Department of Computing Science and Mathematics
University of Stirling

Complex Objects: Aggregates

Ana M. D. Moreira and Robert G. Clark

Technical Report CSM-123

August 1994

Department of Computing Science and Mathematics
University of Stirling

Complex Objects: Aggregates

Ana M. D. Moreira and Robert G. Clark

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email amm@cs.stir.ac.uk
rgc@cs.stir.ac.uk

Technical Report CSM-123

August 1994

Abstract

Aggregation is a kind of abstraction which allows a more complex object, the aggregate, to be formed from simpler objects, the components. Although there is not yet a standard definition of aggregation, the two main cases are aggregates with hidden components and aggregates with shared components. The more interesting case is the former, in which the aggregate encapsulates its components, making them invisible to the other objects in the model.

The ROOA (Rigorous Object-Oriented Analysis) method, developed to integrate LOTOS within object-oriented analysis, models both kinds of aggregation.

1 Introduction

We have developed the Rigorous Object-Oriented Analysis (ROOA) method which integrates LOTOS with object-oriented analysis [10, 11, 12]. In this document we describe *aggregation* within the context of ROOA and show how it can be represented in LOTOS. The description is therefore within the context of object-oriented analysis although most of the discussion is also applicable to object-oriented design.

In entity-relationship models, the term aggregation is used to describe the relationship between an entity and its attributes. Each attribute is a component of an entity, and the entity is an aggregate of its attributes. We are interested in a broader definition of aggregation which describes the relationship between objects and allows a more complex object to be formed from the combination of simpler objects. The complex object is called the *aggregate* object and the simpler objects are called *object components*, or just *components*. Aggregates are described by *part_of*, *whole_part*, *component_of*, or *consists_of* relationships.

We have a broader view of aggregation than some other authors. An important characteristic of our aggregates is that they may add behaviour to the behaviour defined in their components. Moreover, components within the aggregate may communicate between each other without the aggregate's 'knowledge'. The services defined in a component may or may not be on offer by the aggregate. We classify aggregation according to whether or not the components are visible to the other objects in the system and also whether or not the number of components can vary in time.

We make a clear distinction between aggregates and *subsystems*. While an aggregate is an object, a subsystem is only a group of objects, without individual identity.

Although, in the past few years, researchers have paid special attention to aggregation, there is no standard definition of what an aggregate is. This paper reviews the more common views of aggregation and presents our view, proposing a set of properties that aggregates should satisfy. Finally, it shows how ROOA uses LOTOS to model aggregates.

2 The Role of Aggregation

Aggregation is a relationship between several objects which allows a more complex object, the *aggregate*, to be built from a combination of simpler objects, the *components*. Both aggregates and components are objects; they have an identity, they offer services according to a certain behaviour and they have state information which records the results of their services. The behaviour and state information of the aggregate is given by a combination of the behaviour and state information of its components and by extra data and functionality.

Aggregation with hidden components is a kind of abstraction. There are other kinds, generalization being an example. Abstraction is the suppression of detail about an object, except for that relevant to the immediate purpose. Other kinds of abstraction can be applied during implementation where a concept is described by parts, but none of these parts is a true object. Implementation abstraction hides implementation detail from the user. "Real-world" abstraction, such as aggregation and generalization, is useful when thinking about the real world. It defines a more abstract (higher level) object which is useful for our understanding of the problem. This sort of abstraction helps in controlling the size and complexity of large systems during development and, if a system is developed using levels of abstraction, the resulting product is less difficult to understand.

It is important to distinguish between the role of an aggregate and and that of a grouping of objects, such as *subsystems*. We believe that an aggregate must be a representation of an entity from the real world and it may have information of its own, for example the number of components. In the more interesting case, the aggregate manages the interaction between its components and the rest of the objects in the model. Whereas subsystems may be used during analysis, and may have no implementation consequences, aggregates are objects which have first class status in the system and which usually appear in the final implementation.

A subsystem is not an object, it is a grouping of logically related objects. The behaviour of a subsystem is given by the behaviour of its components. Each component communicates directly with objects outside the scope of the subsystem, as if the subsystem did not exist. Wirfs-Brock *et al.*, for example, describe a subsystem as an analysis construct which does not survive in the implementation [16, p30]:

> A *subsystem* is a set of classes (and possibly other subsystems) collaborating to fulfill a set of responsibilities. Although subsystems do not exist as the software executes, they are useful conceptual entities.

These subsystems are groupings of objects useful for understanding a problem, but they may or may not describe a concept from the real world. Aggregates, on the other hand, describe complex concepts from the real world and so they are named objects.

The role played by aggregation varies from author to author. Rumbaugh *et al.* define it as a stronger form of association (conceptual relationship) [14]; Smith and Smith stress its importance as a mechanism to introduce abstraction [15]; and Hartmann *et al.* [5] use aggregation as a structuring mechanism and define its formal semantics, showing how it is supported by the specification language TROLL [8].

We see aggregation as a mechanism for structuring a large system into different levels of abstraction, in which each higher level object is described in terms of simpler objects. As we will see, it uses abstraction, information hiding and encapsulation as basic techniques.

3 Combination of Objects: Another View

Although aggregation seems a simple concept, a standard definition does not exist (see e.g. [1, 2, 5, 13]).

As we are interested in modelling aggregation formally, this section discusses the view supported by Hartmann *et al.* who define a formal semantics for composite objects [5]. According to them, the combination of objects to form a more complex object (such as an aggregate) is modelled by *structure-preserving* mappings between objects, also called *object* *morphisms.* A special case of object morphism is the *object embedding morphism* which describes a complex object ob_1 encapsulating an object ob_2. The set of life cycles of the encapsulated object ob_2 must be preserved and events of ob_1 must use events of ob_2 to modify ob_2's state. This latter requirement is captured by the concept of *calling.* If an event e_1, in ob_1, calls an event e_2, in ob_2, then whenever e_1 occurs, e_2 also occurs.

Hartmann *et al.* state that an object embedding morphism between objects ob_1 and ob_2, assuming that ob_2 is embedded in ob_1, has to satisfy the two following conditions:

1. The events of ob_2 are included in the set of events of ob_1. For the life cycle of the (composite) object ob_1, if we constrain a life cycle to the events of the embedded (or part) object, we have to obtain a valid life cycle of the embedded object.

2. The attributes of ob_2 are included in the attributes of ob_1. For observations of ob_1 projected to the attributes of the part object, we must obtain the same observation as for applying the observation mapping of part object ob_2 to a life cycle of ob_1 restricted to the events of ob_2.

An aggregate object that satisfies these two rules is the coproduct of its components. Such an object does not add any behaviour to the behaviour already defined in its components. Most OOA (Object-Oriented Analysis) methods argue that the interesting situation appears when the aggregate has properties of its own together with the properties of its components [3, 4, 14].

Imposing the condition that every event defined in an object component has also to be defined in the aggregate forbids a component to have services hidden from the aggregate. Such hidden services would allow, for example, communication between components without the aggregate's intervention. Hence, the two rules forbid the communication between object components defined at the same level of the hierarchy and in the scope of the aggregate.

We want our aggregates to have extra functionality which is unknown to their components. Moreover, we want to allow communication between object components without the aggregate's knowledge. Therefore we permit services defined in a component, and which are only needed by another component, not to be on offer by the aggregate. For these reasons our aggregates are different from those of Hartmann *et al.*

4 Transitivity

As a component is an object, it may happen that this object is itself a complex object, composed of other objects. The object at the top level of the hierarchy need not know that one or more of its components are aggregates. This leads to the issue of transitivity. While some authors [4] explicitly say that aggregation is not transitive, other authors [14] define transitivity as one of the most important properties of aggregates.

Transitivity can be discussed according to the semantics of the aggregation relationship. Let us suppose that aggregate ob_1 has the component ob_2 which has a component ob_3. It is the case that if ob_2 is part of the internal structure of ob_1 and ob_3 is part of the internal structure of ob_2, then ob_3 is part of the internal structure of ob_1. However, transitivity is lost when aggregation relationships with different semantics are involved. For example, I am part of a research group and my arm is part of me, but my arm is not part of my research group.

Message connections, for example, are not transitive. If ob_1 has a message connection with ob_2 and ob_2 has a message connection with ob_3, this does not mean that ob_1 has a message connection with ob_3. If this was not so, information hiding and encapsulation would not be available. Nevertheless, indirect communication is possible. When the top level object, object ob_1, requests a service from one of its components, object ob_2, it may well be that ob_2 delegates part of the service to one of its components, object ob_3. However, ob_1 does not need to know the ob_2-services which correspond to calls of ob_3-services. We can say that, in general, ob_1-services call ob_2-services which call ob_3-services. This is discussed by Rumbaugh *et al.* [14] as *propagation of operations* and is what Hartmann *et al.* [5] calls *event-calling.*

5 Classifying Aggregation

We describe an aggregate in terms of its components. We allow a component to be hidden (non-shared) from other objects or shared by other objects. We use the term *aggregation* to refer to the relationship between the aggregate and its components. As a conceptual relationship, aggregation has cardinality. We show cardinality at the end points of the relationship with an amount (k) or a range (n, m) [1].

A hidden component is not visible to other objects in the system and so it can only communicate with the rest of the system through the aggregate, although it can interact with other components in the same aggregate. An aggregate defines the scope of its hidden components and encapsulates each of them.

A shared component can be accessed by other objects in the system. For each shared component, an aggregate has an attribute holding the object identifier of that component.

A component encapsulates its own state and behaviour, in the sense that its state may only be changed using the services defined in its interface. The interaction between an aggregate and its components and among components is via communication [11].

So far, we have been talking about aggregates and components as being objects. We will from here on talk about aggregate classes, component classes, aggregate objects and component objects. However, in situations where the meaning is understood by the context, we may use only the terms "aggregate" and "component".

In the object model of OOA methods the aggregate is represented by the class template at the top level of the hierarchy and the components are the class templates at the bottom (see Figure 1).

For our work, we allow aggregates with either a static or dynamic number of components, but we restrict the number of component classes to be constant. The structure of an aggregate with a constant number of components is defined at requirements-specification time and its composition never changes, while the structure of an aggregate with a variable number of components can have its composition changed during its life time.

Aggregation with a static number of hidden components is called *disjoint* composition by Hartmann *et al.* [5] and *ensemble* by Champeaux *et al.* [4]. Hartmann and his colleagues also distinguish aggregation with a static number of shared components (they call it *static non-disjoint* composition) and aggregation with a dynamic number of shared components (they call it *dynamic non-disjoint* composition).

Some authors treat other aspects of aggregation. For example, Odell classifies aggregates according to six kinds [13]. Most of these kinds of composite objects do not seem to us to be useful in describing software systems.

[1] We omit cardinality when it is not important for the problem being discussed.

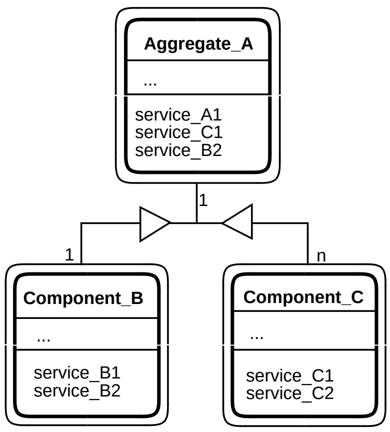

Figure 1: Aggregation

5.1 Aggregation: Hidden Components

A hidden component is defined internally to its aggregate, and so it is hidden to the other objects in the system (as defined in [4, 5, 14]). Furthermore, this object exists only while the aggregate exists. A hidden component may only interact with other components in the same aggregate or with the aggregate. The aggregate manages any interaction between such components and the rest of the model. All the services defined within hidden components which are to be offered to the outside are on offer by the aggregate. When these services are required from the aggregate by another object, the aggregate routes the message to one of its components, receives the answer, if any, and then returns the result. Figure 2 shows a simple object model composed of an aggregate with two hidden components and a client. We have used a modified version of the Coad and Yourdon notation. Each box in the object model represents a class template.

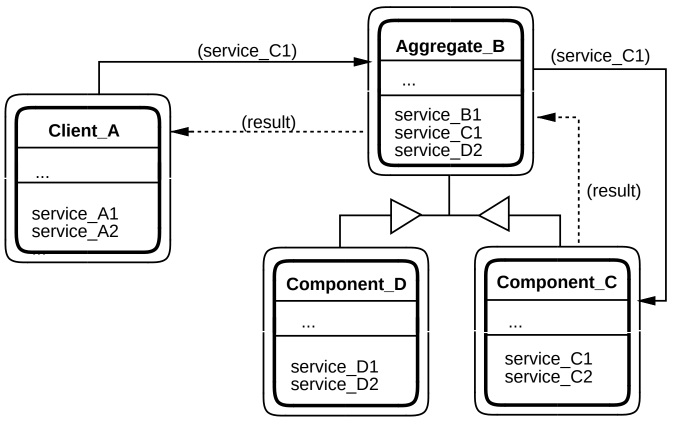

Figure 2: Aggregation with two hidden components

There is a problem in representing aggregation with hidden components. The notation used by object-oriented analysis methods, such as [3, 7, 14] is confusing, since it represents two different levels of abstraction in the same diagram. The aggregate is represented by the top box in the diagram and the components are represented by the lower boxes. However, the aggregate is really composed of everything and the lower boxes should be represented inside the top box.

In Structured Analysis, when drawing Data Flow Diagrams (DFDs), for example, we use a diagram for each level of abstraction: a DFD at level *n+1* replaces a process in the DFD at level *n*. This approach is not used to build object models. In a CASE tool, such as ObjecTool[2], subjects (or modules, or subsystems) can be expanded to show their object components, even if the resulting final diagram has a flat structure. However, an aggregate cannot be simply replaced by its components, because it is not just the collection of its components, as a subsystem is; it has extra functionality. If aggregates are useful to structure a system into levels of abstraction, we should be able to use them to build object models accordingly. Figure 3 represents this idea.

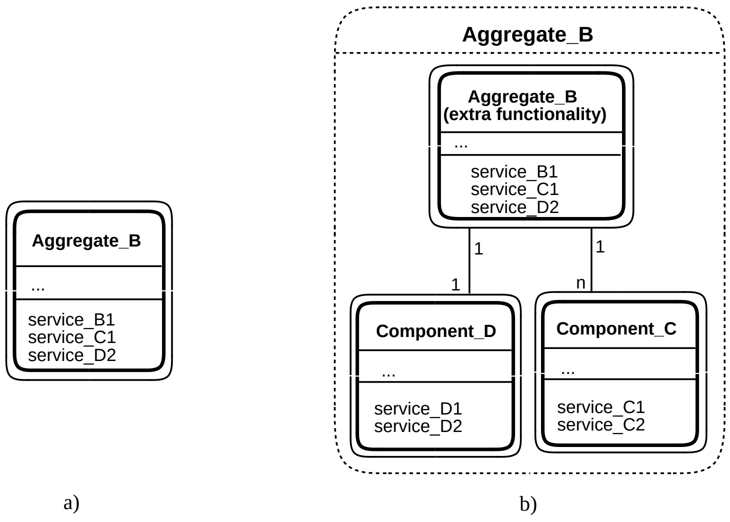

Figure 3: a) Higher level of abstraction: aggregate; b) Lower level of abstraction: the components and the aggregate's extra functionality

We want to use existing object-oriented CASE tools, and so we accept the standard OOA terminology. We propose an alternative interpretation of the diagram, considering, at the lower level of abstraction, the top box of the diagram to be the interface of the aggregate and the aggregate, i.e. the whole structure, as the dotted box shown in Figure 4.

At one level of abstraction, *Aggregate B* represents the complete structure. At a lower level of abstraction we have *Component C*, *Component D* and the extra functionality the aggregate offers (represented by *Aggregate B* which is now regarded as the interface to the components). We choose to name the complete aggregate with the same name as the box at the top level of the aggregation, since they represent the same concept.

If the number of instances of each component is static, the internal structure of the aggregate never changes. If the number of object components is dynamic, the aggregate may create new objects or even remove some of the existing ones, changing its internal structure. Notice however, that in both situations, the interface of the aggregate never changes and the components only exist while the aggregate exists. The creation of new component objects is the exclusive decision of the aggregate.

[2]ObjecTool is a trademark of Object International, Inc.

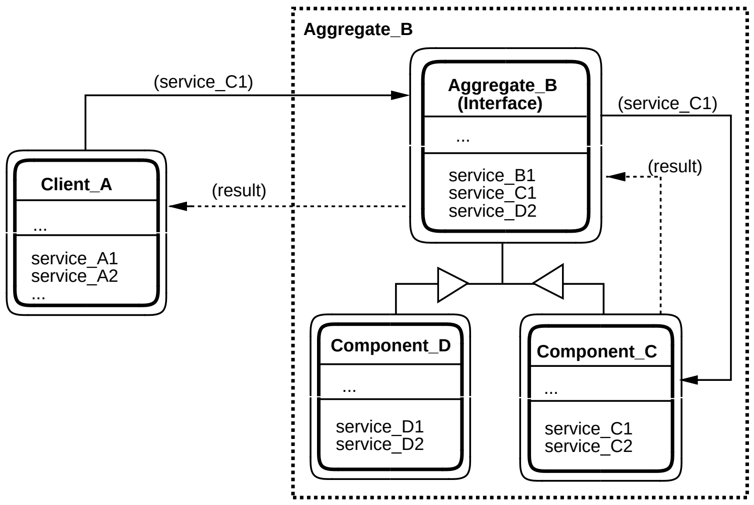

Figure 4: Aggregation with hidden components marked as a single object

5.2 Aggregation: Shared Components

A shared component has a life of its own, that is, it can exist independently of the aggregate. While a non-shared component is encapsulated in the aggregate and hidden from the rest of the model, a shared component is visible outside the scope of the aggregate and therefore it can communicate directly with other objects. The aggregate knows its shared components by having a specific attribute for each one. This attribute holds the object identifier of the component. Figure 5 shows the example given in Figure 2, but now *Component C* is visible outside the aggregate.

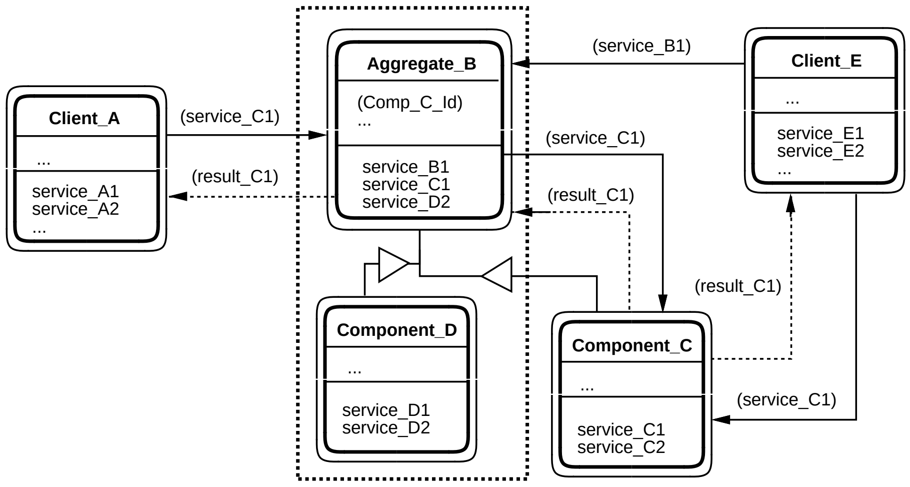

Figure 5: Static aggregation with one shared component and one hidden component

The top box represents the whole aggregate and each lower box its components. As *Component C* is shared, it is outside the dotted box. The services offered by the shared component can also be offered by the aggregate. It is a client's decision to choose *Aggregate B* or *Component C* for communication. When an object, outside the scope of the aggregate, communicates with the aggregate, it may or may not give the identifier of the shared object involved.

There are situations which may be relevant in understanding certain problems, but which are not shown by an object model. When we talk about a shared component, we do not specify in which terms this component is shared. For example, it may be important to be able to determine whether or not a given instance of a component class is shared, or if the sharing concerns different instances.

Let us consider two examples. In the first one, we define an aggregate `Car` with a component `Engine`. The engine class is shared by another aggregate, `Plane`. It seems reasonable to assume that one instance of `Engine` may not be shared by a car and a plane, at the same time. Therefore, in Figure 6 we have sharing of a concept, but not a sharing of objects, even if the concept of relationship is defined between instances and not between classes.

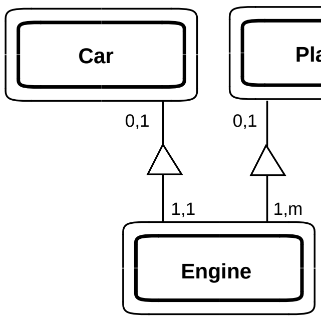

Figure 6: Sharing the class template, but not the objects

For the relationship between *Engine* and *Car*, *Engine* has optional cardinality, which means that an engine may or may not have a relation with an object in *Car* (similarly for the relationship between *Engine* and *Plane*). In such a situation, the object model does not distinguish between cases where an object component can belong to two aggregate objects at the same time, belong to only one of them, or to none of them.

The case in Figure 6 can be seen as a mixture of sharing and hiding. The component class is shared, but the component object is hidden. A component class being shared means that we can use the definition of the component to build different aggregates. (If the component class was not shared it would not be visible to other aggregates.) A component object being hidden, means that the object has to be instantiated within the aggregate and no other objects know about it.

Now, consider the aggregates `Family` and `Research Group` sharing the same component `Person`. In this case, an instance of person can be simultaneously shared by an instance of family and by an instance of research group. As we can see in Figure 7, we cannot show the difference between this situation and the situation depicted in Figure 6 in an object model.

We can have static aggregation, when the number of components is constant, and dynamic aggregation, when the number of components is variable. Both cases are dealt with in a similar way to aggregation with hidden components, except that now, the creation of an object component may be initiated either by the aggregate or by one of the other objects.

Hartmann *et al.* allow the creation of a dynamic aggregate with non-existing object components [5]. Such components, at the moment the aggregate is defined, have an empty life cycle, but their identifiers have to be pre-determined.

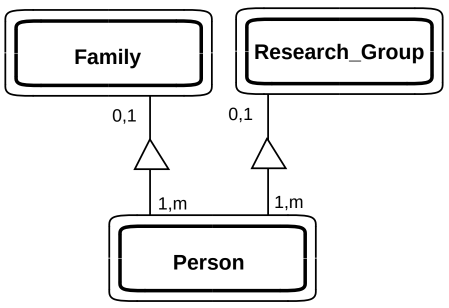

Figure 7: Sharing the class template and the objects

According to some authors, only aggregation with hiding represents a useful mechanism [7, 14]. Rumbaugh *et al.*, for example, argue that aggregation with sharing should be treated as an ordinary conceptual relationship [14]. We believe that there are advantages in showing such a relationship as an aggregation as it can improve our understanding of a system and may give directions for reusability. (We justify this view in Sections 8.2 and 8.4.) While a conceptual relationship is likely to change when it is put in a different context, an aggregation may not change.

With respect to deletion, a shared component can only be removed when no other object in the system has access to it.

5.3 Catalog Aggregation

Some authors classify aggregation into *physical* and *catalog*, according to its cardinality [1]. In physical aggregation, each component object is part of at most one aggregate object of a given class, while, in catalog aggregation, each component object may be part of many aggregate objects of the same class. That is, in a physical aggregation, each relationship from each component to the aggregate has multiplicity *one*, while in catalog aggregation, each relationship from each component to the aggregate has the multiplicity *many.* This is shown in Figure 8.

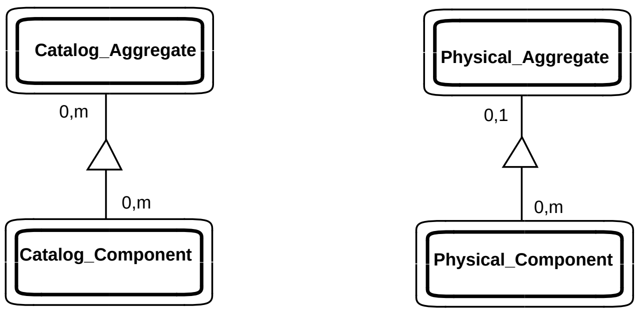

Figure 8: Catalog aggregation and physical aggregation

Catalog aggregation describes sharing between instances of the same class: two different instances of a given aggregate class share the same component object. In Figure 9 the object `Edward` is a member of the `Neural Networks` and `Artificial Intelligence` research groups.

When we allow catalog aggregation we lose information hiding, since an object component

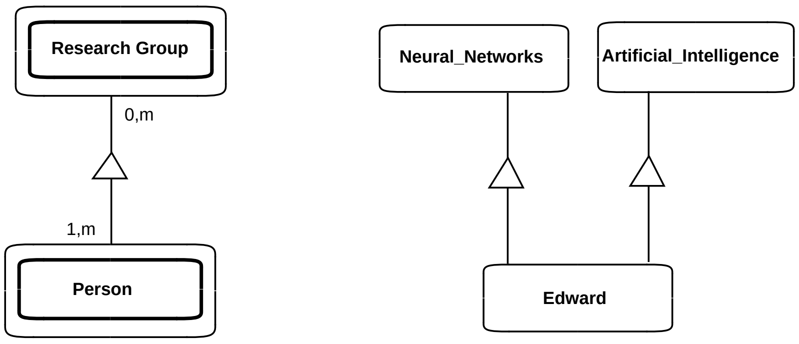

Figure 9: One person belongs to two research groups

has to be known by two aggregate objects. Therefore, we treat this kind of aggregation as aggregation with shared component classes. Catalog aggregation is helpful when designing problems where components of the same type are interchangeable.

When discussing the design of relational data bases, Smith forbids catalog aggregation by imposing the following rule [15]: for a given aggregate class A with the component classes C_1, ..., C_n, two distinct real-world instances of A must not determine the same instances of C_i. That is, a given member of C_i must not be shared by two instances of A.

Kim *et al.*, when defining *dependent objects*, are implicitly defining physical aggregation [9]. Some authors classify the complex object of a physical aggregation into *aggregate*, if it has a $1:1$ relationship with each component, and *collection*, if it has a $1:N$ relationship with each component [2]. A collection is useful to define homogeneous sets of objects, for example a `Football Team` is a collection of objects of the class `Athlete`. However, we do not consider this a special kind of aggregation and we use the term aggregate for both cases.

6 Properties of Aggregation

There are many different views of aggregation and in the previous sections we have discussed some of them. Authors seem not to agree on which specific properties aggregates should satisfy. However, we can identify one point which they all have in common: interesting aggregates have hidden components. Aggregates with hidden components are implemented using encapsulation. This permits an aggregate to be seen as a single object at one level of abstraction, and so it can be used as a structuring mechanism.

We agree that most advantages come from using aggregates with hidden components. However, instead of treating and representing aggregation-with-sharing as a regular form of conceptual relationships, we use the terminology of aggregation. By doing this, we are giving more information about the composition of an object, improving the understability of the relationship that connects the two objects.

We propose that aggregates have the following properties:

- Aggregates may have extra functionality in addition to the functionality defined in their components.

- Object components at the same level of abstraction (within the same aggregate) can communicate with each other without having to communicate via their aggregate.

- There is a mechanism of service delegation (propagation of operations) between objects at consecutive levels of abstraction.
- An aggregate acts as the interface to, or manager of, the aggregate components. It calls services of the components, but the components do not call services of the aggregate.
- In general, aggregation is not transitive if the semantics of the relationships involved differ. Aggregation is never transitive with respect to message connections. Suppose that aggregate A has a component B and B has a component C. If A communicates with B, and B communicates with C, this does not imply that A communicates (directly) with C.
- Aggregation is antisymmetric: if B is a component of A, then A is not a component of B.

Aggregates with hidden components satisfy the following extra properties:

- An aggregate encapsulates its components.
- Each component is hidden from the rest of the world, i.e. it is not visible outside the scope of the aggregate.
- The deletion of an aggregate implies the deletion of all its components.

Aggregates with shared components satisfy the extra following properties:

- The components cannot be encapsulated as they must be visible to other objects outside the scope of the aggregate.
- If necessary, a component may know about its aggregates. This is the case for catalog aggregation, where it may be useful for a component to know to which aggregates it belongs.
- The deletion of a component is only possible if there are no other aggregates with references to it. It may be initiated by other objects in the model.
- The deletion of an aggregate does not imply the deletion of its components.
- The aggregate can exist after the deletion of its components.
- The creation of a component may be initiated by other objects in the model.

7 Managing Complexity

Aggregation gives a mechanism for structuring large systems. Such structuring may be accomplished either *top-down* or *bottom-up*. In a top-down approach, aggregates can show up early. As the development evolves, these complex objects are refined and their components identified. Even when we identify aggregates early, we do not have to guarantee their correct classification into hidden or shared. As we will show in Section 8.4, moving from one type to the other is very simple and brings no problems. In a bottom-up approach, we may start identifying the lower level objects and then we may group some of them to form aggregates. Object-oriented developments have better results when we use a mixture of these two approaches.

In some situations, depending on the style of the requirements, it may be difficult to start with a top-down approach. We can identify two reasons for this. First, if the requirements are written in a functional way, when dividing a system according to its functional areas it may be that different areas describe parts of the same object. If each team takes a functional area, some of the teams may have different views of the same object. For a large project it is necessary to give names to candidate objects, without spending much time considering whether or not a given object is important. The analysts can use their knowledge about the real world. Next, we can make groupings of objects according to their role in the system and keep interactions between groupings low. Each group of object is then given to different teams who proceed with the analysis using a mixture of top-down and bottom-up approaches.

Second, the requirements document may describe the problem in a very flat style, without hierarchical structure. It is also possible that the users describe physical details of the problem and are unable to abstract concepts. If this happens, the candidate objects we start identifying are certainly low level objects. As the development proceeds, more complex objects, such as aggregates, are identified.

Most OOA methods start by identifying objects. Depending on the style used to write the requirements document, we can identify aggregates sooner rather than later. We believe that in a large project the objects we start identifying are a mixture of complex and simple objects. We can identify complex objects in two ways: by analysing relationships and similarities between objects (bottom-up composition) and, while describing an object we may identify it as being a complex object (top-down decomposition).

Therefore, by analysing relationships and similarities between objects we can identify more complex objects (bottom-up composition) and while describing an object we may identify it as a complex object (top-down decomposition).

8 Modelling Aggregation in LOTOS

We have shown in [11] how to model objects and classes in LOTOS. An object is normally modelled by a process and one or more ADTs, but when an object in the object model only plays the role of attribute of another object, it is modelled by a single ADT.

To be useful as an abstraction, an aggregate must play an important role in the system and so it is modelled as a process and one or more ADTs. If the object only plays the role of an attribute, although an attribute may have internal structure, we do not regard it as an aggregate. For example, although we can consider the attribute `Address` to be composed of `House_Number`, `Street Name`, `Post Code`, `City Name` and `Country Name`, `Address` is modelled as an ADT which is a combination of sorts (one sort for each component). Also, if the components of an object only play the role of attributes, the object is modelled as an ordinary object, not as an aggregate.

This section is concerned with showing how to model aggregates in LOTOS. As many authors suggest, we model aggregation with shared components as conceptual relationships. The aggregate and each component are defined by separate processes, and the aggregation (the relationship) is modelled in the same way as for conceptual relationships. Unless something different is said in the requirements, the aggregate knows its components, but the components do not know about the aggregate.

An aggregate with hidden components is modelled as a process which encapsulates a process for each of its components together with a process manager, or interface. As a rule, we give the same name to both the process representing the aggregate and the process representing the interface. The following subsections discuss this in detail. A hybrid aggregate has hidden and shared components. The hidden components are modelled as processes embedded by the process defining the aggregate, while the shared components are modelled as separate processes. A reference to each separate process is modelled as an ADT and given as a parameter of the aggregate's process.

While a simple object may be regarded as a sequential machine, an aggregate has the implicit connotation of having internal parallelism [4].

Having a static or a dynamic number of object components does not complicate the problem, since we can easily define object generators. If we are dealing with aggregation with hidden components, the object generators for each component are defined inside the aggregate and therefore they are not visible from the outside. This allows the same set of identifiers to be used by different object generators defined inside different aggregates. The advantage of this is its simplicity. However, each component is then only uniquely identified when the aggregate identifier is given together with the component identifier. We do not see this as a problem, since the components are not visible to the other objects in the system and so the aggregate identifier must always be used to access them.

To demonstrate how aggregates can be modelled in LOTOS, we use a simple video player with four functions: load a tape, play a tape, stop playing and eject a tape. The video has two components: a motor and an eject mechanism. Its behaviour is given by the finite state automaton depicted in Figure 10.

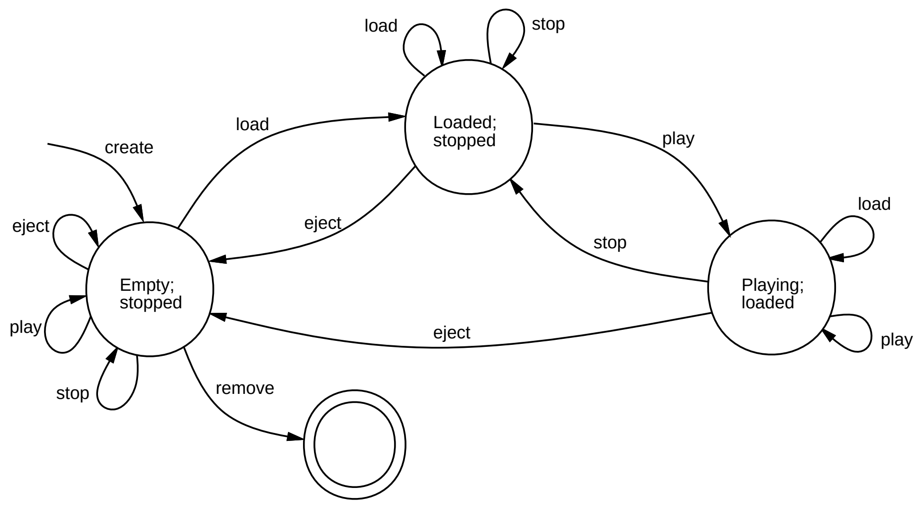

Figure 10: Behaviour of the video player example defined as a finite state automaton

8.1 Aggregation with Hiding

If the number of components is constant, we can define the internal composition of the aggregate at specification time. Creation or deletion of the aggregate implies the creation or deletion of its components. Therefore, the deletion of a component is impossible without the deletion of the aggregate.

Let us suppose that the video player has one motor and one eject mechanism. Figure 11 shows the class templates for `Video`.

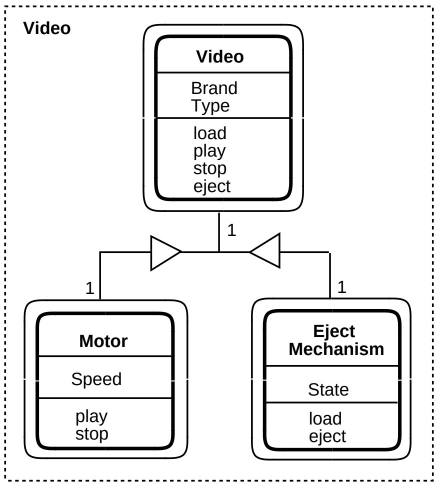

Figure 11: Video player aggregate

As the components are not shared, we draw a dotted box (structure) around the aggregate. The semantics of this is that the class components `Motor` and `Eject Mechanism` are encapsulated within the aggregate `Video`, and so they are not visible outside `Video`. By following the algorithm given in [11] we can build an Object Communication Diagram (OCD) as depicted in Figure 12. (In this simple example, the rest of the system is the interface scenario.)

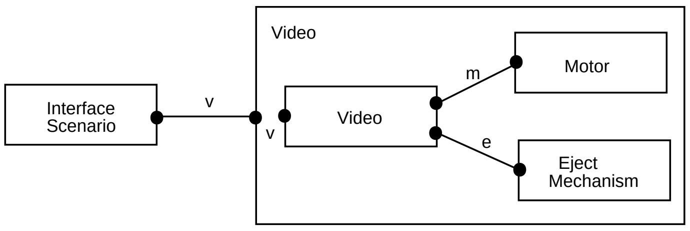

Figure 12: Object Communication Diagram

In LOTOS, the top level behaviour expression takes the form:

```
( Video[v](Make_Video(id1 of Video_Id, empty of State_V,
                      id1 of Motor_Id, id1 of Eject_Mechanism_Id))
|[v]|
  Interface_Scenario[v]
)
where
  process Video[v](this_video: Video_State): noexit :=
    hide m, e in
    ( Video[v, m, e](this_video)
    |[m,e]|
      ( Motor[m](Make_Motor(Get_Motor_Id(this_video), rest of State_M))
      |||
        Eject_Mechanism[e]
          (Make_Eject_Mechanism(Get_Eject_Mechanism_Id(this_video),
                                 empty of State_E))
      )
    )
  where
    process Video[v, m, e]      ... endproc (* Video *)
    process Motor[m]            ... endproc (* Motor *)
    process Eject_Mechanism[e] ... endproc (* Eject_Mechanism *)
  endproc (* Video *)
```

As we discussed before, we name the complete aggregate with the same name as the box at the top level of the aggregation. This is shown in LOTOS by having two processes with the same name, one encapsulating the other. At one level of abstraction the outside process represents the aggregate (i.e. the box at the top level of the hierarchy) while at a lower level of abstraction the inner process represents the box at the top level of the hierarchy which is now regarded as the interface to the component objects.

As gates `m` and `e` are hidden in the external process `Video`, `Motor` and `Eject Mechanism` are encapsulated within `Video` and defined after the keyword `where`. The two components are therefore hidden from `Interface Scenario`. In this example we are only creating one instance of each class template `Motor` and `Eject Mechanism`, but we could create others by having more process instantiations in the behaviour expression. The operations `Get_Motor_Id` and `Get_Eject_Mechanism_Id` are defined in the ADT that defines the sort `Video_State`, as follows:

```
  type Video_Type is Video_Id_Set_Type, State_V_Type, Motor_Id_Set_Type,
                     Eject_Mechanism_Id_Set_Type
    sorts Video_State
    opns
      Make_Video             : Video_Id, State_V, Motor_Id,
                               Eject_Mechanism_Id   -> Video_State
      Change_State           : Video_State, State_V -> Video_State
      Get_Video_Id           : Video_State          -> Video_Id
      Get_Motor_Id           : Video_State          -> Motor_Id
      Get_Eject_Mechanism_Id : Video_State          -> Eject_Mechanism_Id
      ...
  eqns forall v: Video_State, n: Video_Id, m: Motor_Id,
              e: Eject_Mechanism_Id, s, s1: State_V
    ofsort Video_Id
        Get_Video_Id(Make_Video(n, s, m, e)) = n;
        Get_Video_Id(Change_State(v, s))     = Get_Video_Id(v);
    ofsort Motor_Id
        Get_Motor_Id(Make_Video(n, s, m, e)) = m;
        Get_Motor_Id(Change_State(v, s))     = Get_Motor_Id(v);
    ofsort Eject_Mechanism_Id
        Get_Eject_Mechanism_Id(Make_Video(n, s, m, e)) = e;
        Get_Eject_Mechanism_Id(Change_State(v, s))     = Get_Eject_Mechanism_Id(v);
    ofsort ...
  endtype
```

In LOTOS, defining an aggregate with a dynamic number of components is not difficult, since we have the facility of defining object generators. An object generator allows us to create multiple instances of objects. Supposing that the number of components was dynamic, the external process Video would be defined as:

```
process Video[v](this_video: Video_State): noexit :=
  hide m, e in
    ( Video[v, m, e](this_video)
    |[m, e]|
      ( Motors[m]({} of Motor_Id_Set)
      |||
        Eject_Mechanisms[e]({} of Eject_Mechanism_Id_Set)
      )
    )
where
   ...
endproc (* Video *)
```

in which Motors and Eject Mechanisms are object generators, each one initialised with an empty set of identifiers. As the components are hidden, their process definitions are in the scope of the external process Video, after the keyword where. As an example, let us consider the object generator Motors:

```
process Motors[c](mts: Motor_Id_Set): noexit :=
  c !create ?id: Motor_Id [id notin mts];
  ( Motor[c](Make_Motor(id of Motor_Id, rest of State_M))
  |||
    Motors[c](Insert(id, mts))
  )
where
  process Motor[c](this_motor: Motor_State): noexit :=
    ( [Get_State(this_motor) eq rest] ->
        ( c !play !Get_Motor_Id(this_motor);
          exit(Change_State(this_motor, run))
        )
    []
      ...
  endproc (* Motor *)
endproc (* Motors *)
```

Motors holds the set of identifiers already created. (Notice that for simplicity we are using the same gate c to create a motor and to operate the motor, but we could use two different gates.)

When Interface Scenario requires a service, Video routes the request to the right component and then returns the result, if any. When more than one object component is ready to synchronize, one component will be chosen non-deterministically. The inner Video process, i.e. the process defining the interface of the aggregate, can be defined as:

```
process Video[v, m, e](this_video: Video_State): noexit :=
  ( hide create_motor, create_eject in
     [Get_State(this_video) eq loaded] ->
```

```
          ( v !play !Get_Video_Id(this_video);
            m !play ?m1: Motor_Id [m1 IsIn Get_Motor_Id_set(this_video)];
            exit(Change_State(this_video, playing))
          []
            ...
          )
      []
        ...
      []
        create_motor;
        m !create ?idm: Motor_Id;
        exit(Add_Motor(this_video, idm))
      []
        create_eject;
        e !create ?ide: Eject_Mechanism_Id;
        exit(Add_Eject_Mechanism(this_video, ide))
      ) >> accept upd_video: Video_State in Video[v, m, e](upd_video)
  endproc (* Video *)
```

where `create_motor` and `create_eject` are internal events. The behaviour expression:

```
  m !play ?m1: Motor_Id [m1 IsIn Get_Motor_Id_set(this_video)];
```

may synchronize with any motor which is in the right state and whose identifier is known by the aggregate.

The creation of new motors is defined in the body of the inner process `Video`, by using the internal event `create_motor` and then synchronizing on event:

```
  m !create ?idm: Motor_Id;
```

with the corresponding event defined in the object generator. The `Eject_Mechanism` is dealt with in a similar way.

The operation `Add_Motor` and `Add_Eject_Mechanism` are defined in the ADT `Video_Type`. This ADT has to be changed to support sets of motors and sets of eject mechanisms.

When other objects in the system know about the existence of the components, they can initiate the creation of a new component. However, the components are not visible, and so it is always the aggregate's responsibility to select the right component and to create new ones, by using the object generators.

As a result of having encapsulated components, another instance of `Video` can use the same component identifiers. This means that the identifier of a component must be combined with the identifier of the aggregate to give the full object component identifier.

Although a dynamic number of hidden components can be specified in LOTOS, we believe that it is not a common case. One situation is when a component breaks down. If a component is not responding to the services required, the aggregate can substitute it with a new component. The AT&T ESS5 switch has 'auditors' which go around checking invariants in software modules which have a certain functionality. If a module does not satisfy the invariant, the auditor shuts them down and reinitialises them, or replaces them with other instances [6].

8.2 Aggregation with Sharing

A shared component exists independently of the aggregate. It is modelled in the usual way, with a class template and perhaps an object generator, but outside the aggregate. The aggregation relationship will then be modelled as a conceptual relationship in the ADT that defines the state information. In general, the aggregate has a reference to the shared component, while the shared component only has a reference to the aggregate if it is explicitly required.

As the components are shared, the deletion of the aggregate does not imply the deletion of its components, as happened in the previous section. However, the deletion of a component may imply the deletion of the aggregate.

Let us recall the video player example, where we have two object components, one for each component class. Figure 13 shows the OCD supposing that `Motor` and `Eject Mechanism` can be directly accessed by `Interface Scenario`.

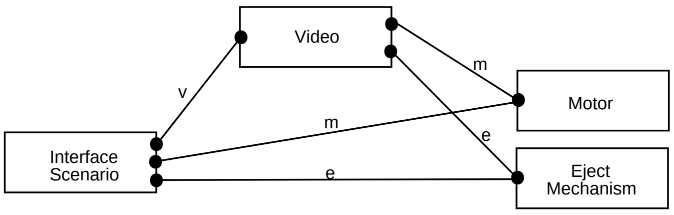

Figure 13: Object Communication Diagram

In LOTOS, the top level behaviour expression takes the form:

```
( Interface_Scenario[v, m, e]
|[v]|
  Video[v, m, e](Make_Video(id1 of Video_Id, empty of State_V,
                              id1 of Motor_Id, id1 of Eject_Mechanism_Id))
)
|[m, e]|
  ( Motor[m](Make_Motor(id1 of Motor_Id, rest of State_M))
  |||
    Eject_Mechanism[e]
       (Make_Eject_Mechanism(id1 of Eject_Mechanism_Id, empty of State_E))
  )
where
 ...
```

The difference between this situation and aggregation with hidden components is that the communication gates `m` and `e` are visible from the `Interface_Scenario`. Process `Video` corresponds to the interface in the case of aggregation with hiding, although here we want to regard it as the whole aggregate. The ADT that specifies the video state information is unchanged. The information about the relationship between `Video` and `Motor` and between `Video` and `Eject Mechanism` is given by the parameters of `Make Video`. Instead of modelling these relationships in the ADT, we could model them as extra parameters of the process `Video`, as we do for normal conceptual relationships. However, we prefer the first option.

If we had a dynamic number of object components, the parameters of `Make Video` would be sets of identifiers, instead of single identifiers, and the LOTOS top level behaviour expression would instantiate an object generator for each component, instead of instantiating each component.

We can also model catalog aggregation. For example, suppose that two videos could share a motor and an eject mechanism. This can be represented as:

```
( Interface_Scenario[v, m, e]
|[v]|
  ( Video[v, m, e](Make_Video(id1 of Video_Id, empty of State_V,
                              id1 of Motor_Id, id1 of Eject_Mechanism_Id))
  |||
    Video[v, m, e](Make_Video(id2 of Video_Id, empty of State_V,
                              id1 of Motor_Id, id1 of Eject_Mechanism_Id))
  )
)
|[m, e]|
  ( Motor[m](Make_Motor(id1 of Motor_Id, rest of State_M))
  |||
    Eject_Mechanism[e]
       (Make_Eject_Mechanism(id1 of Eject_Mechanism_Id, empty of State_E))
  )
where
 ...
```

When modelling catalog aggregation we may want to change the ADT of the component to deal with an extra attribute which gives the set of aggregates in which it takes part. In the example given in Section 5.3, it is desirable that a person knows about his or her research groups.

While in the case of static aggregation with hiding the aggregate had sole responsibility for creating its components at specification time, in the case of dynamic aggregation the creation and deletion of an object component may be initiated by other objects in the system. Also, if the object requiring the service knows the identifier of the object component which will be involved in the operation, we can use that information when asking the service to `Video`, instead of using value generation:

```
process Video[v, m, e](this_video: Video_State): noexit :=
  ( [Get_State(this_video) eq loaded] ->
    ( v !play !Get_Video_Id(this_video) ?m1: Motor_Id
         [m1 IsIn Get_Motor_Id_set(this_video)];
      m !play !m1;
      exit(Change_State(this_video, playing))
    []
      ...
    )
  []
   ...
endproc (* Video *)
```

8.3 Sharing Concepts but not Objects

In Sections 5.2 and 5.3 we discussed some different views of sharing. When the component class is shared, the object components may or may not be. The LOTOS behaviour expressions in Section 8.2 give us both shared component classes and shared object components. A shared component class and non-shared object component can be obtained by proceeding as we did for hidden components, but now the component classes are defined outside the scope of the external `Video` process:

```
process Video[v](this_video: Video_State): noexit :=
  hide m, e in
  ( Video[v, m, e](this_video)
  |[m,e]|
    ( Motor[m](Make_Motor(Get_Motor_Id(this_video), rest of State_M))
    |||
      Eject_Mechanism[e]
          (Make_Eject_Mechanism(Get_Eject_Mechanism_Id(this_video), empty of State_E))
    )
  )
where
  process Video[v, m, e] ... endproc (* Video *)
endproc (* Video *)

process Motor[g] ... endproc (* Motor *)
process Eject_Mechanism[g] ... endproc (* Eject_Mechanism *)
```

8.4 Hiding and Sharing: Moving Around

Having modelled aggregates with hidden components and aggregates with shared components in LOTOS, let us discuss the changes necessary to transform one into the other. The basic difference in modelling these two kinds of aggregates is concerned with encapsulation and information hiding. While hidden components are encapsulated by the aggregate and hidden from the other objects in the model, shared components are modelled as separate processes which are visible from outside the aggregate.

As we have discussed in Section 5.1, in order to encapsulate a hidden component into its aggregate, we create a higher level structure which we name with the same name as the class template at the top of the aggregation hierarchy in the object model.

To transform an aggregate with hidden components into an aggregate with shared components, we follow the two steps:

1. Replace the process that defines the higher level structure, i.e. the outside process, with its behaviour expression which joins the inner process with the components. The inner process now plays the role of the higher level structure.

2. Make all the gates visible outside the aggregate's scope, removing the `hide` operator, and add those gates to any process instances which need them.

The opposite, i.e. transforming an aggregate with shared components into an aggregate with hidden components, can be accomplished in two steps:

1. Encapsulate the aggregate and components processes within an extra process. Name this process with the same name as the previous aggregate. (Now, the outside process represents the aggregate and the inner process represents the interface of the whole structure with the other objects in the model.)

2. Hide, in the encapsulating process, the gates which are used to communicate with the aggregate components.

The procedures above can both be applied when dealing with object generators.

9 Conclusions

Aggregation is a useful concept which can be used to control the size and complexity of a large system. Aggregation with hidden components mainly uses the concepts of abstraction, encapsulation and information hiding. This helps in providing a top-down approach which aids the software engineer developing the system and, at the same time, guides the reader to understand the system.

Aggregation with shared components does not bring as many advantages and many authors advocate that it should be treated as an ordinary conceptual relationship. However, we believe that there are advantages in showing it in an object model. It will give hints about the structure of a system, helping us to understand it, and it can also give directions for reusability. While a conceptual relationship is more likely to change when the system is put in a different context, an aggregation may not change and so we can see it as reusable in other contexts. That is why we propose modelling it within the ADT, instead of modelling it as an extra argument in the process template, as we do with conceptual relationships.

LOTOS can to model aggregation with shared and with hidden components. Encapsulation is dealt with by changing the scope of processes and using the `hide` operator. By defining sets and object generators, LOTOS also deals well with the cases of a static and a dynamic number of object components.

10 Acknowledgments

We would like to thank Peter Ladkin, Ken Turner and Pedro Guerreiro for their helpful comments on an earlier version of this report. Also, we would like to thank Gary Marsden for discussions about the video example.

This work has been supported by the Junta Nacional de Investigação Científica e Tecnológica (JNICT), Portugal.

References

[1] M. Blaha. Aggregation of Parts of Parts of Parts. *Journal of Object-Oriented Programming*, 6(5):14–20, September 1993.

[2] Franco Civello. Roles for Composite Objects in Object-Oriented Analysis and Design. In *Proceedings OOPSLA'93, ACM SIGPLAN Notices*, volume 28, pages 376–393, October 1993.

[3] P. Coad and E. Yourdon. *Object Oriented Analysis.* Yourdon Press, Prentice-Hall, 2nd edition, 1991.

[4] D. de Champeaux, D. Lea, and P. Faur. *Object-Oriented System Development.* Addison-Wesley, 1993.

[5] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In *ECOOP'92*, Lecture Notes in Computer Science, 615, pages 57–77. Springer-Verlag, June/July 1992.

[6] R. Hoare, C.A. How did Software Get so Reliable Without Proof?, March 1994. BCS Proof Club, Edinburgh.

[7] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach.* Addison-Wesley, 1992.

[8] R. Jungclaus, G. Saake, R. Hartman, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Technical Report 91-04, Informatik-Bericht, TU Braunschweig, 1991.

[9] W. Kim, J. Banerjee, H. Chow, J.F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. In *OOPSLA'87*, volume 22, pages 118–125, September 1987.

[10] A.M.D. Moreira and R.G. Clark. LOTOS in the Object-Oriented Analysis Process. In *BCS-FACS Workshop on Formal Aspects of Object-Oriented Systems*, Imperial College, London, December 1993. *BCS-FACS (British Computer Society – Formal Aspects of Computing Science).*

[11] A.M.D. Moreira and R.G. Clark. ROOA: Rigorous Object-Oriented Analysis. Technical Report TR 109, Department of Computing Science and Mathematics, University of Stirling, Scotland, October 1993.

[12] A.M.D. Moreira and R.G. Clark. Combining Object-Oriented Analysis and Formal Description Techniques. In *Proceedings of ECOOP'94.* Springer Verlag, Lecture Notes in Computer Science, 1994. To appear.

[13] J. Odell. Six Different Kinds of Composition. *Journal of Object-Oriented Programming*, 6(8):10–15, January 1994.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall, 1991.

[15] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation. *Communications of the ACM*, 20(6):405–413, June 1977.

[16] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.