Abstract Data Types and UML Models*

Heinrich Hussmann[1], Maura Cerioli[2], Gianna Reggio[2], Francoise Tort[3]

[1] Department of Computer Science, Dresden University of Technology, Germany
[2] DISI, Università di Genova, Italy
[3] LSV, ENS Cachan, France
Email: hussmann@inf.tu-dresden.de

Abstract. Object-oriented modelling, using for instance the Unified Modeling Language (UML), is based on the principles of data abstraction and data encapsulation. In this paper, we closely examine the relationship between object-oriented models (using UML) and the classical algebraic approach to data abstraction (using the Common Algebraic Specification Language CASL). Technically, possible alternatives for a translation from UML to CASL are studied, and analysed for their principal consequences. It is shown that object-oriented approaches and algebraic approaches differ in their view of data abstraction. Moreover, it is explained how specification methodology derived from the algebraic world can be used to achieve high quality in object-oriented models.

1 Introduction

It is a frequently made assumption that object-orientation is based on the principles of data encapsulation and data abstraction. In this paper, we closely examine the relationship between object-oriented modelling and the classical algebraic approach to data abstraction. Further below, we will come to the conclusion that object-oriented specifications of software in general do not follow the classical data abstraction principles in an obvious way. There is a concept mismatch, which leads to problems in translating semi-formal object-oriented specifications into formal algebraic specifications.

In order to be concrete, we use throughout this paper language the international standard language UML (Unified Modeling Language) [RJB99] as an example for a semi-formal object-oriented specification language. We explain the concept mismatch by sketching the principles and problems of a translation from a small part of UML to an algebraic specification language. At the algebraic side of the translation study we use the recently developed language CASL (Common Algebraic Specification Language) [CoFI98].

Besides a clarification and comparison of concepts, a second goal of this paper is to describe a method for analysing a UML specification in order to find out to which extent it follows clean data abstraction principles. In this sense, a methodology arises which uses an algebraic specification viewpoint to judge on the quality of UML specifications.

1.1 Context of this Work

The work reported here was carried out in the framework of the European “Common Framework Initiative” (CoFI) for the algebraic specification of software and systems. The CoFI initiative [CoFI] is a working group that brings together research institutions from all over Europe. In CoFI, a specification language called “Common Algebraic Specification Language” (CASL) was developed which intends to set a standard unifying the various approaches to algebraic specification and specification of abstract data types [CoFI98]. This language CASL is used in the examples given in this paper.

This is not a traditional paper on formal methods. It is a goal of the CoFI group to closely integrate its work into the world of practical software engineering. As far as specification languages are concerned, this means integration with UML. But this is neither yet another paper on the translation of UML models into a formal specification language. This paper uses the idea of a translation from UML into CASL as a guideline to obtain observations of more general importance.

This paper is structured as follows. In the rest of this introductory section, we discuss the principle of data abstraction and its relationship to high-quality software architecture. Moreover we give a very brief introduction to UML. Section 2 gives a straightforward derivation of an algebraic abstract data type from an isolated class of a UML model. Section 3 discusses how to generalise the approach to more complex systems of classes and points out the problems appearing in such a step. In section 4, we describe in more detail the different philosophies behind encapsulation in UML and algebraic specification. Section 5 outlines a methodology to improve object-oriented (UML) specifications according to insights gained through an analysis in terms of algebraic specifications. Section 6 then draws some conclusions from this work.

* This work was partially supported by the European Union as part of the ESPRIT Working Group CoFI.

1.2 Data Abstraction and Software Architecture

Data abstraction is the principle of specifying a data type together with its characteristic operations in such a way that the internal structure of the data is kept hidden and that the specification has a clear meaning independent of the context in which it is used.

The idea of data abstraction was a very successful contribution from the more theoretically oriented software research community towards industrial practice. The principle itself, and in particular specific abstract data types like queues or stacks nowadays belong to the basic body of knowledge of computer science and are well covered in standard software engineering curricula. Modern class libraries like the Java libraries or STL are organised according to the data abstraction principle. Less successful were the formal languages, which were developed for abstract data type specification. This is partly due to the fact that there is a standard repertory of abstract data types and only little pressure exists in daily development practice to specify completely new data types. The strong theoretic background of abstract data types (see e.g., [LEW96]) gives the approach much power, but also has deterred some potential users.

There is more to the data abstraction principle than just a mechanism for describing standard data types. Abstract data types provide a useful paradigm for the construction of stable software modules [Par72]. The usage of the principle (and again not the specification languages themselves) turned out as practically successful for construction of very large software systems.

Another principle closely related to data abstraction is the idea of layered software architecture. Using abstract data types, it is rather natural to think of software being organised as a structure of layers of increasing abstraction. This principle has been investigated theoretically (e.g., in [W+83]) for abstract data types, and has found its successful practical use in building software systems as hierarchies of software modules. Rather recently, these principles are getting high attention in the context of software architecture research [SG96].

In order to illustrate this situation, Figure 1 shows a typical layered software architecture. The lower layers are implementation dependent and define general mechanisms usable in any kind of software system. Higher layers proceed to more application specific and more implementation independent information. Each layer is based on a clear interface to other layers. In a strict interpretation of the layered architecture, each layer is only allowed to use the layer immediately below it, but also relaxation of the principle is possible. For instance, Figure 1 follows in spirit the architecture advocated, e.g. in [AW98].

Such a layered architecture makes use of the principle of data abstraction. Each layer can be seen as an abstract data type which on one hand exports a clear interface to the upper layer, but on the other hand also imports a clear interface from the underlying layer.

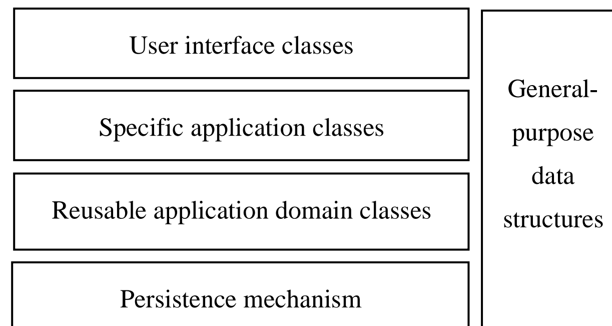


Fig. 1. Layered architecture

The core idea of the layering principle is that there is a hierarchic order among units such that any unit can make use only of a well-defined set of underlying units. Cyclic dependencies of units are avoided. This principle ensures a sound mathematical semantics (as in the theory of abstract data types) and a high level of practical maintainability. Since the principle is well established in the coarse-grained overall software architecture, it would only be logical to apply data abstraction principles also in the intermediate kind of granularity which is the specification of a software unit. This leads to the question how the standard language for this purpose, i.e., the UML class diagram, is related to data abstraction principles.

In this paper we concentrate on the construction of UML object models in analysis and design, and show how such models appear in the light of the data abstraction and layering/hierarchy principles. We will try the effect of a translation into abstract data type specification for selected examples of object-oriented models, in order to draw conclusions for the structure of the UML model. Algebraic specifications appear here indirectly as an auxiliary tool for uncovering semantic properties of UML – we do neither define a full algebraic semantics for UML nor do we advocate the direct use of algebraic specifications in software design. However we claim that some background knowledge from abstract data types helps in finding a methodology for better structured UML models.

1.3 Unified Modeling Language

The Unified Modeling Language (UML) [RJB99] is an industry standard language for specifying software systems. UML has evolved out of an amalgamation of the central ideas of many semi-formal graphical notations for software specification. This language is unique and important for several reasons:

- UML contains more or less all concepts that were discussed in object-oriented modelling in the past. Therefore, it is an ideal vehicle to discuss fundamental issues scientifically.
- UML is very well accepted by the software industry, so using UML improves the practical applicability of any scientific result.
- Compared to other pragmatic modelling notations in Software Engineering, UML is very precisely defined and contains large portions that are similar to a formal specification language.

For reasons of limited space, we do not give a detailed introduction into UML here. However, the discussion below will cover only one diagram type defined in UML, which is the class diagram. So for the reader, some basic knowledge of an arbitrary object-oriented modelling notation containing class diagrams should be sufficient.

2 Understanding an Isolated UML Class as an Abstract Data Type

Most textbooks on object-oriented programming and modelling devote some introductory paragraphs to basic principles like information hiding, function encapsulation and data abstraction. Many people seem to have the understanding that object-oriented modelling is just an adaptation of the ideas of data abstraction to a state-based structure. We start the investigation with simple cases where data abstraction principles are obeyed in object-oriented specifications, but the next section will proceed to examples where this is not the case.

In standard cases, there is a one-to-one correspondence between a UML class and an abstract data type. Such a simple example is the UML class diagram (containing a single class) shown in Figure 2. The example is taken from the Java utility class library (java.util.Vector), with drastic simplifications. The figure shows a UML class box, where the name of the class is in the upmost compartment, the second compartment carries the attributes of the class (none in this case), and the third compartment contains operations (methods) together with their signatures.

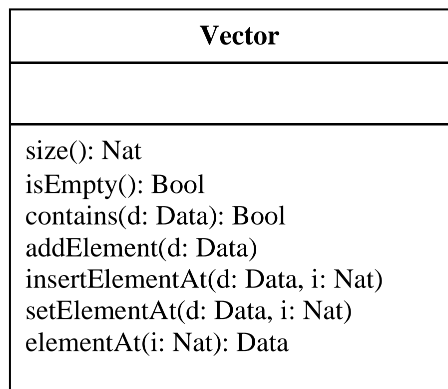


Fig. 2. Utility class Vector

This is certainly an abstract data type. All information that belongs to the implementation of a vector is encapsulated in the class, and the class only depends on primitive classes providing the data types Data, Nat and Bool. It is also typical for such a class to provide no or only very few attributes, since the internal structure is hidden. There is a large number of classes which follow this style, including most of the classical data type libraries. Sometimes, these classes are called *utility classes*.

UML contains a formal specification language to add more precision to class diagrams, the so-called Object Constraint Language OCL [WK98]. Using OCL, the exact meaning of all operations can be described. In the case of a utility class, this is possible without making reference to any user-defined construct outside the class. As an example, see the following OCL specification of operation *addElement*.

```
Vector:: addElement(d: Data)
post self.size() = self.size@pre() +1 and
       self.elementAt(size-1) = d and
       Set{0..self.size@pre()-1}
         forAll(i | self.elementAt(i) = self.elementAt@pre(i))
```

The OCL specification just states a postcondition for the operation *addElement* making reference to the actual object on which the operation is executed (*self*). The three parts of the postcondition state that

- the size is increased by 1,
- the last element in the vector after the update is equal to the parameter given to the operation, and
- all other elements in the vector remain unchanged.

It is a relatively easy task to give a formal algebraic specification for a class like Vector. Below is a corresponding specification in CASL.

```
spec VECTOR =
        BOOL and NAT and DATA
  then
```

```
    sort Vector
    op initVector:      Vector
    op size: Vector     Nat
    op isEmpty: Vector     Bool
    op contains: Vector × Data     Bool
    op addElement: Vector × Data     Vector
    op insertElementAt: Vector × Data × Nat  ?  Vector
    op setElementAt: Vector × Data × Nat  ?  Vector
    op elementAt: Vector × Nat  ?  Data
    vars v: Vector; d: Data; i; Nat
    axioms
      size(addElement(v,d)) = size(v) + 1;
      i == size(v)      elementAt(addElement(v,d),i) = d;
      i < size(v)       elementAt(addElement(v,d),i) = elementAt(v,i);
    ... axioms for other operations omitted here ...
end
```

We do not want to elaborate too much on the details of a translation from UML classes into algebraic specifications, since this has been described already for other specification languages (see e.g. [BGHRS97, EFLR99, KC99, Lano96, SF97, WK96]). However, a few hints may be helpful for the reader.

There are two different identifiers in the algebraic specification derived from the class name. “VECTOR” is the name of a specification unit, whereas “Vector” is the name of a sort in the algebraic specification. In the formalisation of a class, it is natural to define just one sort corresponding to the class name, which will be called “class sort” in the following. The algebraic specification lists the primitive specifications it relies on (BOOL, NAT, DATA). This is the point where the “imported interface” is defined which was mentioned in the above discussion of layered architectures. Then the signature of sorts and operations is defined, followed by a number of axioms specifying the operations precisely.

The operations are specified by argument and result sort, similar as in UML. However, operations are viewed slightly differently in abstract data types and in object-oriented modelling.

An operation in an abstract data type (an *algebraic operation*) explicitly specifies all data sorts on which the semantics of the operation depends. It is the signature of a pure function.

An operation in OO modelling (a *UML operation*) always has an implicit argument of class sort (*self* in OCL). This sort appears as an explicit data sort in the abstract data type.

For those operations that transform the object, a result of class sort is added. If additionally a result is delivered by the operation, it may be necessary to specify two algebraic operations in correspondence to a single UML operation.

Constructor operations are often not shown explicitly in UML classes (although the language admits this). The algebraic operation for the creation of an initial value (here “initVector”) corresponds to a default constructor operation.

Attributes of a UML class box can be understood as operations in the abstract data type using get- and set-operations. Since there are no attributes in the example, this case does not appear in the algebraic specification above.

The semantics of the operations can be specified by standard algebraic specification techniques, as it is shown in the example above. We have included here only those axioms which correspond exactly to the three parts of the OCL specification from further above. In fact, it is even possible to provide a schematic transformation from OCL pre- and postconditions to axioms of algebraic specifications, as has been shown elsewhere [HKH98]. However, in some cases a formulation of the algebraic axioms can be found, which is easier to understand than the OCL specification. This effect can also be observed in the vector example above.

At this point of the discussion, it can be said that the translation between UML class and abstract data type works smoothly. There is a sublanguage of UML class diagrams which corresponds to abstract data types in a straightforward way. This sublanguage can be characterised as follows:

- Classes are isolated. There are no associations to other classes.
- For each class, it is possible to provide a full formal specification of all operations (in OCL or in an algebraic specification language) which makes use only of the attributes and of other operations of the class itself.

In the next section, we discuss in more detail the inclusion of associations within a class diagram.

3 Understanding Associated UML Classes as Abstract Data Types

3.1 An Example for Associated Classes

Unfortunately, the style of translation applied above does not work well for realistic class diagrams, since it is unable to deal with associations. Associations are references to objects, and therefore they violate the principle of locality assumed in abstract data types. The following simple class diagram illustrates this effect.

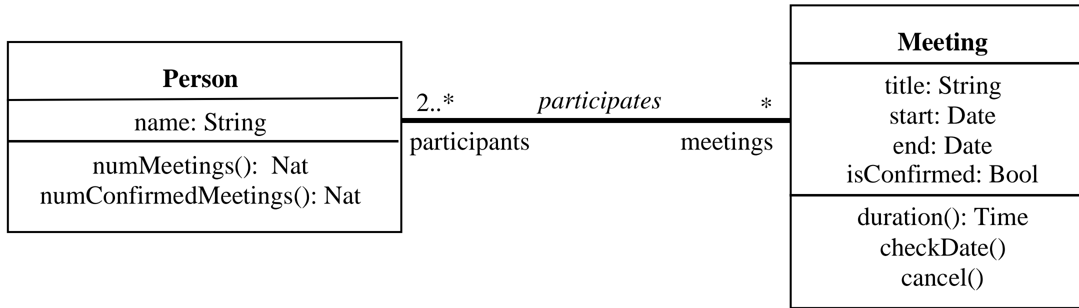


Fig. 3. Two associated classes

The intuitive explanation of this example is that we have meetings in which a number of people may participate. Participants know about the meetings they are involved with, and meetings (meeting objects) "know" their participants. So associations like *participates* are relations between objects of the involved classes.

The meaning of the attributes used in this example is self-explaining, with the possible exception of *isConfirmed*: A meeting may have an unconfirmed status or it may be confirmed which means that none of the participants has a time conflict with another meeting. The operations of the example are explained further below.

3.2 Problems in Translating Associated Classes to Abstract Data Types

An interesting semantic question is whether an association is a local part of an object or whether it belongs to the surrounding environment in which an actual instance of the class (an object) is embedded. A translation into abstract data types shows that the first interpretation is problematic. A naive translation gives the following (faulty!) CASL specification:

```
spec FAULTY_PERSON =
      FAULTY_MEETING and STRING and NAT and SET_OF_MEETINGS
then
      sort Person
      op initPerson:     Person
      op getName: Person  ?  String
      op setName: Person × Name     Person
      op meetings: Person     Set [Meeting]
      op numMeetings: Person     Nat
      op numConfirmedMeetings: Person     Nat
      axioms
        ...
end

spec FAULTY_MEETING =
      FAULTY_PERSON and STRING and BOOL and DATE and SET_OF_PERSONS
then
      sort Meeting
      op initMeeting:     Meeting
      op getTitle: Meeting  ?  String
      op setTitle: Meeting × String     Meeting
      op getStart, getEnd: Meeting  ?  Date
      op setStart, setEnd: Meeting × Date     Meeting
      op getIsConfirmed: Meeting  ?  Bool
      op setIsConfirmed: Meeting × Bool     Meeting
      op participants: Meeting     Set [Person]
      op duration: Meeting     Time
      op checkDate: Meeting     Meeting
      op cancel: Meeting     Meeting
      axioms
        ...
end
```

This is not a proper modular definition of two abstract data types. The main problem is that there is a *cyclic import structure* among the two specifications. In order to make the sort *Meeting* visible, FAULTY_PERSON imports the specification FAULTY_MEETING, and in order to make the sort *Person* visible, FAULTY_MEETING imports FAULTY_PERSON. This is not admitted in algebraic specification languages, for good reasons. There is no clear layered structure here. Instead, it would be adequate to *merge together* the two specifications into one specification defining two sorts. This is the first hint for the fact that we do not deal with two well-encapsulated units here, but with a complex mixture of local and global aspects.

By the way, the problem encountered here is not due to the CASL language or algebraic specifications. This can be seen from the following quotation, which discusses a translation from a differnt object-oriented modelling language (OMT) to a different formal specification language (B): "Cycles A B, B A are not allowed (they would lead to cycles in the machine inclusion relation, which are not allowed). If such cycles are required in the system [...], then the entities must all be placed in a single abstract machine." [Lano96, p. 79]. It should be noted that B is a so-called model-based specification language with support for implicit states, so the essential problem remains if a specification language closer to the object-oriented paradigm is used.

A related problem is that *UML allows non-local operations*. The UML Reference Manual states clearly that "an operation specifies a transformation of the target object (and possibly the state of the rest of the system reachable from the target object) ...." [RJB99, p.369]. This means that UML objects have a "global" view of their environment in contrast to the items described by an abstract data type. In our example, the only truly local operation is *duration* in class *Meeting*. All the other operations use information outside the object:

- The *numMeetings* operation in *Person* needs to know about the set of *Meeting* objects to which the current *Person* object is linked. In the faulty translation from above, this would be a local operation, since there is local access to the set of all object references linked to a *Person* through the *participates* association. However, given the mentioned problems with the translation from above, access to the association information has to be considered as global to the object.
- The *numConfirmedMeetings* operation in *Person*, counting the number of confirmed meetings for a person, is clearly non-local, since it has to read the *isConfirmed* attribute of foreign *Meeting* objects.
- The *checkDate* operation in Meeting is meant to actually check whether the current *Meeting* object has a date conflict. For this purpose, it has to query other *Meeting* objects for their dates, and has to find out which of these *Meeting* objects belong to persons, which participate in the current meeting. This is absolutely non-trivial and non-local. An OCL specification for this operation is given below in Figure 4.
- The *cancel* operation of *Meeting* most clearly is non-local. The associations between *Meeting* and *Person* objects are even modified by this operation.

```
context Meeting :: checkDate()
  post: isConfirmed =
          self.participants ->
                collect(meetings) ->
                      forAll(m | m <> self and m.isConfirmed implies
                              (after(self.end,m.start) or after(m.end,self.start)))
```

Fig. 4. OCL constraint for *checkDate()*

On the algebraic specification level, this poses the question where to locate the corresponding algebraic operations in the structure of specification units. As an example, consider the operation *checkDate*. In order to specify axioms equivalent to the OCL constraint in Figure 4, operations defined in the FAULTY_MEETING specification needs to be visible, for instance to talk about accesses to the *isConfirmed* attribute and navigation over *participants* to Person *objects*. On the other hand, of course, the operations from FAULTY_PERSON have to be visible, for the purpose of navigating from *Person* objects to their meetings. So this analysis shows again that the operation has to be placed at some global level where it is able to access information from both involved classes.

Interestingly, current work on the integration of algebraic specification with the object-oriented paradigm (e.g. [DF98, BHTW99]) essentially ignores the kind of problem described here, and makes the implicit assumption that there is one-to-one correspondence between a class in the object-oriented specification and a sensible specification unit in the algebraic specification.

From the methodological and architectural viewpoint we are taking here, it is an obvious conclusion that *a UML class in general does not encapsulate its local data according to the same data abstraction principles, which are used in algebraic specifications.*

Different conclusions may be drawn at this point:

- The example from above may be regarded as showing bad style of specification, i.e. not properly encapsulating private data. However, it is quite obvious that both classes contain the characteristic information for their objects, and as can be seen from the OCL sample above, this style of specification is perfectly admissible in UML. In the next two sections we will discuss two ways how the insight gained by the analysis of non-locality can be used to better understand or improve the object-oriented (UML) specification.
- The translation from object-oriented specifications to abstract data types may be "lifted" to a global view of the overall object community instead of the compositional approach from above. We give the basic ideas for such a translation in the subsection just below.

3.3 A More Adequate Translation for Associated Classes

In order to show that a translation of general UML class diagrams into an algebraic language like CASL is possible, even in presence of the mentioned problems, we briefly outline the basic idea of such a translation. A more detailed technical definition of the translation is currently being worked out within the CoFI working group.

In this approach, one global abstract data type is constructed out of the overall class diagram. The central concept is here a sort for a global system *state*. The following CASL fragments show the overall approach for the running example:

```
spec STATE =
      PERSON_ID and MEETING_ID and ...
then
      sort State

      op initState:    State

      pred containsPerson: State × PersonId
      op getPersonName: State × PersonId  ?  String
      op setPersonName: State × PersonId × String  ?  State
      op numMeetings: State × PersonId  ?  Nat
      op numConfMeetings: State × PersonId  ?  Nat

      pred containsMeeting: State × MeetingId
      op getMeetingTitle: State × MeetingId  ?  String
      op setMeetingTitle: State × MeetingId × String  ?  State
      ... other attributes omitted ...
      op duration: State × MeetingId  ?  Time
      op checkDate: State × MeetingId  ?  State
      op cancel: State × MeetingId  ?  State

      pred getParticipates: State × PersonId × MeetingId
      op setParticipates: State × PersonId × MeetingId  ?  State

      axioms
        ...
end
```

All class definitions are translated jointly into one big abstract data type. For each class of the UML class diagram, a separate specification of object identifiers is assumed (PERSON_ID, MEETING_ID providing the sorts *PersonId*, *MeetingId*). For each class *C*, there is a *containsC* predicate which asks for a given object identifier whether an object is currently known in the object configuration. Attributes and operations of the class are now translated in a style that adds two arguments to the algebraic operation, which are the global state and the referred object identifier (self). Using global operations and predicates on the state, like *getParticipates* and *setParticipates*, association links (in this case for *participates*) can be set and queried among concrete object identifiers.

This approach is definitely adequate to UML class diagrams. However, from an algebraic point of view, it essentially gives up the data encapsulation principle for the individual classes. There are variants of the scheme which build up the global specification in a more structured and modular way, but due to the problems of the cyclic import mentioned above they are semantically equivalent to the global approach shown above.

4 Encapsulation Philosophies

The most important observation from the preceding analysis is that object-oriented specification and algebraic specification seem to deal differently with the concept of data encapsulation. In this chapter, we will try to point out a difference in philosophy between the classical abstract data type approach and object-oriented modelling which has its roots in a distinction between abstract *specification* and concrete *implementation*[1]. The idea for making this distinction is taken from [CD94].

[1] It is not exactly correct to use the words "analysis" and "design" instead of "specification" and implementation". The specification style meant here is usually already part of the design model, whereas the implementation style is used in a design model close to the end of the design phase, in transition to coding.

In an object-oriented ***specification*** framework like UML/OCL, a distinction between specification and implementation is made, but is not very clearly explained in the standard literature. In an object-oriented specification, only the overall effect of an operation is described, without taking care of details like visibility and access rights. Just for the purpose of specifying the result of an operation, navigation towards foreign objects is absolutely natural, and access to global information is helpful for specification of any local operation. As it is observed in [Civ98], modern object-oriented modelling in languages like UML takes a very system-centric approach and focuses on the collaboration of objects rather than on an object-centric specification of individual classes.

In object-oriented ***implementation***, however, aspects like access rights and visibility have to be observed. Moreover, the model is enhanced by additional information and possibly refined by auxiliary classes and operations.

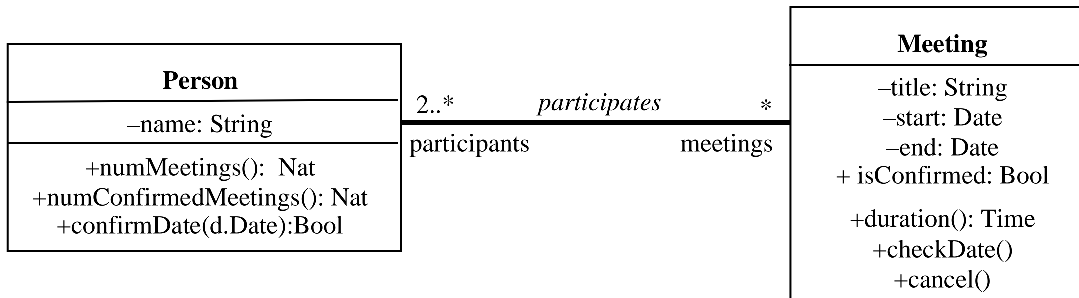
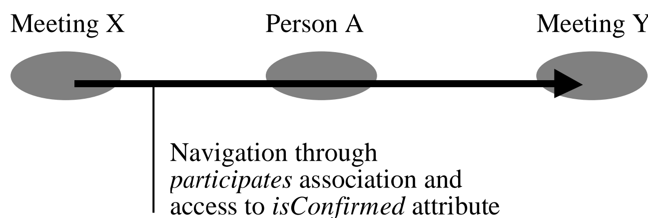


Fig. 5. Associated classes, closer to implementation

In figure 5, the running example is used to explain the effect of a more implementation-oriented view for the object-oriented model. The plus and minus signs indicate visibility of attributes and operations in UML syntax (public and private). In an object-oriented implementation of the example system, it would be necessary to make the attribute *isConfirmed* of *Meeting* publicly visible. In fact, an implementation method for the *numConfirmedMeetings* operation would be allowed to send a message to a *Meeting* object for retrieving its status. Moreover, in order to implement the *checkDate* operation, a new *confirmDate* operation should be added to the Person class. Now the effect of *checkDate* on a *Meeting* object is essentially to send messages to all participating *Person* objects to confirm the date. Figure 6 visualises the differing degrees of encapsulation achieved in the specification and the implementation views.

Specification:



Implementation:

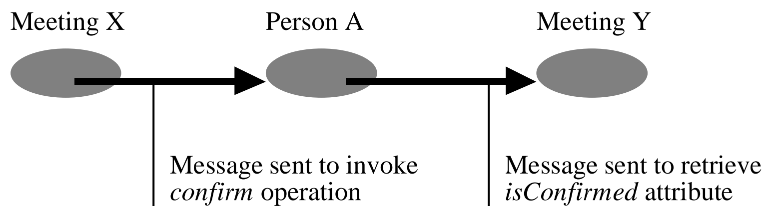


Fig. 6. Encapsulation principles in specification and implementation

In the specification view, the overall, system-oriented effect of the operation *checkDate()* is described by making free use of information from almost anywhere in the current system state. In contrast, in the implementation view with its enhanced specification, each object deals only with its local data, including locally known references to other objects, and sends out requests to neighbouring objects when it needs to know about “foreign” data.

The key point is here that a higher degree of locality and encapsulation is achieved than in the specification, by enriching the operation lists of the classes and by switching to a message-passing semantics. Sending a message to a locally known object and reading the result may be the equivalent to a complex navigation over the object community – however, the global actions, which are caused by sending a message, are invisible to the

invoking object. It seems to be one of the key advantages of the object-oriented approach to modelling and implementation that this smooth transition between a global, system-based view and a local, interaction-oriented view exists.

So the object-oriented approach clearly distinguishes two different encapsulation philosophies for the specification view and the implementation view of operations. In the algebraic specification methodology, however, strict encapsulation principles are already highly recommended on the specification level. These principles are of the same strength as in the implementation view of the object-oriented approach. By enforcing modularity very early, the algebraic approach achieves better-structured specifications but somehow misses the elegant transition between global view and local realisation that is possible in the object-oriented modelling approach.

When seen from the perspective of algebraic specification, this analysis shows some clear deficiencies in current formal specification practice:

- First, an approach is needed for providing a “weak” structuring of high-level specifications, as it can be done in object-oriented modelling. The CASL translation of the UML class diagram using a global state clearly has much less structure than the original diagram. This seems not to be a question of language constructs but of methodology. Current practice of algebraic specifications seems to enforce a relatively strict modularity already rather early in the development life cycle, maybe too early for obtaining an overall view of the system.
- Second, the seamless transition from a structural definition of classes into a specification of a system of active objects is a key factor for the success of object-oriented specifications. In order to deal adequately with object-orientation, algebraic methodologies and languages would be helpful which allow studying the refinement relationship and transformations between a system-oriented, global view and a view of co-operation of locally active objects. This, of course, is closely related with the integration of reactiveness and concurrency into algebraic specification approaches.

5 Layered Architectures

The last section left a rather negative impression about the modelling power of algebraic specifications. Nevertheless, we will show in this section that an analysis of an object-oriented specification in terms of a traditional hierarchical abstract data type system may give fruitful hints towards a better structure of the object-oriented specification.

In fact, the attempt to translate the object-oriented specification from Figure 3 into the “faulty” algebraic specifications included a thorough analysis of the locality of operations. In order to obtain a good system structure, it is the next step to separate the operations into groups that can be specified locally. For the example, this proceeds as follows:

Class *Person*:
    Attribute *name*:
        Can be specified locally, as in FAULTY_PERSON
    Operation *numMeetings*:
        Requires information about *participates* association
    Operation *numConfMeetings*:
        Requires information about *participates* association and *isConfirmed* attribute.
Class *Meeting*:
    Attributes *title*, *start*, *end*:
        Can be specified locally
    Attribute *isConfirmed*:
        Can be specified locally, but needs to be made available to *Person* operations
    Operation *duration*:
        Can be specified locally
    Operation *checkDate*:
        Requires extensive information about *participates* association.
    Operation *cancel*:
        Requires access to *participates* association

Formal specifications are used here as a tool to derive the degree of locality. In the following examples we use the FAULTY_PERSON and FAULTY_MEETING specifications from above for this purpose. For instance, the only axiom required for specifying the operation *duration* is:

```
duration(m) = timeDifference(getEnd(m),getStart(m));
```

This axiom needs only the locally declared operations to access the attributes. For this axiom, there is no need to import a Person-related specification into the local specification. So it can be called truly local. For the operation *numMeetings*, however, we write the axiom:

```
numMeetings(m) = size(meetings(m));
```

In this case, we need the *meetings* operation, which makes use of the *Person* sort and requires a Person-related specification to be imported. However, nothing more than the sort *Person* is used, so it is not the full specification of the *Meeting* class we need but just the association *participates* towards *Meeting* objects.

Based on the results of the analysis sketched above, a regrouping of operations can be achieved, which leads to the following result:

Local to *Person*:
  Attribute *name*:
Local to *Person+participates*:
  Operation *numMeetings*
Local to *Meeting*:
  Attributes *title*, *start*, *end*:
  Operation *duration*
Local to *Person+participates+Meeting*:
  Attribute *isConfirmed*
  Operation *checkDate*
  Operation *numConfMeetings*
  Operation *cancel*

These four levels correspond to a classical layered architecture, since each level makes use only of the underlying levels. In algebraic specification terms, this means each of the four levels can be written as a separate specification (in the style of the specification introduced in section 3.3). As a starting point, we define a "zero" level, which just introduces the *State* sort. This leads to a specification of the following shape:

```
%% Level 0
spec STATE =
      sort State
      op initState:    State
end

%% Level 1: Person
spec PERSON =
      STATE and PERSON_ID
then
      pred containsPerson: State × PersonId
      op getPersonName: State × PersonId  ?  String
      op setPersonName: State × PersonId × String  ?  State
      axioms  ...
end

%% Level 2: Person+participates
spec PERSON_PART =
      PERSON and MEETING_ID
then
      pred getParticipates: State × PersonId × MeetingId
      op setParticipates: State × PersonId × MeetingId  ?  State
      op numMeetings: State × PersonId  ?  Nat
      axioms  ...
end

%% Level 3: Meeting
spec MEETING =
      STATE and MEETING_ID
then
      pred containsMeeting: State × MeetingId
      op getMeetingTitle: State × MeetingId  ?  String
      op setMeetingTitle: State × MeetingId × String  ?  State
      op getMeetingStart: State × MeetingId  ?  String
      op setMeetingStart: State × MeetingId × String  ?  State
      op getMeetingEnd: State × MeetingId  ?  String
      op setMeetingEnd: State × MeetingId × String  ?  State
      op duration: State × MeetingId  ?  Time
      axioms  ...
```

```
end

%% Level 4: Person+participates+Meeting
spec PARTICIPATION =
      PERSON_PART and MEETING
then
      op getIsConfirmed: State × MeetingId  ?  String
      op setIsConfirmed: State × MeetingId × String  ?  State
      op numConfMeetings: State × PersonId  ?  Nat
      op checkDate: State × MeetingId  ?  State
      op cancel: State × MeetingId  ?  State
      axioms ...
end
```

This is a perfectly legal structured algebraic specification without any circular import problem. When looking at the import structure in more detail, it can be observed that the structure is a bit more complex than just four (five including the basis) consecutive layers. The actual architecture is shown in Figure 7.

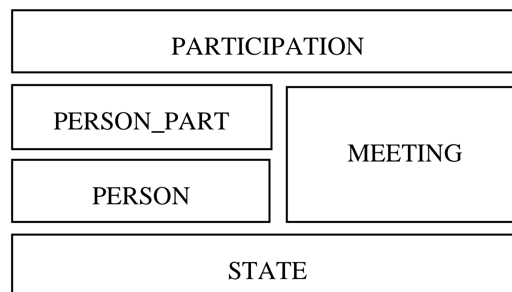


Fig. 7. Layered structure derived from formal analysis

This way, the attempt to translate into a formal specification notation has uncovered very important structural properties of the object-oriented specification. It should be noted that a similar analysis could be made based on the OCL language ([CKMWW99] makes first steps in this direction). However, OCL is missing any structuring concepts, so the result of the analysis is much more difficult to describe than in, e.g., CASL.

In object-oriented modelling, the layered structure can be depicted also within a revised UML class diagram. The main features of UML used in such a restructuring are generalisation (inheritance) and, in some cases, restriction of associations from bi-directional to unidirectional interpretation. Figure 8 gives a UML diagram refined according to the results of the formal analysis. The hollow arrow symbol depicts inheritance in UML notation. So, for instance, any object of class PERSON_PART has the *name* attribute of class *Person*, but in addition it may have an association to a *Meeting* object.

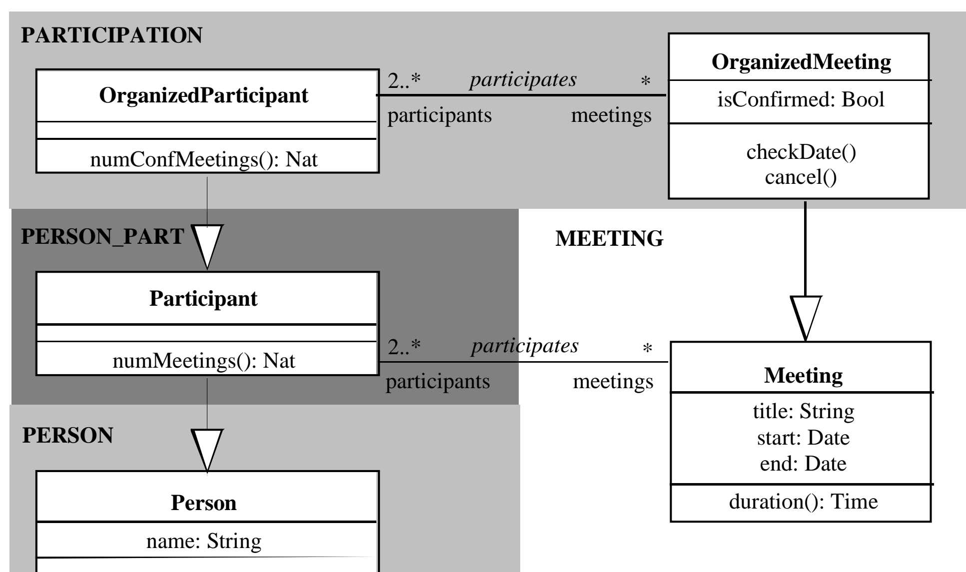


Fig. 8. Revised, layered class diagram

The object-oriented model depicted in Figure 8 clearly corresponds to the results of the formal analysis described in the preceding chapter. Nevertheless, it is perfectly reasonable from the common-sense point of view. There is a class

of persons which is independent of the concept of meetings. This highly improves reusability of the *Person* class. Also the level of a “person who goes to a meeting” (PERSON_PART) makes practical sense. Finally, there are sophisticated mechanisms of meeting organization which have impact to persons and meetings, and those are encapsulated in the highest layer. Simpler architectures, merging some of the layers, may be more adequate in practical situations, but are not in contradiction to the general results of the analysis.

Of course, a very experienced modeler is able to arrive at the same conclusion easily and probably much faster. However, the question in such a case always is what the exact knowledge was which led to the actual conclusion. It is a nice feature of the approach presented here that a purely formal semantic analysis technique gives hints for restructuring a class model.

6 Conclusion and Outlook

6.1 Related Work

There is plenty of literature on mapping object-oriented models to formal specification languages (e.g. [BGHRS97, DF98, EFLR99, GR98, HKH98, Hus97, KC99, L96, SF97, WK96]). However, in most of this literature, the problem discussed in detail in this paper, i.e. the interrelationship between global/association-based and local views of classes is not in the focus of interest. Therefore, most of these approaches also are not very adequate for the application to large, realistic cases. One exception from this rule if the work of [KC99], where the built-in mechanisms of the chosen language Object-Z exactly provide the dual local/global view which was discussed above.

The work presented here is on the borderline between adjacent and somehow competing areas of computer science. For instance, some techniques developed purely in the area of object-oriented programming methodology (e.g. [Civ98] are very similar to the techniques developed here. However, the approach sketched above contains the promise for a much higher degree of formality, i.e. supportability in tools, than purely informal concepts.

6.2 Main Insights

Using an object-oriented analysis and design method does not automatically improve the quality of software – it is a craft if not an art to write good UML specifications and designs. Structuring principles according to data abstraction will provide help for adequate design of class structures in many (but of course not all) cases. Thinking about the ADT “meaning” of a UML class helps in understanding the level of modularity which has been achieved in the OO model. In this paper, we have indicated some basic ideas for an automated analysis that helps in restructuring a UML class model to achieve higher cohesion, lower coupling and therefore higher stability of the structures.

The results described here can be taken as a starting point for several kinds of further investigations. There is a “UML/OCL” branch for further work, since it is interesting to further develop the idea of a “semantic context” for an OCL constraint (based e.g on the work of [CKMWW99]) and to provide prototypical tools for structure analysis of object-oriented specification. In particular, the aspect of specification structuring by enrichment (algebraic style) vs. inheritance (object-oriented style) seems to be worth a more thorough investigation.

On the other hand, there are several branches of further work related to algebraic specification languages. The close integration between an algebraic specification language and UML, which was sketched above, may form the basis for extensions or experimental alternatives to OCL. This can be motivated by the fact that for the specification of utility classes like generally used data types, an algebraic style may lead to more readable specifications than using OCL, so one could think of a combination of algebraic specification, UML and OCL. This aspect will be further followed within the CoFI initiative.

Another insight from this work, which will be further followed with the CoFI group is the question to extend the algebraic semantics to the aspects of active objects and object communication [RACH99]. As it was explained above, this may lead to a similarly elegant integration of global, system-oriented and local, interaction-oriented aspects of a system specification as it is available in object-oriented languages. The long-term perspective of this work is to build a bridge between object-oriented UML specifications and the powerful animation and verification tools that are available for algebraic specifications and term rewriting systems.

References

[AW98] S. Ambler, R. Wiener, Building object applications that work, Cambridge University Press 1998.

[BGHRS97] R. Breu, R. Grosu, F. Huber, B. Rumpe, W. Schwerin, Towards a precise semantics for object-oriented modeling techniques. Proc. ECOOP 97 Workshop reader, LNCS 1357, Springer 1997.

[BHTW99] M. Bidoit, R. Hennicker, F. Tort, M. Wirsing, Correct realizations of interface Constraints with OCL, in: R. France, B. Rumpe (eds.), Proceedings UML'99 Conference, Lecture Notes in Computer Science Vol. 1723, Springer 1999.

[CD94] S. Cook, J. Daniels, Designing object systems – Object-oriented modelling with Syntropy, Prentice-Hall 1994.

[CKMWW99] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, A. Wills, Defining the context of OCL expressions, in: R. France, B. Rumpe (eds.), Proceedings UML'99 Conference, Lecture Notes in Computer Science Vol. 1723, Springer 1999.

[Civ98] F. Civello, Rooted Class Diagrams: a notation for context.independent models, *Journal of Object-Oriented Programming*, Vol 11, No 2, May 1998.

[CoFI] http://www.brics.dk/Projects/CoFI/

[CoFI99] CoFI Task Group on Language Design. CASL-The CoFI algebraic specification language - Summary . http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary. (1999).

[DF98] R. Diaconescu, K. Futatsugi: CafeOBJ Report. World Scientific 1998.

[EFLR99] A. Evans, R. France, K. Lano, B. Rumpe, Developing UML as a formal modeling notation, in: Proc. UML 98 Conference, LNCS Vol. 1618, 1999.

[GR98] M. Gogolla, M. Richters, On combining semi-formal and formal object specification techniques, Proc. WADT 97, LNCS Vol. 1376, Springer 1997.

[HKH98] A. Hamie, S. Kent, J. Howse, A semantics for the OCL, Technical Report University of Brighton, 1998.

[Hus97] H. Hussmann, Formal foundations for Software Engineering methods, Springer 1997 (Lecture Notes in Computer Science 1322).

[KC99] S. Kim, D. Carrington, Formalizing the UML class diagram using Object-Z, in: R. France, B. Rumpe (eds.), Proceedings UML'99 Conference, Lecture Notes in Computer Science Vol. 1723, Springer 1999.

[Lano96] K. Lano, The B Language and Method, Springer 1996.

[LEW96] J. Loeckx, H.-D. Ehrich, M. Wolf, Specification of abstract data types, Wiley-Teubner 1996.

[Par72] D. Parnas, A technique for software module specification with examples. Communications of the ACM 15,5 (1972), 330-336.

[RACH99] G. Reggio, E. Astesiano, C. Choppy, H. Hussmann, Analysing UML active Classes and associated state machines – A lightweight formal approach, submitted for publication.

[RJB99] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language reference manual, Addison-Wesley 1999.

[SF97] M. Shroff, R. France, Towards a formalization of UML class structures in Z, Proc. COMPSAC 97, IEEE, 1997.

[SG96] M. Shaw, S. Garlan, Software Architecture – Perspectives of an emerging discipline, Prentice-Hall 1996.

[WK96] M. Wirsing, A. Knapp, A formal approach to object-oriented software enginering, Proc. RWLW96, Electronic Notes in Theoretical Computer Science, Vol. 4, 1996.

[WK98] J. Warmer, A. Kleppe, The Object Constraint Language, Addison-Wesley 1998.

[W+83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy, On hierarchies of abstract data types, Acta Informatica 20 (1983), 1-33