treatment of semi-formal model constructs and constraints within a single translation framework and conversion tool is novel, and leads to the generation of much richer formal specifications than is otherwise possible. This paper complements previous analyses by paying particular attention to the generation of complete Object-Z structures. Integration of proposals to extend the OCL to include action constraints also boosts the expressivity of the translated specifications. The main features of a tool support are described.

Keywords. UML, OCL, Object-Z, constraints, actions.

1 Introduction

The Unified Modelling Language (UML) provides a graphical notation to express the design of object-oriented software systems. It has become the *de facto* industry standard for software design, its widespread use encouraged through intuitive appeal, considerable educational resources, and the availability of CASE tools. Quality and consistency in design is enhanced through the development of a rolling UML standard under the auspices of the Object Management Group (OMG) [OMG01].

UML class diagrams are not usually sufficiently precise to set out all relevant aspects of a specification. Beyond straightforward constraints, for example concerning association multiplicities, there exist a range of complex and sometimes subtle restrictions that are not easily conveyed in diagrammatical form. The Object Constraint Language (OCL) has been integrated into the UML standard as a means of precisely expressing side-effect-free constraints on models. Broadly, these fall under the headings introduced by Meyer [M88]. Invariants are used to express a range of restrictions over class features. System behaviour is then clarified through pre and postconditions, describing, respectively, the applicability and impact of particular operations.

On the other hand, formal specification languages are intended to provide precise and complete models of proposed software systems. Their goal is the unambiguous description of system structure and functionality [DR00]. Like other formal techniques, Object-Z, in particular, employs strict notations of mathematics and logic that permits rigorous analysis and reasoning about the specifications. Its main strength lies in providing a clear means of establishing consistency between model design and implementation, as well as a refined staging post towards executable code.

Semi-formal modelling methodologies and formal specifications therefore should be seen as complementary techniques, each with their own particular strengths and limitations. Bruel and France [BF98] characterize informal structured techniques (ISTs) as emphasizing ease-of-use and understandability at the cost of rigor, and

formal specification techniques (FSTs) as emphasizing formality at the expense of ease-of-use and understandability. As such, there are clear benefits in developing integrated methodologies.

Previous research has helped to identify weaknesses in the semi-formal approach. In particular, it has been recognised that some UML modelling constructs lack precise semantics, sometimes leading to differences of interpretation, and therefore inconsistencies between design and implementation [FELR98]. Conceptual mapping of UML constructs with Object-Z has helped to clarify such issues, by highlighting contentious areas and enforcing an informed interpretation of meaning [KC00]. We believe that previous analysis has provided a sufficient semantic base for the UML in which model constraints can now be investigated and mapped.

This paper provides a mapping from an integrated model of UML class diagrams and OCL specifications into an Object-Z formal specification. This mapping builds upon existing work on the formal specification of diagrammatical modelling constructs, which enables the generation of class skeletons in Object-Z, and extends it by giving a joint treatment of model constructs and constraints. This leads to richer formal specifications than has otherwise been proposed. Arguably, joint treatment of UML constructs and OCL constraints may be seen as a prerequisite for the development of practical, but formal, modelling tools, which would allow rigorous reasoning about the system models developed by the software engineers.

By building upon the most popular informal modelling techniques, formal specification may be seen, as it should be, as a natural process of model refinement as opposed to a competing design paradigm. Overall, it is hoped that continued analysis of the links between informal and formal modelling methodologies should lead to a more mature understanding and use of both technologies, both in the industrial and educational contexts.

The remainder of this paper is organised as follows. Section 2 describes the translation mechanism of UML models with OCL constraints into Object-Z. Section 3 presents additional features to the OCL aimed at the generation of complete Object-Z structures, and also shows how recent proposals to extend the OCL to include action constraints may be applied to the formal specification domain. Section 4 describes the main features of a developmental tool support. Section 5 concludes the paper by comparing our work with existing approaches to methods integration and highlighting some areas for further work.

2 Mapping UML and OCL Constructs into Object-Z

The aim of our work is to define a mapping between UML class diagrams incorporating OCL constraints and Object-Z formal specifications. The choice of Object-Z as our target formal specification language has been driven by the following considerations. Firstly, Object-Z provides a uniform representation formalism and semantics for expressing both diagrammatical models (e.g. UML class diagrams) and constraints (e.g. OCL expressions). Secondly, the language preserves most features of the object-oriented structure of informal UML/OCL models. This has two main advantages: (i) Object-Z models seem to be more accessible to software engineers than any other standard formal specification language such as pure first-order logic, and (ii) errors detected within the Object-Z specification produced by our mapping could more easily be traced back into the initial UML/OCL model.

The mapping is here described informally but defines a unique relation between UML and Object-Z features, as formalised within the translation tool described in Section 4. The underlying functional translation does not cover every construct of the UML standard, but focuses on class diagram models. The mapping of the UML diagrammatical modelling constructs mainly defines class signatures for the resulting Object-Z formal specification, providing the concrete syntax employed in the translated OCL constraints. Only a few features of the class diagram, such as multiplicity of attributes and associations, give rise to additional Object-Z predicates. In addition, certain restrictions have been imposed over the UML standards in order to guarantee the integrity of the translated specifications.

In Section 2.1 we present our mapping between UML class diagrams into Object-Z skeleton structures and signatures, following Kim & Carrington's Object-Z semantic definition of UML class diagrams [KC00]. These structures provide the formal context for the examination and translation of associated OCL constraints. In Section 2.2, we then present the mapping between the OCL constraints into Object-Z predicates within the context of the structures generated from the UML diagrams.

2.1 Formalising UML Model Constructs

The translation of the UML class diagrams (without OCL) into Object-Z structures is presented here using examples based on the UML diagram given in Figure 1.

Classes. Consider the simple UML class diagram of Figure 1. Ordinary UML classes like *Account* and *Person* may be mapped into an Object-Z class construct of the same name, with class features transcribed to the enclosed schemas defining state variables, constants and class operations. Features marked public (+) are included within the class construct visibility list, while those that are unadorned or marked private (-) are not.

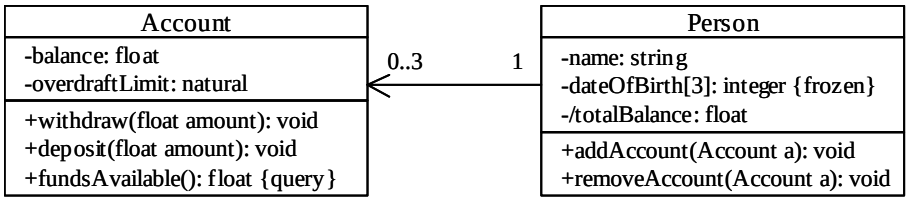


Fig. 1. A UML class diagram for persons and bank accounts

Attributes. UML attributes are mapped as variables of the same name, declared within the state schema of the corresponding Object-Z class construct or within the separate constant definition schema when marked with UML's *{frozen}* property string. Attribute type declarations are required for translation to Object-Z, which supports a range of well known domains corresponding to most basic programming types.[1]

User-defined classes may also be employed as types within UML models and Object-Z specifications; for example, a person's *sex* might have been enumerated (male, female) within the UML model corresponding to the definition of a named domain, *Sex = {male, female}* in Object-Z.

Attributes with multiplicities greater than one may be mapped as finite sequences of the base UML type, combined with a cardinality restriction. A person's *dateOfBirth* attribute therefore corresponds to the declaration of the state variable *dateOfBirth: seq Z* and predicate *#dateOfBirth = 3*. Derived attributes, marked (/) in the UML, are distinguished from primary variables within Object-Z schema through the Δ separator.

Operations. UML class operations may be translated as individual Object-Z operation schema with the same name, with parameters and return values mapped as input and output communication variables adorned (?) and (!) respectively. Although parameter names are optional within the UML, and return values are not named, both must be supplied for the purposes of translation to Object-Z. As with attributes, UML operations marked public are included within the class construct visibility list.

Based on the discussion so far, Figure 2 provides a translated class skeleton for class *Account*[2].

---

[1] Indeed, since the UML is not prescriptive on types, there seems no reason why specialised Object-Z domains should not be included directly in UML class diagrams, even where implementation languages lack direct support. For example, class *Account*'s *overdraftLimit* has been specified as a non-negative quantity (natural number) in Figure 1.

[2] The visibility list in Object-Z models is here denoted with the symbol "|".

Associations may be represented through the instantiation of additional state attributes in Object-Z, depending upon the navigability specified across the UML association line.[3] Figure 1 depicts navigability from class *Person* to class *Account*, implying an additional attribute within the Object-Z class *Person*. Its name is mapped from the target class rolename (since none is specified in this example, *account* by default) and its type is the power set of the target class. Bi-directional associations are mapped as if they were two separate uni-directional associations. Association multiplicities are reflected in additional state axioms constraining the size of such sets, in this case *0 <= account <= 3*. Figure 3 provides a mapping for class *Person*, reflecting the navigable association with class *Account*. The class association management operations are described later.

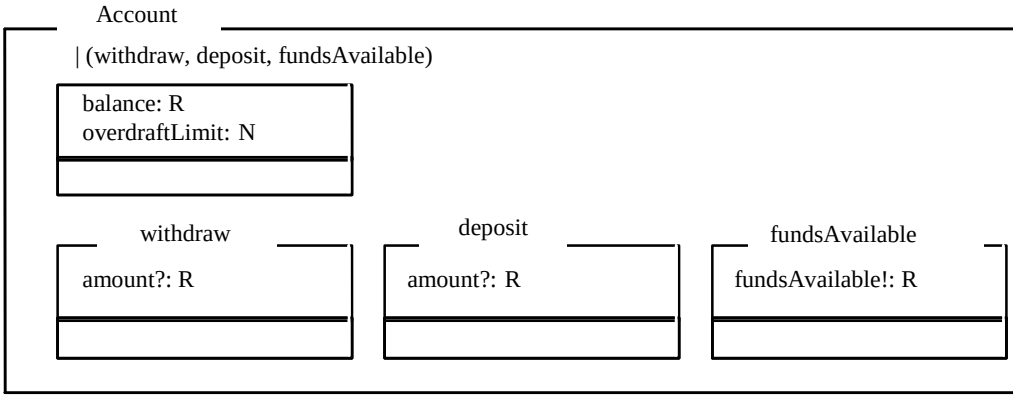
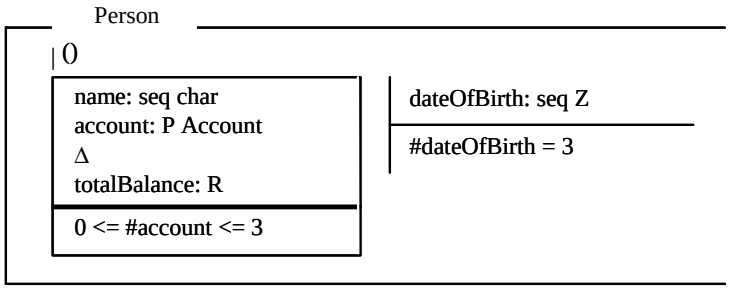


Fig. 2. Object-Z class skeleton for class *Account*.

Aggregation and composition. UML's open diamond merely distinguishes the compound class from its parts, "no more, no less" [BRJ99]. Translation of aggregations therefore proceeds much as for ordinary associations, with the compound class construct containing an additional state variable of type power set of the part class. UML composition, by contrast, implies that instances of the part class may belong to just one instance of the compound class. Mapping is straightforward in that Object-Z provides a notational shorthand (©) denoting unshared containment. Composition between an account and the transactions made on that account, for example, may be captured through the declaration of a state variable *transactions: Transaction*© in the state schema of class *Account*.



[3] This treatment assumes that control and management of the association is locally based. Alternatively, associations may be centrally controlled through a separate database structure maintaining a list of link instances. Roe [DR02] outlines proposals for specifying central control of associations within a UML model, and provides a mapping to Object-Z.

Fig. 2. Object-Z class skeleton for class *Person* reflecting the UML association with class *Account*

Association classes permit class like features to be added to UML associations. Such classes may be formalised in Object-Z as described above, but with the addition of two state variables corresponding to the rolenames and types of the classes participating in the association. Depending upon the navigability specified across the association line, the participating class constructs will contain an additional attribute whose type is a power set of the association class, and constrained in size by the multiplicity specified at the opposite association end.

Generalisation. Mapping of UML generalisation is straightforward in that Object-Z provides a simple notation denoting inheritance, with child classes naming inherited classes just below their visibility list. Specialised subclass features may then be mapped as described earlier.

2.2 Translating OCL Constraints into Object-Z Notation

OCL is intended as a precise, unambiguous language but at the same time one that may be used by mainstream practitioners of object technology. As a result, its designers consciously avoided the very strict notations employed in formal specification languages including Object-Z. Warmer and Kleppe [WK99] argue that "all experience with formal or mathematical notations leads to the same conclusion: The people who can use the notation can express things precisely and unambiguously, but very few people can really use such a notation".

In the context of full grammars, nevertheless, differences between OCL constraint and Object-Z predicate notations are not substantial. Moreover, a fairly clear distinction can be made between restrictions specified over single-valued model features (e.g. a class attribute), and those concerning higher order groups or collections of objects (e.g. an association set). In the former case, OCL expressions are likely to require only modest amendment to form valid Object-Z predicates. But in the latter case, OCL and Object-Z syntaxes diverge, precisely because formal notations dealing with sets and sequences are typically more complex.

Mapping basic OCL features. OCL constraints begin with a statement of the constraint context, either a model class in the case of invariants or a class operation in the case of preconditions and postconditions. This is followed by a keyword indicating the constraint type and a Boolean expression formalising the constraint in terms of model features accessible from the contextual class. Returning to Figure 1, class *Account* may be formalised in OCL as follows:

```
context Account
  inv: balance + overdraftLimit >= 0


context Account::withdraw(float amount): void
pre: amount <= balance + overdraftLimit
post: balance = balance@pre - amount

context Account::deposit(float amount): void
post: balance = balance@pre + amount

context Account::fundsAvailable(): float fundsAvailable
post: result = balance + overdraftLimit
```

OCL invariants are mapped into either predicates of the class state schema or predicates of the class constant definition schema, describing restrictions which must be satisfied by all instances of a particular class at all times. Preconditions and postconditions meanwhile are transcribed to the predicate section of the corresponding Object-Z operation schemas, again with identical semantics. Preconditions must be satisfied just before execution of the operation and postconditions must hold on completion. No attempt is made in either technology to describe what action should be taken in the event that a constraint is broken.

Joint mapping of the contextual model constructs ensures that the concrete constraint terms will have meaning within equivalent Object-Z predicates. And in these examples, the operations have universal meaning. The boolean expression components of these and similar OCL constraints therefore also form valid Object-Z predicates with only very minor modifications, as shown in Figure 4. References to object attributes in OCL postconditions are translated in primed (after execution) format, except where appended by the *@pre* keyword. Similar references in OCL preconditions are translated into an unprimed format. Return value names are substituted for OCL's *result* keyword in Object-Z.

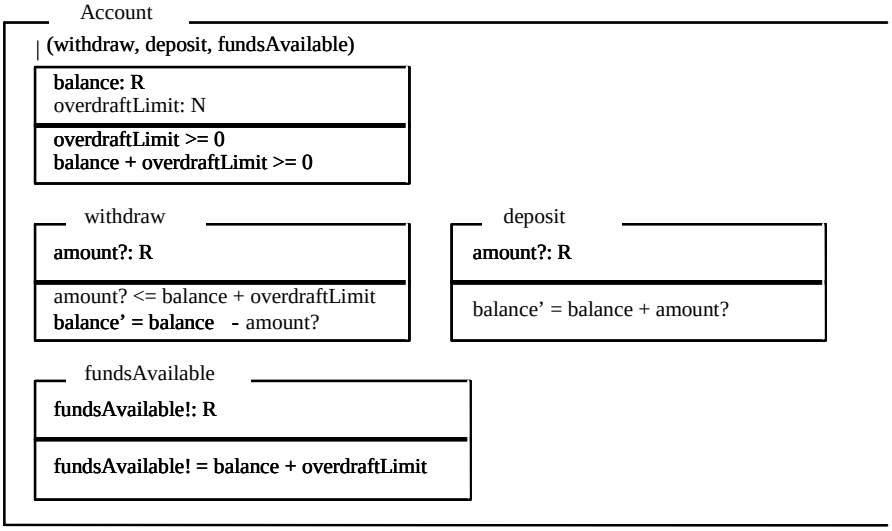


Fig. 3. Class *Account* with state and operation predicates

OCL constraints may also contain references to contextual class operation that return values. Because OCL is intended to be side-effect-free, only references to query operations are permitted, i.e. those operation that return a value but do not alter an object state. The mapping of such constraints will replace the references to the query operations with the translation of the right-hand side of their respective result post-conditions.

For instance, consider the OCL constraint

```
context smallCircle::Area(): integer
  post: result = Π * radius * radius
context smallCircle
inv: Area() ≤ limit
```

The mapping will include in `smallCircle` class schema the invariant

```
Π * radius * radius ≤ limit
```

OCL supports a range of predefined basic types including Booleans, integers, real numbers and strings. In each case, standard operations over these types are predefined in OCL and Object-Z and so notational translation is largely a matter of substitution of formal symbols in place of the natural language operators preferred in OCL. Boolean operators, for example, are written naturally in OCL: *not*, *and*, *implies* and so on. Basic type literals, for example *true*, *2.4* or *'hello'*, may be used freely in both OCL and Object-Z constraints.

Mapping of OCL Constraints over Collections of Objects. Most UML associations define a relationship between one object and a group of other objects. OCL provides an abstract type *Collection*, and three concrete collection types, set, sequence and bag, each supporting predefined operations facilitating the expression of constraints over groups of objects. Table 1 summarises those operations defined over all OCL collection types, and provides equivalent Object-Z notation[4]. Operations shown in the

[4] See Warmer and Kleppe [WK99] for descriptions of operations.

lower half of the table employ an iterator variable, which scans across all elements in the collection returning either a new collection or Boolean value as a result.

Table 1. Operations defined over all OCL collection types

Operation : description *OCL syntax*	*Object-Z notation*[5]
size() : the number of elements in the collection	
collection->size()	*#collection*
count(object) : the number of occurrences of an object in the collection	
collection->count(object)	*#{ c:collection | c = object }*
includes(object) : true if the object is an element in the collection	
collection->includes(object)	*object ∈ collection*
includesAll(collection) : true if all elements of the parameter collection are elements of the current collection	
collection->includesAll(parameter)	*parameter ⊆ collection*
isEmpty() [**notEmpty**()] : true if the collection contains no/one or more elements	
collection->isEmpty()[notEmpty()]	*collection =[<>] ∅*
sum() : the sum of all elements in the collection where the element type supports addition	
collection->sum()	*Σc:collection*
collection->sum(feature)	*Σc:collection • c.feature*
Operations that iterate over the elements in a collection[6]:	
select(*b-exp*) [**reject**(*b-exp*)] : results in all elements (e) in the collection for which <*b-exp*> is [not] true	
collection->select[reject](e | <b-exp>)	*{e | [¬]<b-exp> }*
collect(*expression*) : results in a new collection, derived from, but containing different objects to the original collection based upon <*expression*>.	
collection->collect(e | <expression>)	*{e:collection | <expression>}*
forAll(*b-exp*) : true if < *b-exp* > is true for all elements in the collection	
collection->forAll(e | <b-exp>)	*∀e:collection • <b-exp>*
exists(*b-exp*) : true if < *b-exp* > is true for at least one element in collection	
collection->exists(e | <b-exp>)	*∃e:collection • <b-exp>*

OCL also supports a range of well known operations specific to sets such as *union*, *intersect*, *minus*, *symmetricDifference*, *including* and *excluding*. Specialised sequence operations likewise are provided in OCL including *first*, *last*, *at*, *append* and *prepend*. In all cases translation to Object-Z notation largely entails the straightforward substitution of formal symbols for natural language operator names. OCL set and sequence literals, e.g. `Set{1..10}` or `Sequence{23,11,67},` likewise require only minor amendment in Object-Z notation.

These notational transformations are best illustrated with some examples. Returning to class *Person* of Figure 1, derivation of the secondary variable *totalBalance* and the class association management operations may now be formalised in OCL as follows:

```
context Person
  inv: totalBalance = account->sum(balance)

  context Person::addAccount(Account a): void
  pre: not(account->includes(a))
  post: account = account@pre->including(a)

  context Person::removeAccount(Account a): void
  pre: account->includes(a)
  post: account = account@pre->excluding(a)
```

[5] Object-Z translations employ set limiters {} as required. Table 1 collection operations also apply to sequences.

[6] In a fuller syntactic form, the iterator type may also be specified. In a translation context, this is only helpful in the case of OCL's generic *iterate* collection operation. This is considered beyond the scope of this paper.

Adding some further constraints for illustrative purposes

```
context Person
inv: account->exists(a | a.balance >= 0)
--persons may not simultaneously overdraw all accounts

context Person
inv: account->forAll(a | a.overdraftLimit <= maxLimit)
--each person's accounts are subject to some maximum
  overdraft limit
```

Based on the patterns described in Table 1, the earlier skeleton construct for class *Person* may now be enriched with a variety of translated state and operation predicates, as shown in Figure 5. OCL invariants are transcribed to the lower predicate section of the class state schema, and pre and postconditions to the relevant operation schema. In general, translation of OCL constraints over association partners according to these basic patterns poses few problems. However, it is worth noting that where features of association partners are referenced in OCL constraints, such features should be both strictly navigable and also visible to the external environment in order to guarantee correctness of the corresponding Object-Z predicates.

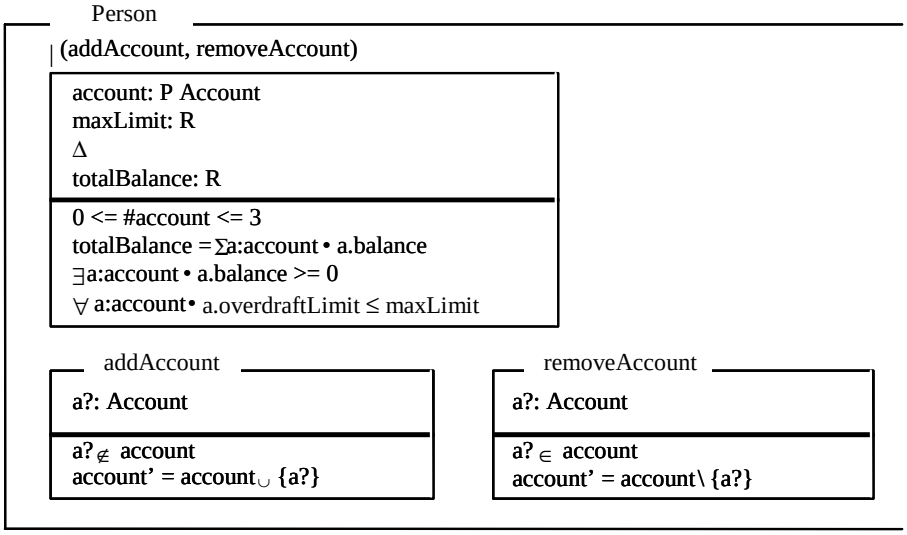


Fig. 4. Class *Person* with state and operation predicates

3 Extensions

This paper has highlighted some modest restrictions to the UML and OCL standards needed to guarantee correctness of the translated Object-Z specifications: declaration of attribute types are required; likewise, operation parameters and return values must be named. Some further proposals are outlined below, in a translation context, each aimed at the generation of complete and more expressive Object-Z structures. The tool support described in Section 4 incorporates most of these features.

3.1 Initial Configuration

Object-Z class constructs also contain an initial schema (always named INIT) which sets out additional predicates involving class features. An object is said to be in its initial configuration whenever the values of the class features satisfy the initial schema predicates.

UML attribute declarations may optionally specify a default initial value, which might obviously be mapped as a class initial schema axiom. However, initial conditions might not be restricted solely to statements of equality. Moreover, they may equally apply to object associations. For example, we may wish to specify that

instances of class *Person* initially have no bank accounts. In either case, such conditions cannot be expressed easily in the UML.

It is proposed that initial conditions be expressed using OCL definition constraints. These are normally used to define pseudo-attributes with the aim of simplifying and avoiding repetition in complex OCL expressions, but provide a convenient mechanism for expressing initial configurations. Taking the example above, we may write:

```
context Person
def: let INIT : boolean = (account->isEmpty())
```

which translates as the initial schema predicate:

$account = \varnothing$

The OCL expression defines a pseudo-attribute *INIT* of type boolean, like Object-Z's INIT schema evaluating true when a person's *account* set is empty and false otherwise. This amounts to an elevation of the *INIT* classifier to reserved status in OCL definition constraints. Beyond this, ordinary OCL definition constraints would have to be mapped as full attributes or operations of the corresponding Object-Z class construct. In the context of translations to formal specifications, it is suggested that such features are better specified within the UML model itself.

3.2 Operation delta-lists

Consider the following hypothetical operation formalised in OCL:

```
context someClass::changeState(): void
post: attribute1 = attribute2
```

Postconditions are merely statements of what is true on completion of an operation. Tempting as it is to read this postcondition left to right as an assignment, it is not possible to know which, or indeed whether both, of the attributes will change when the *changeState* method is executed. The impact of the operation could be clarified through additional postconditions stating explicitly which class features do not change, but this requirement is heavy in the case of sophisticated classes. Under the current standard therefore, OCL operation constraints retain a degree of ambiguity.

Object-Z operation schemas simplify the problem by including a Δ-list, explicitly stating those state variables subject to change during the execution of an operation. In order to facilitate a straightforward translation, it is proposed that the OCL syntax be extended to include a new keyword *modifies* in the context of operations. Class *Account*'s *withdraw* operation would, for example, be clarified as follows:

```
context Account::withdraw(float amount): void
modifies: balance
--pre and postconditions as before
```

Drawing on the standard UML notation, those operations that do not alter an object's state may be marked *modifies: query* mapping to an empty Δ-list in Object-Z.

3.3 System classes

Dupuy et al [DLC97] propose that each UML class should map to two Object-Z class constructs, the first describing class features and the second describing the set of existing instances of the class and operations on this set. Section 2 of this paper focused on the first aspect, i.e. the translation of UML classes as templates for object instantiation. Like Kim and Carrington [KC00], we believe that the second function is better treated through the specification of a system level class. Such classes are normally explicitly defined within Object-Z specifications but are typically left implicit in a UML class diagram.

The system class may be represented diagrammatically as a class box which physically contains the other model classes, see Figure 6. The use of a new «SystemClass» class stereotype helps to make this role a little clearer. The mapping of the contained constructs proceeds as described in earlier sections. The system class construct will incorporate set attributes corresponding to all the contained model

classes, each constrained by the cardinality restrictions specified in the UML diagram (a maximum of 50 persons is imposed for illustrative purposes).

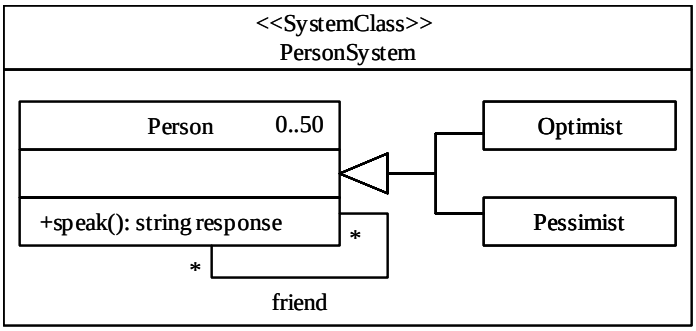


Fig. 5. System-level class containing other model classes

The UML *PersonSystem* system class could, for instance, be accompanied by the following package of OCL constraints:

```
context PersonSystem
inv: person->forAll(p | person->includesAll(p.friend))

context PersonSystem
inv: person->includesAll(optimist)
inv: person->includesAll(pessimist)

context Optimist::speak(): string response
modifies: query
post: result = 'the glass is half full'

context Pessimist::speak(): string response
modifies: query
post: result = 'the glass is half empty'
```

The first constraint reflects the fact that for any class *C*, attributes of the same type *C* should belong to the system sets; in this case, each person's *friend*s must also be members of the system *person* set. The next constraint is needed to denote sub typing; i.e. all optimists and pessimists *are* persons. Polymorphism is reflected in the specialised notation (↓) used in the declaration of the system person set. The remaining constraints formalise polymorphic behaviour within the *Person* class hierarchy. The integrated UML/OCL model given above can then be mapped into the following Object-Z specification shown in Figure 7.

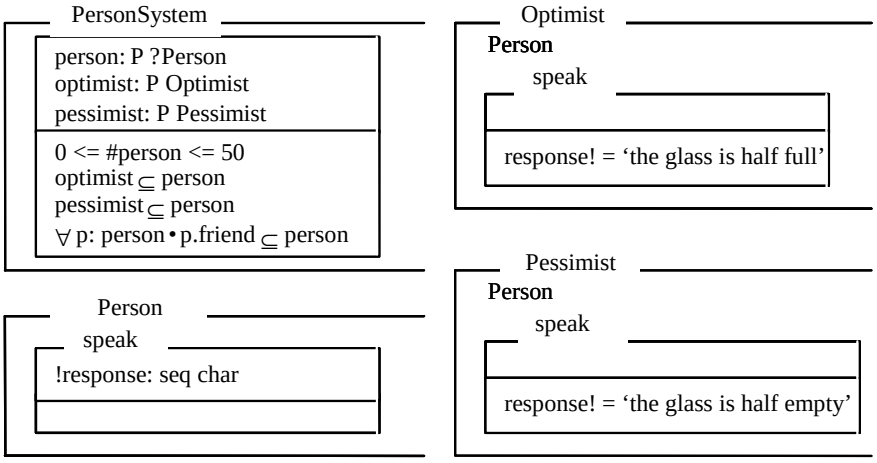


Fig. 6. Object-Z system specification

3.4 Action constraints

The basic OCL constraint types – invariants, preconditions and postconditions – are static. Specifically, there is no means of specifying that events have happened or will happen. Kleppe and Warmer [WK00] have proposed an extension to the current OCL standard to incorporate action constraints. This permits UML operations to be formalised in terms of sets of events or method calls being sent to a target set of receivers accessible or navigable from the sender context. It is claimed that the incorporation of action constraints significantly improves the expressivity of the OCL [WK00]. In this section we propose a way of mapping such action constraints into Object-Z specifications, by capturing the static properties that such constraints would impose on the dynamic system behaviour.

Syntactically, OCL action constraints are specified in terms of a comma-separated list of included actions or events, a target set of receivers, and an optional guard condition:

```
action: [if <condition>] to <targetSet> send <eventSet>
```

In [WK00] it is suggested that such constraints can be used in two different OCL contexts, namely as part of an operation specification and as part of the invariant of a class. In the first case, the condition is evaluated at postcondition time of the operation, and, if true, each event is sent to each target. The effect of this constraint in a given operation is, therefore, to extend its postcondition by taking into account the performance of the action. In the second case, the action constraint explicitly extends a given invariant of a class by specifying events that ought to occur when otherwise the invariant would be violated. As pointed out by Kleppe and Warmer, this second case could equally well be described by introducing a new operation that has the action clause as its postcondition [WK00]. In view of this, our mapping provides a translation of the action constraints as part of an operation specification, which can also be used for translating the second kind of action clause.

An OCL operation specification with action clauses would in general be of the following form:

```
context classname::operation(args)
[pre: pre-expression]
[post: post-expression]
action: [if <condition>] to <targetSet> send <eventSet>
```

We consider first the basic case when the targetSet and the eventSet are singleton sets containing respectively a *target* receiver and an *action*. The mapping consists of four steps:. Step 1 introduces a new operation, say *checkConditionTrue*, for checking the truth of the action condition. This operation is defined as an Object-Z operation expression that reflects the Boolean expression of the condition and uses query methods to access the condition's attributes. Step 2 defines an operation *eventOperation*, as a conjunction operation between the *checkConditionTrue* and the *target.action*; step 3 demotes the current operation to $operation_0$ to include just its basic pre and postconditions, and step 4 redefines the given operation as a sequential operation expression of the form $operation_0$; *eventOperation*. In the case of more than one action clause specification for the same operation, steps 1 and 2 have to be repeated for each action clause, and step 4 should be the sequential composition of $operation_0$ with the conjunction of the different *eventOperations*.

Let us consider the following action clauses example taken from [WK00]:

```
context CustomerCard::invalidate()
[post: valid = false]
action: if Cust.special to Cust send politeInvalidNote()
action: if not Cust.special to Cust send InvalidNote()
```

The mapping will generate within the class `Cust` two operations `IsSpecial` and `IsNotSpecial`, which respectively check whether the attribute *special* is true or false, within the class `CustomerCard` two operations `EventPoliteInvalidNote` and `EventInvalidNote` given by the two conjunction compositions

```
EventPoliteInvalidNote=CheckIsSpecialTrue∧Cust.PoliteInvalidN
ote
EventInvalidNote = CheckIsNotSpecialTrue ∧ Cust.InvalidNote
```

where

```
CheckIsSpecialTrue = Cust.IsSpecial
CheckIsNotSpecialTrue = Cust.IsNotSpecial
```

The final composite operation

```
invalidate=
   invalidate0 ;(EventPoliteInvalidNote ∧ EventInvalidNote)
```

Another example of OCL action constraint involving parameter passing and its translation is given below.

```
context PersonSystem::speakSomeone(Person p): void
action: to p send speak()

EventSomeoneSpeak = [ p?:Person ] • p?.speak
SpeakSomeone = [p?:Person] • p?.self || EventSomeoneSpeak
```

Note that more elaborate conditions, for instance those using references to query operations and/or possibly involving different contexts, step 1 will still generate one single operation *checkConditionTrue* using the mapping process illustrated in Section 2.2.

In more general cases, event clauses could include more than one action and target receiver. In the former case, the current semantics of multiple actions does not necessarily specify whether the actions are executed sequentially or in parallel [WK00]. In our mapping we assume that such actions are in conjunction with each other, in the sense that their overall static effect on the dynamic behaviour of the system is given by the conjunction of their respective post-conditions. In this case, the mapping process for action clauses described above has step 2 extended to allow the definition of an *eventOperation*, as a conjunction operation between the *checkConditionTrue* and the conjunction of the *target.actions* for each action included in the eventSet[7]. For example, the following OCL action clause

```
context Person::compositeOperation(): void
action: to self send listen, speak
```

is mapped into the following Object-Z operation

```
compositeOperation = EventsOperations
```

where `EventsOperations = self.listen ∧ self.speak`

It is important to notice, however, that the conjunctive composition of operations in Object-Z is semantically more powerful than the OCL conjunction, as it assumes that values of input parameters with the same name are equal. In the current language of OCL action constraints there is no means by which such Object-Z semantics for conjunctive composition can be captured. A more general solution for the translation of complex action clauses into Object-Z would require specialised extensions of the OCL action constraint vocabulary to encompass different types of composition of actions in the eventSet, such as sequential (;), conjunction (∧), choice ([]) and parallel (||) compositions. Because of the weaker expressive power of the OCL language such compositions of actions will not necessarily be complete with respect to the corresponding composition operators of Object-Z.

Special cases of action clauses with multiple target receivers are those where multiple receivers are collection type attributes in the UML/OCL model. Returning, for instance, to the *PersonSystem* model of Figure 7, we could have action constraints

[7] Note that if the targetSet includes only one target receiver, each of these *target.action* will refer to the same target, otherwise there will have to be as many as *target.actions* (with different target and different action) to cover the Cartesian product between the targetSet and the eventSet.

defined over the singleton target set <person>, where person is a collection of objects of type Person. Two examples are given below:

```
context PersonSystem::speakAll(): void
action: to person send speak()

context PersonSystem::speakSome(): void
action: to person->select(p|p.opinionIsValued) send speak()
```

In the first case, the target set comprises all objects in the system set *person*, each of which receives the *speak* message. In Object-Z, this is mapped as follows:

```
EventAllSpeak = ∀p:person • p.speak
speakAll = EventAllSpeak
```

In the second case the mapping would generate the following Object-Z operation:

```
EventSomeSpeak = ∀x:{p:person|p.opinionIsValued} • x.speak
speakSome = EventSomeSpeak
```

4 Tool Support

Based upon the mapping described in this paper, a tool support has been implemented in order to provide an automatic generation of comprehensive Object-Z formal specifications from a user-defined UML class diagram models incorporating OCL constraints. To the best of our knowledge, this is the only tool available supporting a translation of UML model constructs and OCL constraints into Object-Z. Dupuy *et al*'s mappings for OMT static models are supported through a CASE tool [SD], but model constraints must be provided as annotations already specified in the Z language.

The tool comprises of 5 main functionalities: (1) a user input, (2) UML/OCL model validation, (3) translation of (UML) model constructs, (4) translation of the (OCL) model constraints and (5) visualisation of the resulting Object-Z specification. The user interface allows the user to specify UML models incorporating OCL constraints in a straightforward fashion: these models are entered as text files, based upon a customized XML Document Definition Type (DDT). As part of a standard menu/toolbar driven application, the interface offers conventional file storage facilities, and, in particular, provision of a custom text editor, which facilitates more efficient specification and presentation of user input. The user-input is then validated to check whether the UML model does comply with the UML standard.

Assuming that there are no critical errors, the UML/OCL model is compiled into an internal representation of the corresponding Object-Z specification. The compilation process first translates the UML model into an Object-Z skeleton and then translates OCL constraints into Object-Z predicates. This ordering is necessary because all model constraints are context-specific. Even though the tool does not support full OCL syntax and OCL type conformance checking, it does ensure that the translated predicates have meaning within the context of the UML model itself. In addition, a certain degree of control over the compilation of the Object-Z specifications is provided. In particular, users have the option of viewing subclasses in a flattened format, with super-class features imported and conjoined with those of the child. The OCL constraints are translated on a class-by-class basis. In cases of constraint errors detection (e.g. a reference to a non-existent or non-accessible UML model feature), helpful error messages are reported back to the user, and the faulty OCL constraint is included in the final Object-Z specification but with its initial OCL syntax, instead of Object-Z translation. This is because the tool does not offer a visualisation of the initial UML model, and we believe that inspection of the Object-Z output would help the user to understand the causes of the OCL constraint errors. An example of a screenshot of the tool is given in Figure 8.

From an UML perspective, the automated translation covers all of the most commonly used modelling constructs, such as class attributes, operations, associations and generalisation. Fowler [FS00] suggests that such features "will comprise 90 percent of [a modeller's] effort in building class diagrams". The tool support goes

beyond this, covering also features like visibility of attributes, recognition of different kinds of associations, frozen variable parameters, provision for association classes and enumeration types.

From the OCL perspective, the vast majority of OCL features and operations may be used in the input OCL constraints. The main omission are some mathematical operations over integers and real (e.g. “abs”, “max”), “if….then…else”, the collection operations “collect” and “iterate”, and the sequence operation “subsequence”. A preliminary implementation of the action constraints has also been incorporated in the current version of the tool.

A detailed description of the design and Java implementation features of the tool can be found in [DR02]. The implementation and user documentation can be obtained from the first author.

5 Conclusion & Related Work

This paper provides a mapping process of UML models incorporating OCL constraints into full Object-Z specifications. Extensions to UML class constructs have also been suggested, which make the initial UML models more informative and facilitate the generation of well-defined Object-Z classes. A recent extension of the OCL language regarding the use of action constraints is also covered by our mapping and a translation process for such constraints into Object-Z operation have been proposed. A tool support has also been developed, which allows automated generation of Object-Z specifications from (XML) type of UML/OCL model.

A number of approaches have been adopted to integrate formal and informal techniques. The precise UML group (pUML), for instance, focuses on formalising UML by developing rigorous semantics for its modelling constructs [FELR98]. Translation of UML models into formal specifications is not an objective, though the group argues that formalisation of the UML is a prerequisite for such work. Mostly relevant to this paper are Kim and Carrington’ work on Z semantics for UML [KC00], and Beckert *et* al.’s work on first-order semantics for Object Constraint Language [BKS02].

Kim and Carrington have suggested a precise and descriptive semantics for basic UML modelling concepts using Object-Z, and a formal description for UML class constructs. For instance modelling concepts such as “types”, “attributed”, “parameters” have been defined in terms of specific Object-Z classes, basic UML principles of an UML class construct have also been precisely defined as invariants of a special-purpose Object-Z class called “UMLClass” (e.g. attributes have unique names, operations have unique signatures, ect..). Similar specifications have been given for associations. Such formal Object-Z description of UML class constructs provides a precise definition of the UML rules used for developing well-formed class diagram models. Building upon this formal underpinning, Kim and Carrington have then proposed a formal mapping of UML classes and associations into Object-Z classes. Our work builds upon these results and complements them by providing also the translation of OCL constraints within the generated Object-Z skeleton. The formal Object-Z meta-model of UML constructs is implicitly considered in our tool support during the validation phase of the user input, whereas the Object-Z skeleton generated by our mapping is, in most respects, similar to that generated by Kim and Carrington’s formal mapping. However, in the absence of extensions to UML syntax, Kim and Carrington’s resulting Object-Z specifications lack key features such as initial configuration, operation Δ-list, fully specified operation return type, which have instead been included in our mapping by proposing appropriate extension to the UML class constructs. In our opinion, such extensions help complete the UML models. Furthermore, by translating OCL constraints, our mapping provides a complete Object-Z specification with invariants and predicates for operations.

Beckert et al. have proposed in [BKS02] a declarative formal representation of OCL using first-order predicate logic. The main difference between this approach and ours is that we express the formal specification of an UML/OCL model using Object-Z language and therefore relaying on a theorem prover for Object-Z, whereas the

choice of first-order logic as formal language allows the use of existing interactive theorem provers for classical logic. Their choice of formal language has mainly be driven by the need of facilitating formal reasoning and verification of OCL constraints. The driving motivation of our work relies instead on the need of providing refinement of informal system modelling into formal system descriptions, which preserves some of the features (e.g. object-orientation) of the initial informal model, and which can eventually lead to a formal specification for verification purposes. Given the first-order classical logic underpinning of Object-Z specification and the well defined process of generating Object-Z predicates from OCL constraints, we believe that Beckert *et al*. approach could, for instance, be equally applied to our Object-Z specifications, perhaps in a more direct way in order to facilitate automated verification of our resulting Object-Z specifications.

Our formal mapping of UML/OCL model relates therefore to both the above existing work. At the other end of the spectrum, there are efforts to introduce more intuitive graphical extensions to formal methods, including Dick and Loubersac's [DL91] treatment of VDM and Duke and Rose's [DR00] extensions to UML class diagrams capturing Object-Z's promotion and composition of operations using the sequential, conjunctive, choice and parallel operators.

Both the mapping and the tool support presented in this paper could further be extended. The mapping of UML models could, for instance, be expanded to cover the translation of more complex UML constructs such as interfaces, abstract classes, qualified associations and static class features. We believe that all these constructs could find a corresponding representation in Object-Z specifications. The mapping process of the OCL constraints could also be extended to include the translation of OCL complex action constraints. For instance, the OCL language, and correspondently our mapping, could be extended to allow the use of composite operators such as parallel, sequential and disjunctive operators. This would allow not only a sound and complete translation of OCL action constraints into Object-Z composite operations, but also a more expressive way of describing system's actions within the OCL specification. Finally, our tool could also be enhanced to cover in particular a more exhaustive automatic translation of OCL action constraints, as well as the mapping of those OCL features mentioned in Section 4 which are not yet implemented.

Overall, it is hoped that by building upon the most popular informal modelling technique, this work can contribute towards the overall aim of making formal specifications into a natural process of model refinement as opposed to a competing design paradigm. The analysis shown in this paper of the links between informal and formal modelling methodologies provides a means for gaining more mature understanding and use of both technologies. For modellers familiar with either the UML or Object-Z but not both, the conceptual analysis described in this paper and the possible use of an associated tool offers a way to familiarise with a new object-oriented technology.

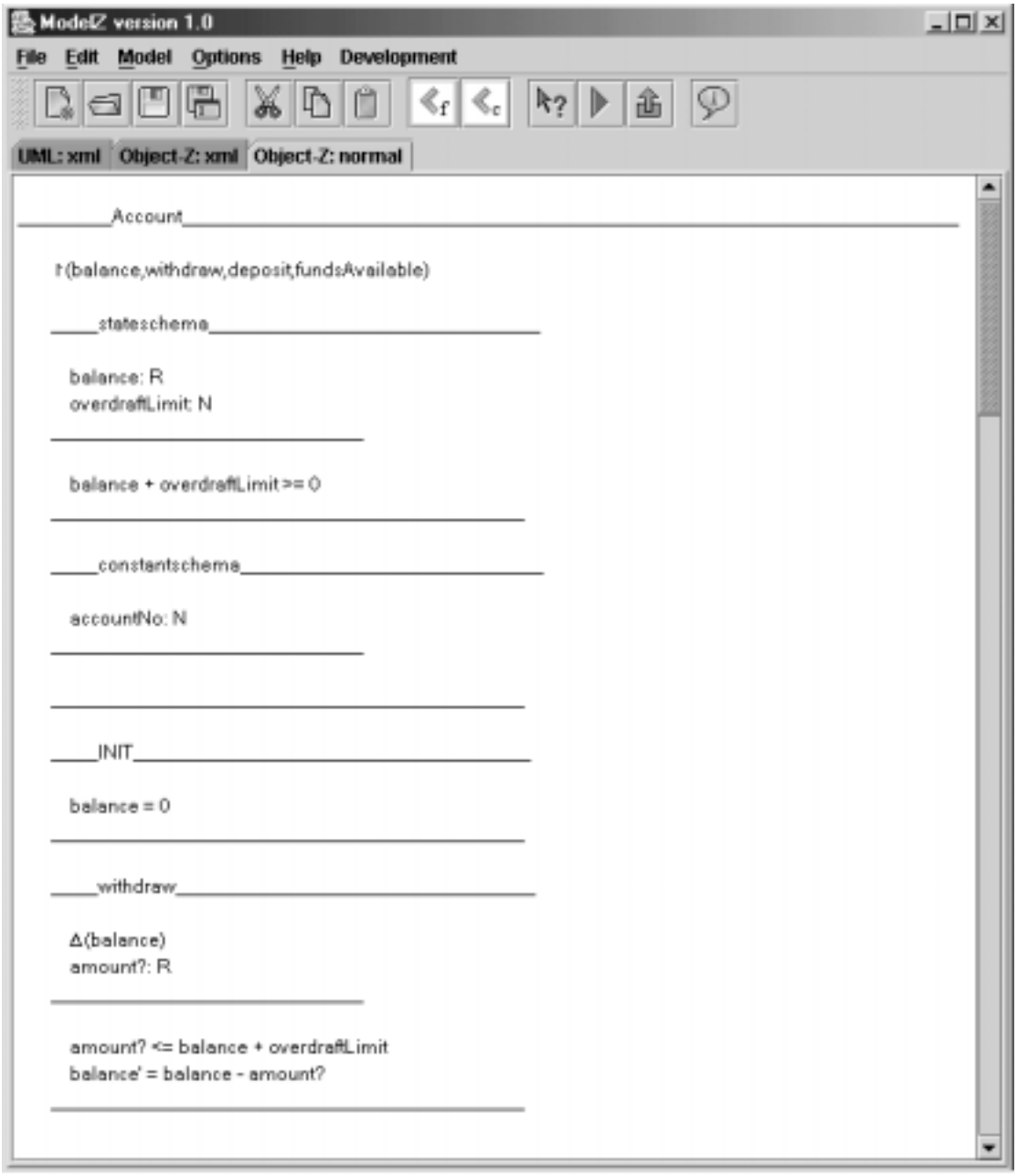


Fig. 7. Example run of the generation of an Object-Z class using our tool support

References

[BKS02] B. Beckert, U. Keller and P. Schmitt, "Translating the Object Constraint Language into First-order Predicate Logic", to appear in *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FloC), Copenhagen, Denmark*, 2002.

[BRJ99] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.

[BF98] J-M. Bruel and R. France, "Transforming UML models to Formal Specifications" in L. Andrade, A. Moreira, A. Deshpande, S. Kent (eds), *Proceedings of the OOPSLA'98 Workshop on Formalizing UML: Why? How?, October 1998, Vancouver, Canada*.

[DL91] J. Dick and J. Loubersac, "Integrating Structured and Formal Methods: a visual approach to VDM" in *3rd International Conference ESEC'91, October 1991, Milan, Italy*, Lecture Notes in Computer Science, Volume 550, Springer, 1991.

[DR00] R. Duke and G. Rose, Formal Object-oriented Specification using Object-Z, MacMillan Press, 2000.

[SD] S. Dupuy. RoZ version 0.3: an environment for the integration of UML and Z, www-lsr.imag.fr/Les.Groupes/PFL/RoZ/.

[DLC97] S. Dupuy, Y. Ledru and M. Chabre-Peccoud, Integrating OMT and Object-Z in *Proceedings of BCS FACS/EROS ROOM Workshop*, 1997.

[FS00] M. Fowler with K. Scott, *UML Distilled (second edition)*, Addison-Wesley, 2000.

[FELR98] R. France, A. Evans, K. Lano and B. Rumpe, "Developing the UML as a Formal Modelling Notation", *Computer Standards and Interfaces*, No 19, 1998.

[KC00] S. Kim and D. Carrington, "A Formal Mapping between UML Models and Object-Z Specifications" in J. Bowen, S. Dunne, A. Galloway and S. King (eds), *ZB2000: Formal Specification and Development in Z and B, First International Conference of B and Z users, York, UK, Aug/Sept 2000, Proceedings*, Lecture Notes in Computer Science, Volume 1878, Springer, 2000.

[M88] B. Meyer. *Object-oriented Software Construction*, Prentice-Hall, 1988.

[WK00] A. Kleppe and J. Warmer, "Extending OCL to Include Actions" in A. Evans, S. Kent, B. Selic (eds), UML 2000 – The Unified Modelling Language, Advancing the Standard, Third International Conference, York, UK, October 2000, Proceedings, Lecture Notes in Computer Science, Volume 1939, Springer, 2000.

[OMG01] Object Management Group, Unified Modelling Language(UML), Version 1.4, www.omg.org, 2001.

[DR02] D. Roe, "Mapping UML Models incorporating OCL Constraints into Object-Z", MSc individual dissertation, Imperial College, 2002.

[SF97] M. Shroff and R. France, "Towards a Formalization of UML Class Structures in Z" in COMPSAC, *Proceedings, 21st International Computer Software and Applications Conference (COMPSAC'97), August 1997, Washington DC.*

[WK99] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1999.