KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE
ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 0034

CHARACTERISING AGGREGATIONS WITH EXISTENCE DEPENDENCY

by
M. SNOECK
G. DEDENE

D/2000/2376/34

Characterising Aggregations with Existence Dependency

Monique Snoeck, Guido Dedene

K.U.Leuven - Management Information Systems Group
Naamsestraat 69
3000 Leuven
Belgium

Email: Monique.Snoeck, Guido.Dedene@econ.kuleuven.ac.be

ABSTRACT: *The concept of aggregation is considered as one of the basic principles in object-oriented analysis. There is however no standard definition of this concept and each object-oriented analysis method has its own definition of aggregation. The aim of this paper is not to discuss the different types of aggregation that exist. However, having assessed the complexity of the concept, we will illustrate how a basic set of formal concepts is sufficient to define of the structural and behavioural aspects of different existing flavours of aggregation. If a development method wants to offer a rich concept such as aggregation, it can define the semantics of the desired flavour of the aggregation using these core formal concepts. Analysts then have the choice to use the aggregation defined by the method or to fall back on the core concepts if a different flavour of aggregation is needed to model the situation at hand.*

KEYWORDS: *object-oriented analysis, aggregation, composition, conceptual modelling*

An earlier and abbreviated version of this paper has been accepted for publication in *L'objet*, under the title "Core Modelling concepts to Define Aggregations".

1. INTRODUCTION

The aggregation concept is considered as one of the basic principles of the object-oriented approach. Yet at the same time, it is the subject of a lot of (heated) discussions. The reason for these discussions are the loosely described characteristics of the concept. One of the latest paper on the subject [HEN 99a][HEN 99b] identifies not less than 9×2^9 different flavours of aggregation. The use of such loosely defined concepts is not enhancing the quality of specifications. It is a well-known fact that the correctness of a software system is directly related to the correctness of its specifications. Correct specifications require the use of precisely, completely and explicitly defined modelling concepts.

The goal of this paper is not to continue the debate on the characterisation of aggregation. Rather, having assessed the complexity of the concept, we have developed a toolbox that allows defining many flavours of aggregation in a precise and formal way. In this toolbox, we put a few simple and formally defined concepts that are the result of several years of experience with a minimalistic approach to object-oriented conceptual modelling. In such minimalistic approach, modelling techniques offer a limited set of very well-defined and simple concepts. Analysts have to unravel the problems to a level where these simple concepts suffice to describe the problem at hand. The disadvantage of such approaches is that the models they generate tend to contain much more elements compared to models using semantic rich concepts. Using a layered approach can however alleviate this problem: a particular pattern of simple lower level concepts can be used to represent a single high level concept. In the diagrams, the patterns of lower level concepts are replaced by a single "high level" icon to make models more readable. The advantage is that the high level concept is defined in terms of the lower level concepts. Because of their simplicity, the lower level concepts are much easier to define formally. The high-level concept benefits from this formal definition: its own formal definition can be inferred from the formal definition of its constituent lower level concepts.

In this paper we will attempt to define various existing kinds of aggregation in terms of the concept of *existence dependency*. Existence dependency has been defined in [SNO 98]. That paper demonstrates that this concept can be used to model any kind of association. Hence, existence dependency is a core concept for associations. In this paper we demonstrate that the same core concept can be used to characterise many different flavours of aggregation.

In addition to the structural aspects of aggregation, we also have to consider the behavioural aspects. Indeed, the concept of aggregation is strongly linked to the concept of propagation of behaviour (from the whole to its parts). The exact details of this propagation (when and how) are however different from case to case. For example, when an order is deleted, the order-lines are deleted as well. However, when a department ceases to exist (due to re-organisations), it is not sure whether all its sub-departments cease to exist as well. There are so many options on how and when to propagate behaviour, that any attempt to model all behavioural aspects of the aggregation using a structural concept only, will be very difficult. In the approach proposed in this paper, the behavioural aspects are modelled using the concepts of atomic and consistent events.

The concepts of existence dependency and of atomic and consistent events are called *core* concepts. By the term ***core*** we first mean that they are simple, unambiguous concepts, defined in a formal way [SNO 98, SNO 99]. In addition, being a ***core*** concept means that it can be used to describe more complex concepts. If a development method wants to offer a rich concept such as aggregation, it can define the semantics of the particular flavour of the aggregation using existence dependency and events. The analysts then have the choice to use the aggregation defined by the method or to fall back on the core concepts if a different flavour of aggregation is needed to model the situation at hand. This layered approach can also be followed on a project by project basis by defining high-level concepts only applicable in the context of one particular project.

The rest of the paper is organised as follows. Section 2 briefly presents the structural aspects of existence dependency. Section 3 presents a structural characterisation of aggregation, using only the existence dependency relationship and the notions of separability and shareability of parts. Section 4 elaborates on the behavioural aspects of existence dependency and introduces the notion of atomic and consistent event. Section 5 then explains how the behavioural aspects of aggregation can be further characterised using the concepts of atomic and consistent events. Finally, section 6 discusses how other characteristics such as homeomerousity, encapsulation, transitivity, ... etc. can be expressed with the proposed core concepts.

2. STRUCTURAL CHARACTERISTICS OF EXISTENCE DEPENDENCY

Existence dependency is defined as follows:

The concept of existence dependency (ED) is based on the notion of the "life" of an object. The life of an object is the span between the point in time of its creation and the point in time of its end. Existence dependency is defined at two levels: at the level of object types or classes and at the level of object occurrences. The existence dependency relation is a partial ordering on objects and object types which is defined as follows:

Definition

Let P and Q be object types. P is existence dependent on Q if and only if the life of each occurrence p of type P is embedded in the life of one particular *and always the same* occurrence q of type Q. p is called the *dependent object,* (P is the dependent object type) and is existence dependent on q, called the *master object* (Q is the master object type).

A more informal way of defining existence dependency is as follows:

If each object of a class P always is associated with minimum one, maximum one *and always the same* occurrence of class Q, then P is existence dependent on Q.

The result is that the life of the existence dependent object can not start before the life of its master. Similarly, the life of an existence dependent object ends at the latest at the same time that the life of its master ends. This is illustrated in Figure 1.

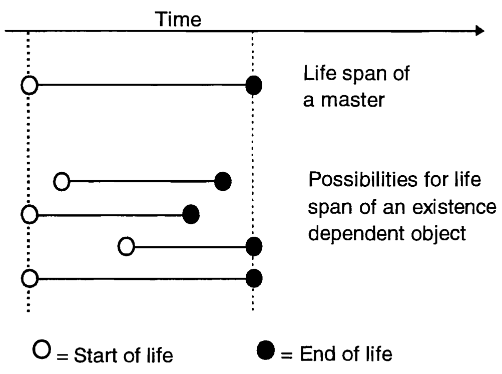

Time
Life span of a master
Possibilities for life span of an existence dependent object
O = Start of life
● = End of life

Figure 1. Life span of master and dependent object

As an example, let us consider the relationships between an object type LOAN and an object type COPY in a library. The life span of a loan of a copy is always embedded in the life span of the copy that is on loan. Indeed, we cannot have a loan for a copy if the copy doesn't exist. And the lifecycle of the copy cannot end as long as the lifecycle of the loan is not ended. In addition, a loan always refers to one and the same copy for the whole time of its existence. Hence the object type LOAN is existence dependent on the object type COPY.

Notice that a total or mandatory relationship implies some kind of existence dependency: if a relationship relates PROJECT to EMPLOYEE and is total on the side of EMPLOYEE, then an occurrence of EMPLOYEE can only be created if at the same time it is related to some PROJECT-occurrence. The other way round, the deletion of an entity occurrence of PROJECT results in the deletion of all occurrences of EMPLOYEE that were *only* related to that occurrence of PROJECT. In this sense, one could say that EMPLOYEE is existence dependent of PROJECT. As pointed out in [DOG 90], [PUT 88], an important difference between a mandatory and an existence dependency relationship lies in the updating rules. Indeed in the example of the project and the employee, an employee can be linked to different projects in the course of time. In other words, the reference from employee to project can be updated. In the definition of existence dependency given above it is required that the existence dependent object refers to one and *always the same* master object: the reference from the dependent to the master object is immutable. According to this definition, an employee is *not* existence dependent on a project. We simply say that the relationship between the two objects is mandatory for employee. Similarly a relationship between a composite and its components can be mandatory without being existence dependent: if an order requires at least one order line to exist, the relationship is said to be mandatory but not existence dependent because in this case it needs not to be always the same order line.

Existence dependency can be augmented with the notion of cardinality. The cardinality of the existence dependency relationship defines how many occurrences of the dependent object type can be dependent on one master object at one point in time[1]. For example, in the library, at one point in time a copy can be involved in at most one loan, while a member can have several loans going on simultaneously. To avoid confusion with classic associations, the existence dependency relationship has its own particular graphical representation given in Figure 2. Since the existence dependency associations are always mandatory for the existence dependent object type and since the cardinality is always one on that side, optionality and cardinality are only indicated for the master object type. A white dot indicates that the participation to the relationship is optional (a master *can* have dependents) while a black dot indicates that participation is mandatory (each master has *at least one* dependent *at any point in time*). The arrow indicates a cardinality of Many (a master can have multiple dependents at any one time) while a straight line without an arrow denotes a cardinality of one (each master has *at most one* dependent *at any point in time*).

Let us remark that in the last case it might appear as if the lifecycle of A equals the lifecycle of B. However, since the relationship expresses existence dependency on the side of B only, an occurrence of A can still be associated with many occurrences of B **consecutively**. For example, assume that A stands for the entity type REAL ESTATE and B for OWNERSHIP. Then an ownership always refers to the same real estate once and for all. But since real estates can be traded, a real estate can be involved in one to many ownerships *consecutively*. For this reason it is important to keep the representation of a one to one relationship asymmetric. If not, it is impossible to discern the existence dependent object type from the master object type.

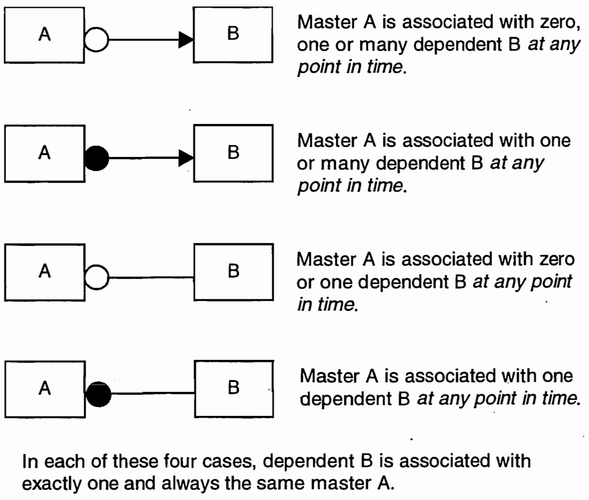

A
B
Master A is associated with zero, one or many dependent B *at any point in time.*
A
B
Master A is associated with one or many dependent B *at any point in time.*
A
B
Master A is associated with zero or one dependent B *at any point in time.*
A
B
Master A is associated with one dependent B *at any point in time.*
In each of these four cases, dependent B is associated with exactly one and always the same master A.

Figure 2. Graphical notation for existence dependency

For the library example the existence dependency graph is given in Figure 3: a copy has zero or one loans at one point in time, a member has zero to many loans at one point in time, and a loan refers to exactly one and always the same copy and member at any time.

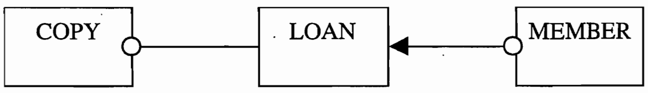


Figure 3. Existence dependency relations for the library

1. Notice that the clause "at one point in time" is essential in the definition of the cardinalities. Over time, most objects of a certain type can have many existence dependent objects of another type.

Most object-oriented analysis methods have an Entity-Relationship like technique for modelling static aspects. In the conceptual model of MERODE [SNO 99][SNO 98], it is the existence dependency graph that fulfils this purpose: all object types have to be related according to existence dependency. At first sight it seems not so obvious that organising object types according to existence dependency is always possible. However, in [SNO 98] we have demonstrated by means of a few examples that this is in fact pretty straightforward. The general idea is that associations between object types either represent an existence dependency or not. In the first case, the association stays as it is. In the second case, the association is transformed into an object type which is existence dependent of all the object types involved in the association. A similar reasoning applies to composition (or aggregation): either the components are existence dependent of the composition or they are not. In the latter case, as many "contract" object types are introduced as there are components in the composition. Each such contract object type relates and is existence dependent of both the composition and the component. It represents the period of time that the component is part of the composition. For example, wheels are part of a car but are not existence dependent of a car since they can be mounted on different cars in the course of their life. Similarly, a car is not existence dependent on a (set of) wheel(s), because the life of a car is not embedded in the life of a particular (set of) wheel(s). As a result, we need a MOUNTING object that represents the period of time that a particular wheel is mounted on a car. This paper elaborates this basic idea of modelling non-existence dependent relationships by means of a "contract" object for the specific case of part-of relationships. This contract object specifies the conditions under which a part and a whole agree to be bound.

3. STRUCTURAL CHARACTERISATION OF AGGREGATION

In this section, we try to characterise structural aspects of the whole-part relationship. We will do so by considering two characteristics. The first is existence dependency of parts on the whole, which will imply some features related to lifetime binding and separability[2]. The second characteristic is the shareability of parts. To simplify the discussion, we will always assume that the whole can have many parts. The cases where the whole can have at most one part can however be described in a completely analogous way. As existence dependency implies inseparability the cases to discuss can be resumed as in Table 1.

Existence Dependency	**Separability**	**Shareability**	**Discussed in paragraph**
no	yes	yes	3.1.1.
		no	3.1.2
yes	no	no	3.2.1
		yes	3.2.2
no	no	yes	3.3
		no	3.3
yes	yes	-	impossible

Table 1. Cases to discuss

In the remainder of this paper we assume that all relationships are whole-part relationships. We also assume that the whole and the part can have emergent properties, which are modelled as attributes or functions (i.e. procedures that return a value) in the class definitions. The whole can also have resultant properties, which are defined as functions in terms of the properties of its parts. The other primary characteristics (irreflexivity, anti-symmetry, and asymmetry) and the remaining secondary characteristics identified in [HEN 99a] are discussed in section 6.

3.1. Separable parts

Separability means that during its lifetime, a part can be detached from the whole and bound to a different whole. As a result, a part can have several bindings with different wholes during its lifetime and by definition the part is not existence dependent on the whole. In case of non-shareable parts, the

[2] We do not consider the case of a whole being existence dependent on a part: according to the given definition of existence dependency this would imply that the whole can only exist in the context of one particular part.

bindings must be consecutive. When parts are also shareable, the parts can have overlapping bindings. These types of aggregation can be modelled with existence dependency by modelling the period that a part is bound to the whole as an object type in itself. The model of the aggregation thus contains three object types: the whole, the part and the binding, the latter being existence dependent on the first two. Indeed, the lifetime of a binding always falls within the lifetimes of both the part and the whole and it refers to the same part and whole for the whole duration of its existence. The binding object type refers to the fact that in an aggregation structure, the lifetime of a part always overlaps the lifetime of the whole [SAK 98]. In addition the binding object is a kind of contract that specifies the conditions under which a part and a whole agree to be bound.

3.1.1. Shareable parts

Let us consider CLUB as an aggregation of MEMBERs (member-bunch composition as in [ODE 94], [HEN 97]). For example, a club can have many members and can initially have no members at all. Members are "shareable" because they can be members of several clubs. A person can also exist without being member of a club and is therefore not existence dependent of club. The BINDING object type is in this example the registration of a person as member of a club.

Shareable parts[3] can be bound to more than one whole at one point in time. Hence PART has a one-to-many relationship with BINDING: one part can have several bindings at one point in time. The four variants for this type of aggregation are shown in Figure 4. In case (a) the whole has zero to many shareable parts. Parts can exist without being part of a whole. In case (b) the whole has zero to many shareable parts. Parts must be attached to at least one whole to exist. In case (c) the whole has at least one, and possibly more shareable parts. Parts can exist without being part of a whole. Finally, in case (d), the whole

[3] The cases only discuss homogenous sharing. Heterogeneous sharing, where parts can be shared across wholes of different types, are modelled with several whole-part relationships which can each be characterised independently according to the cases discussed here.

has at least one, and possibly more shareable parts. Parts must be attached to at least one whole to exist. The same aggregations can be represented in a more concise manner by "minimising" the icon representing the binding (Figure 5). Cases (b) and (d) are examples of constrained sharing, which means that a part must be associated with a whole at any time [SAK 98]. Case (c) models for example clubs that need at least one member to exist while people can be members of several clubs and can exist without being part of a club.

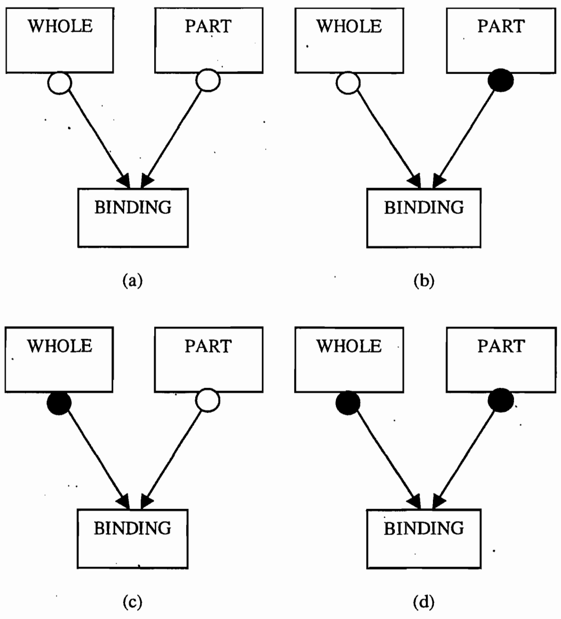


Figure 4. Separable and shareable parts

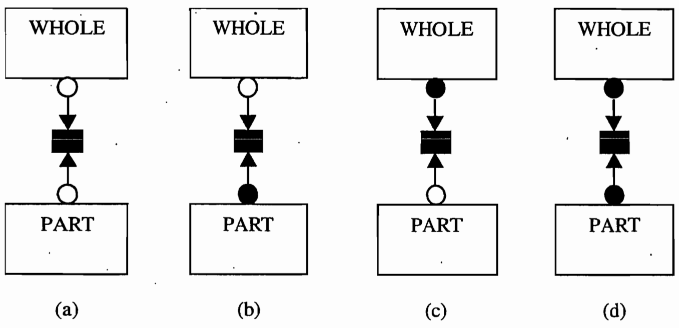


Figure 5. Iconised representation of separable and shareable parts

3.1.2. Unshareable parts

Let us consider stock management in a pharmacy. When a product is (nearly) out of stock, an order line is created for this product. Afterwards, order lines are grouped into orders and send to the appropriate supplier. If however the supplier is not able to respond soon enough (e.g. within two hours) the order line is moved to an order for another supplier. As a result, an order is still an aggregation of order lines. In this case however, the order lines can exist independently of the order and be moved from one order to another. They are unshareable because each order line belongs to only one order at one point in time.

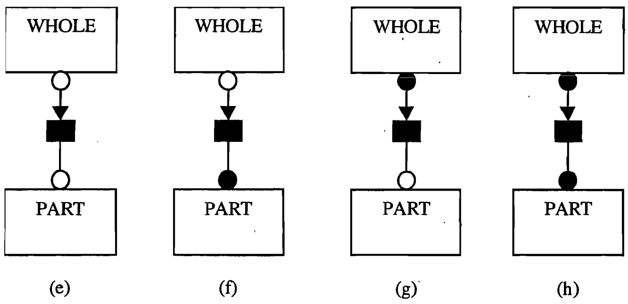


Figure 6. Iconised representation of unshareable parts

Because parts are unshareable, they can be bound to at most one whole at one point in time. As a result, the existence dependency relationship from PART to BINDING has a cardinality of one: a part is involved in at most one binding at a time. The four variants of this type of aggregation are represented in Figure 6. In case (e) the whole has zero to many unshareable parts and parts can exist without being part of a whole. In case (f) the whole has zero to many unshareable parts and parts cannot exist without being part of a whole. In case (g) the whole has one to many unshareable parts and parts can exist without being part of a whole. Finally, in case (h) the whole has one to many unshareable parts and parts cannot exist without being part of a whole. The order and order line example given at the beginning of this paragraph would be an example for case (h): each order needs at least one order line to exist and order lines cannot exist without being part of one order.

3.2. Existence dependent parts

When parts are existence dependent on the whole, their lifetime will always be part of the lifetime of the whole. In addition, as a consequence of the given definition of existence dependency, the parts will also be inseparable from the whole.

3.2.1. Unshareable parts

The simplest situation is when existence dependent parts are in addition not shareable. Parts live and die in the context of the whole. A classic example is an order that is composed of existence dependent order lines. A generic model for this type of aggregation is given in Figure 7. In case (i) the whole can have zero to many parts. In case (j) the whole has at least one constituent part, for example, an order which must contain at least one order line to exist.

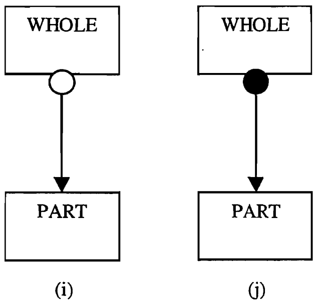


Figure 7. Unshareable existence dependent parts

3.2.2. Shareable parts

In this case, parts are existence dependent of a whole, but can at the same time be shared by other wholes of the same type (homogeneous sharing). Let us for example consider organisational units. On the one hand, we can have a hierarchical relationship, where units are existence dependent of master units.

On the other hand it might be desirable to define temporary units as aggregations of existing units (for example in the context of projects).

A *single* existence dependency relationships between the PART class and the WHOLE class means that each part is existence dependent on exactly one master whole object. When parts can be shared, they can be referred to by other objects than this "master" whole. Hence, the relationship with the sharing whole must be modelled as an additional relationship between the whole and the part. If this additional relationship is not an existence dependent one (such as in the example given above), this is modelled by an object type BINDING that captures the period that a part is shared by a whole different from the master whole. Four variants of this type of aggregation are shown in Figure 8. Depending on the cardinalities of the different existence dependencies, this type of aggregation has many other variants.

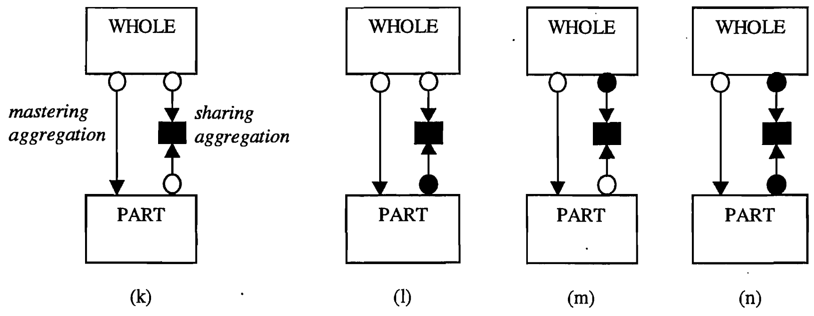


Figure 8. Shareable existence dependent parts

In fact this situation combines two types of aggregation in one schema: one aggregation relationship expressing existence dependency and another aggregation relationship which is non-existence dependent. This type of aggregation will not be considered any further as it can be decomposed in two relationships that can be considered separately.

Notice that apart from its relationship (of any kind) to the whole, a part can at any time be involved in a relationship (meronymic or not) with other objects. For example, order lines are components of an order, but are at the same time existence dependent on the product they refer to.

3.3. Non-existence dependent but inseparable parts

Being inseparable does not necessarily means that the parts are existence dependent as well. Imagine for example the case of a paper and a journal issue (example adapted from [KOL 97]). A paper is not existence dependent on a journal issue nor the other way round. However, once a paper has been accepted and published in a journal issue, it has become a part of this issue and is in addition inseparable from the issue and cannot be shared by other issues. Pictures are an example of non-existence dependent parts that are inseparable from the magazine or newspaper in which they have been published, but that can be shared. These kinds of aggregation are represented as non-existence dependent inseparable aggregations (see Figure 9). The inseparability must be enforced by defining the behavioural aspects in the proper way. This aspect of aggregation is called changeability in [KIL 94]. No changeability means that aggregations can be "frozen": once parts are associated with a whole, this association cannot be changed any more.

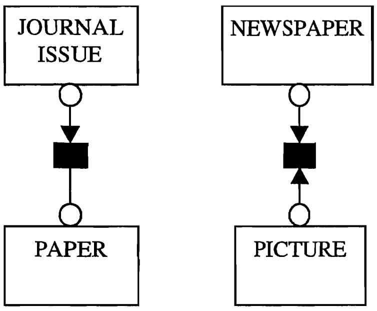


Figure 9. Examples of non-existence dependent inseparable parts

4. BEHAVIOURAL ASPECTS OF EXISTENCE DEPENDENCY

4.1. Modelling behaviour with events

Events are a fundamental part of the structure of experience [COO 94]. Events are atomic units of action: they represent things that happen in the world. Without events nothing would happen: they are the way information and objects come into existence (creating events), the way information and objects are modified (modifying events) and disappear from our universe of discourse (ending events). Events are not attached to a single object class. One event can affect more than one object. In the library for example, returning a book is modelled with an event *return* that affects both a loan (which will be ended by this event) and a copy (which state will be modified by this event). As a result, the behavioural part of the object model can be modelled by means of an object-event table. In the object-event table, there is one column for each object type and one row for each identified event type. A row-column intersection is marked with a 'C' when the event creates the object, with an 'M' when it modifies the state of the object and with an 'E' when it ends the life of the object. A marked entry in a column means that the object class has to be equipped with a method to implement the effect of the event on the object. A possible object-event table for the library example is given in Table 2.

	COPY	MEMBER	LOAN
borrow	M	M	C
renew	M	M	M
return	M	M	E
acquire	C		
sell	E		
lose	E	M	E
enter		C	
leave		E	

Table 2. Object-event table for the library

In this way the object-event table identifies the methods that have to be included in the class definition of object types. According to the object-event table in Table 2, the class COPY will have methods for borrow, renew, return, acquire, sell and lose; the class MEMBER will have methods for borrow, renew, return, lose, enter and leave; and the class LOAN will have a method for borrow, renew, return and for lose.

Obviously, existence dependency has a number of implications on the lifetimes of object types. In [SNO 98, SNO 99] the implications of existence dependency on the object-event table are studied in detail. The result are a number of modelling constraints that apply on the object-event table in order to be consistent with the existence dependency relationships that are defined between the object types in the object-relationship diagram. For modelling aggregation, two of these rules have to be known.

The first rule is that a master object type is always directly or indirectly affected by all event types that affect one of its dependent object types. This can be explained by the fact that the state of a master object is among other factors also determined by the state of its dependent objects. Events that affect dependent objects and hereby change the state of that dependent object thus also change the state of the master object, at least indirectly. Hence, when an existence dependent object is involved in an event, its master objects are automatically involved in this event as well. For example, a state change of a loan, e.g. because of the return of the copy, automatically implies a state change of the related copy and member: the copy is back on shelf and the member has one copy less in loan. By applying this rule, all possible places for information gathering and constraint definition are identified. For example, the *borrow* method of the class MEMBER is the right place to update the number of copies a member has in loan and to check a rule such as 'a member can have at most 5 copies in loan at the same time'. The *borrow* method of the class COPY is the right place to count the number of times a copy has been borrowed. Notice that this is a rule that applies to analysis or conceptual models only. At implementation time, methods that are empty because no relevant business rule was identified, can be removed to increase efficiency. This rule is called the **propagation rule** and states that if an existence dependent object type is involved in an event type, which is marked by a C, M or E entry in the object-event table, the master object type is involved in this event type as well, which should be marked by a C, M or E in the object-event table. As a result, the set of events that affect an existence dependent object type is a subset of the set of events that affect the master object type. In the given example, because LOAN is existence dependent of COPY and MEMBER (Figure 3), the set of events that affect LOAN is a subset of the set of events that affect COPY and of the set of events that affect MEMBER (Table 2).

The next consideration is that the life cycle of a dependent object and its master object are strongly interrelated. Indeed, a dependent object type cannot be created before its master exists nor can it exist after its master has been ended. Creating an existence dependent object means that either the master is created at the same time (e.g. creating the first order line creates the order) or that the master object type already exists (e.g. opening an account for an existing customer). In the latter case, the creation of a dependent object type modifies the state of the master. Since the set of events that affect the dependent object type is a subset of the set of events that affect the master object type, this means that the set of creating event types of the dependent object type is a subset of the creating and modifying event types of the master. Modifying a dependent object type always modifies the state of the master. Finally, ending a dependent object type also modifies the state of the master. If the last dependent object type is ended, then the master can be ended at the same time or later. We call these constraints the **type of involvement rule**.

Type of involvement rule

If in the column of an existence dependent object type a row contains a 'C' then on the same row a 'C' or 'M' must appear in the column of each of its master object types.

If in the column of an existence dependent object type a row contains an 'M' then on the same row an 'M' must appear in the column of each of its master object types.

If in the column of an existence dependent object type a row contains an 'E' then on the same row an 'E' or 'M' must appear in the column of each of its master object types.

Notice that the type of involvement rule subsumes the propagation rule. A more thorough argumentation for these two rules and the role they play in checking the coherence between the structural and behavioural part of a conceptual model can be found in [SNO 99] [SNO 98].

The events that are included in the object-event table have to be atomic units of actions that occur at one point in time and that are not decomposable. They do however not have to keep the set of objects in a consistent state. Some constraints in the structural and behavioural model of objects imply the grouping of these atomic events into units of consistency. These groups of events are called consistent events. In contrast with atomic events, they do keep the set of objects in a consistent state: before and after the occurrence of a consistent event all specified rules are satisfied. Consistent events for aggregations will be discussed in the next section.

4.2. Object Interaction

A major advantage of the object-event table is that it does not assign the responsibility for an event to a particular object. The object-event table assumes that objects participate simultaneously to an event. This can be achieved by assuming a broadcasting mechanism that notifies the participating objects of the occurrence of an event.

This means that when an event occurs and is broadcast, all corresponding methods in the involved objects will be executed simultaneously provided each involved object is in a state where this event is acceptable. If one of the objects is in a state where the event cannot be accepted, the event is rejected by the system. This way of communication is similar to communication as defined in the process algebras CSP [HOA 85] and ACP [BAE 86] and has been formalised in [DED 95], [SNO 99]. Message passing is more similar to the CCS process algebra [MIL 80]. There exist various mechanisms for the implementation of such synchronous execution of methods. For the purpose of analysing the behaviour of aggregations, we will assume that there is an event handling mechanism that filters the incoming events by checking all the constraints this event must satisfy. If all constraints are satisfied, the event is broadcasted to the participating objects; if not it is rejected. In either case the invoking class is notified accordingly of the rejection, acceptation, and successful or unsuccessful execution of the event. This concept is exemplified in Figure 10 for the event types *borrow, renew* and *return* for the library example of Table 2. For each type of business event, the event handling layer contains one class that is responsible for handling events of that type. This class will first check the validity of the event and, if appropriate, broadcast the event to all involved objects by means of the method 'broadcast'.

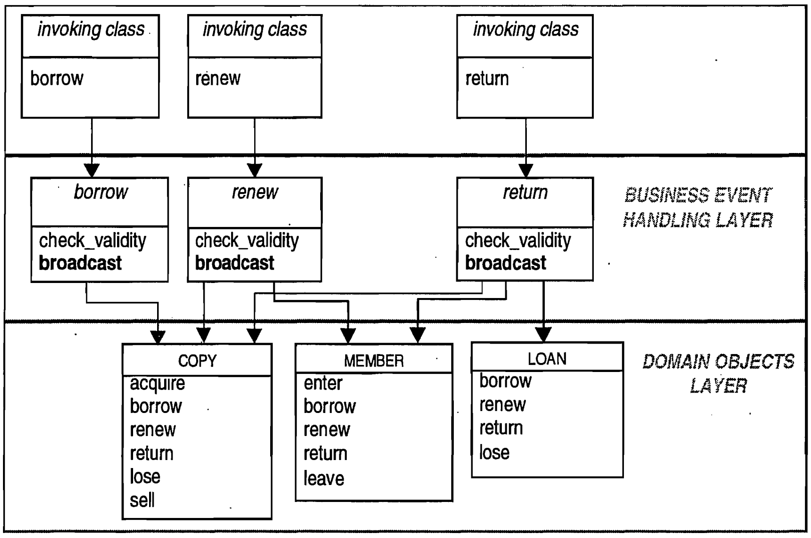


Figure 10. Event handling

In a conventional object-oriented approach, object interaction is achieved by having objects send messages to each other. This is documented by means of collaboration diagrams. Because of the absence of the broadcasting paradigm, events must be routed through the system in such a way that all concerned objects are notified of the event. As there is no generally accepted schema, the routing schema must be designed for each type of event individually. An additional problem is the identification of the object where the routing will start. In most examples given in object-oriented analysis textbooks, the business events are initially triggered by some information system event.

For example, in a library system, the *renew* business event is triggered by the counter application for the library clerk. Such interactions can be represented by including information system objects such as user interface objects in the collaboration diagram. From a conceptual modelling perspective, we would prefer object interaction to be independent from information system services. For example, the business event *renew* can also be triggered by other information system services such as a web interface for library members.

Notice that the concept of the object-event table allows to model interaction at a much higher level of abstraction than is the case with message passing. Moreover, the interaction pattern is independent of the number of objects involved in an event. At conceptual modelling level, we should not burden ourselves with event notification schemas. How exactly objects are notified of the occurrence of an event is a matter of implementation. When using object-oriented technology this will be done with messages, but when using other technologies, both traditional and modern (e.g. distributed component technologies), (remote) procedure calls can do as well.

5. BEHAVIOURAL ASPECTS OF AGGREGATION RELATIONSHIPS

In the previous section, events were described as being atomic, non-decomposable units of action. This definition does not require that the object base remains in a consistent state after the occurrence of an atomic event: an atomic event is a unit of action, but not always a unit of consistency. Indeed, some constraints imply the grouping of events in order to keep the set of existing objects in a consistent state. For example, if the relationship between a whole and its existence dependent parts is mandatory for the whole, this means that when an occurrence of the whole is created an occurrence of the part must be created at the same time. These groups of events (for example, a *cr_whole* event combined with a *cr_part* event) are called ***consistent events***. They are composed of atomic events. In contrast with the latter, they always keep the set of existing objects in a consistent state: before and after the occurrence of a ***consistent event***, all specified rules are satisfied. The consistent events are the only events that are visible to the classes that invoke services from the aggregation classes. They are put in a separate layer on top of the layer with the atomic events. In this section, we describe such consistent groups of events that are required by the structural constraints of the aggregation or that are required to model the propagation of behaviour from the whole to the parts.

Only consistent events are visible outside the scope of the aggregation. Methods for the atomic events are kept local to the aggregation classes. The names of the consistent events will be preceded by 'C_' to discern them from the atomic events. The definition of some consistent events can be inferred from the cardinality constraints. Propagation of behaviour and deletion from the whole to the parts are defined as additional characteristics of the aggregation by defining the appropriate consistent events.

For conceptual modelling purposes, the event handling schema is the best suited to document the consistent events since it avoids the design of a message passing schema. Each consistent event will be documented by a collaboration diagram that shows the different elements in the different layers. To illustrate that it is also possible to have events handled directly by the domain objects rather than by an event handling layer, each consistent event will also be accompanied by an interaction diagram that gives a possible message passing scenario. In these interaction schemas the invoking class for the consistent events will be omitted since they can be invoked by any class. Notice that the given message passing scenario is only one out of many more possibilities.

5.1. Separable parts

The atomic events for this type of aggregation are shown in the generic object-event table in Table 3. The table is the same for shareable and unshareable parts.

Due to the propagation of event-participation from the existence dependent object type to the master object types, WHOLE and PART acquire the methods *cr_binding*, *mod_binding* and *end_binding*. The *cr_binding* event models what happens when a part is added to the whole. Apart from creating a binding, this event can change some resultant properties in the whole (to be specified in the whole.cr_binding method) and some properties of the part (to be specified in the part.cr_binding method). Similarly, the *mod_binding* and *end_binding* events can have an effect on resultant properties of the whole (to be specified in the whole.mod_binding end whole.end_binding methods) and some properties of the part (to be specified in the part.mod_binding and part.end_binding methods). The *mod_whole* and *mod_part* event types are the defaults for the modification of the emergent properties of the whole and the part.

	WHOLE	PART	BINDING
cr_whole	C		
mod_whole	M		
end_whole	E		
cr_part		C	
mod_part		M	
end_part		E	
cr_binding	M	M	C
mod_binding	M	M	M
end_binding	M	M	E

Table 3. OET for non-existence dependent parts

5.1.1. Deletion

The concept of existence dependency implies some general constraints on deletion: a master object type cannot be deleted as long as there exist existence dependent objects for this master. As a result, in this type of aggregation, the whole cannot be deleted as long as there are parts attached to this whole. Similarly, a part cannot be deleted as long as it is attached to a whole. These constraints can be implemented as a restriction to the ending events by means of preconditions or they can be implemented as cascading deletes. The choice between these two options must be determined by the analyst. For example, one could choose to implement the ending of a part with a cascading delete. This means that the *C_end_part* event will trigger the *end_binding* event for each of the bindings of the concerned part. Finally the *end_part* event is trig-

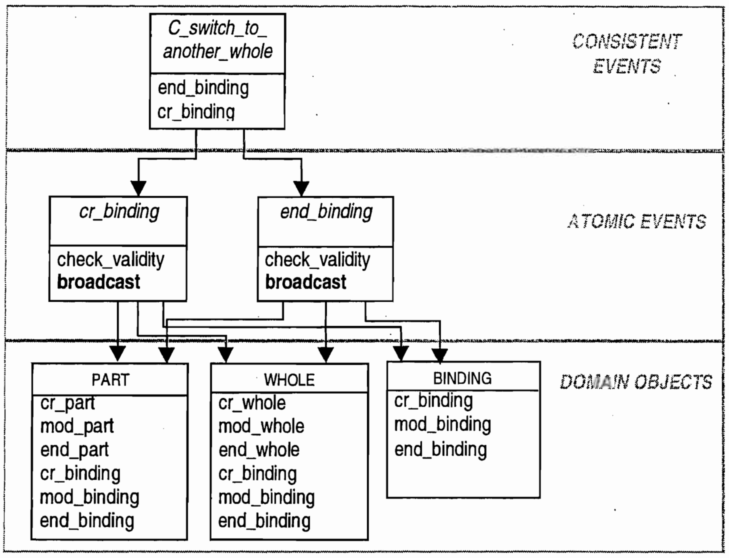


Figure 14. Switching a separable unshareable part from one whole to another

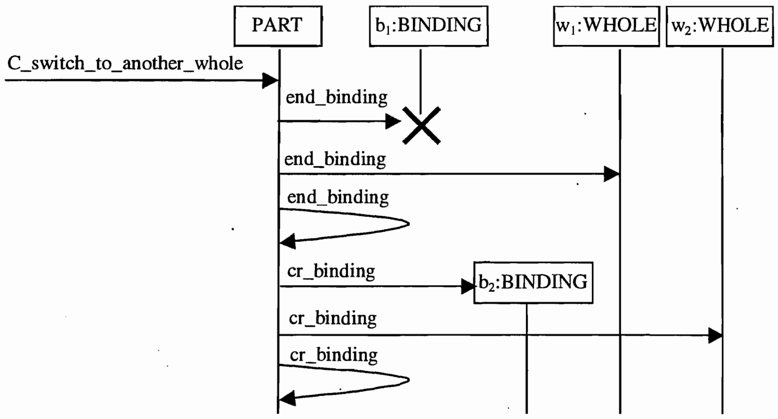


Figure 15. Interaction schema for switching a separable unshareable part from one whole to another.

5.1.2. Modification

The propagation of modifying events of the whole towards the parts (Figure 16 and Figure 17) follows a pattern similar to the cascading deletion of the whole.

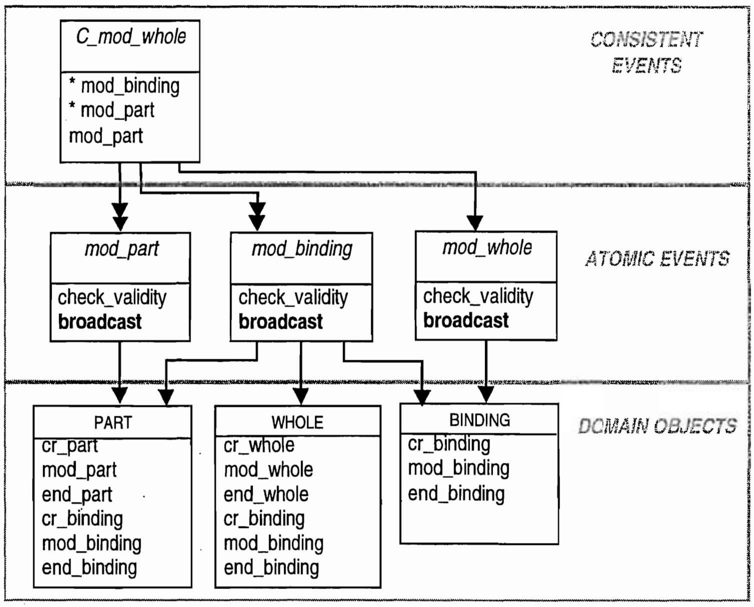


Figure 16. Propagating modifications from the whole to its parts

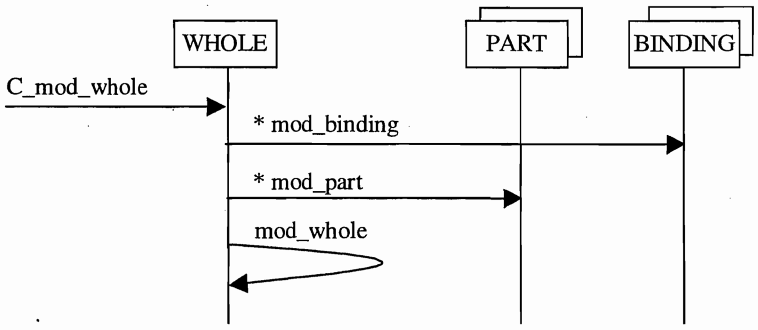


Figure 17. Interaction schema for propagating modifications from the whole to its parts

5.1.3. Creation

When the relationship from the whole to the part is mandatory, this means that when a whole is created, it must immediately be bound to an existing part or, in case no suitable part exists, a part must be created for this whole. This event handling scenario is shown in Figure 18 and a possible interaction schema is given in Figure 19. The dashed arrows mean that the event is called under certain conditions only.

Similarly, when a part must be attached to a whole at any time, the creation of a part must be accompanied by the creation of the appropriate binding with an existing whole. If the whole does not exist, creation of the part must be either prohibited of the whole must be created at that moment. The latter event handling scenario is also represented in Figure 18. An example of a message passing schema is given in Figure 20.

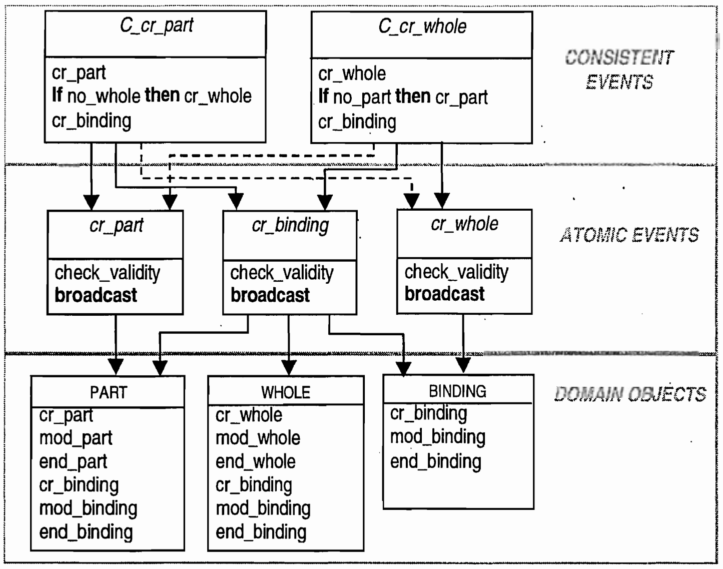


Figure 18. Creation of a whole with its mandatory part; Creation of a part mandatorily part of a whole

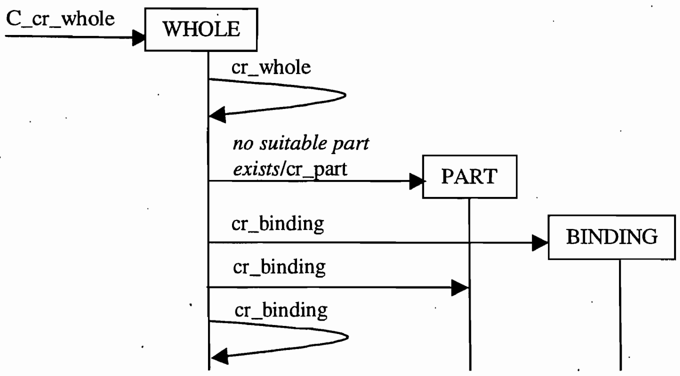


Figure 19. Interaction schema for the creation of a whole with its mandatory part

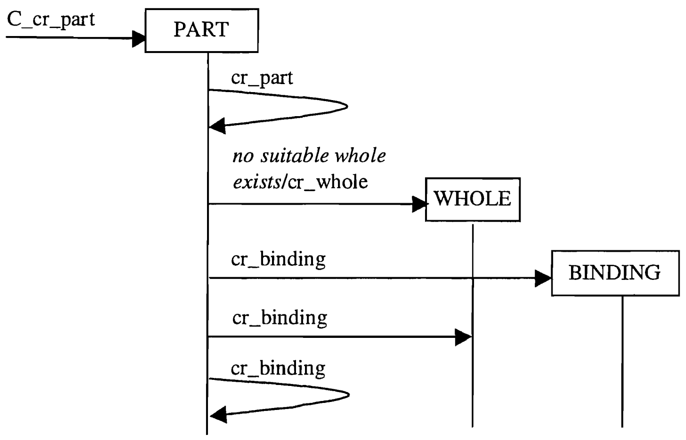


Figure 20. Interaction schema for the creation of a part mandatorily part of a whole

5.2. Existence dependent parts

In case of existence dependent parts, the generic object-event table for the aggregation is as in Table 4.

	WHOLE	PART
cr_whole	C	
mod_whole	M	
end_whole	E	
cr_part	M	C
mod_part	M	M
end_part	M	E

Table 4. OET for existence dependent parts

5.2.1. Deletion

Again, the deletion of the whole can be implemented as a restricted delete, whereby deletion is prohibited as long as parts exist, or as a cascading the delete. The sequence chart for the cascading delete is given in Figure 21.

Parts can be deleted any time, except when the relationship is mandatory. In that case, when the last part is deleted, the whole must disappear as well (Figure 21, Figure 22, and Figure 23).

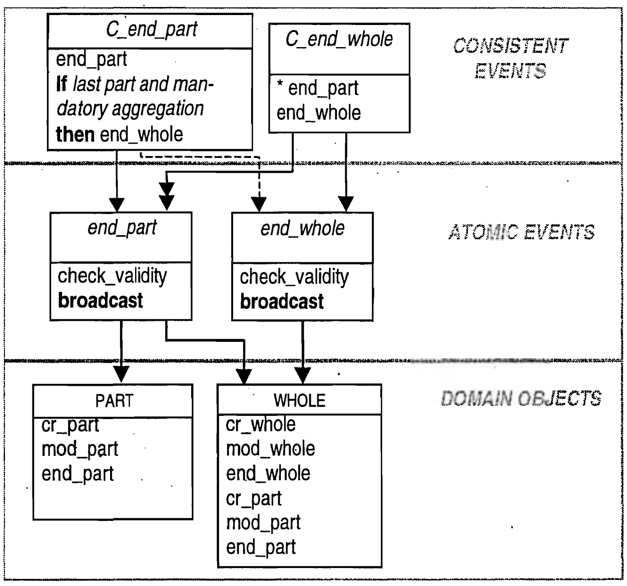


Figure 21. Cascading end_whole and cascading end_part

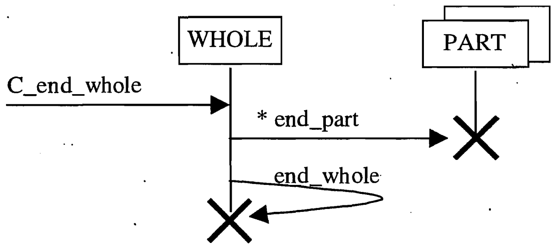


Figure 22. Interaction schema for a cascading end_whole

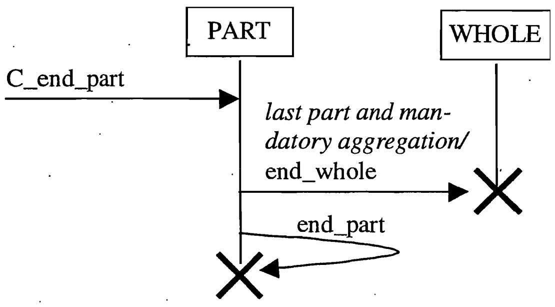


Figure 23. Interaction schema for a cascading end_part

5.2.2. Modification

The propagation of modifications from the whole to its parts is similar to the implementation of a cascading delete (Figure 24 and Figure 25).

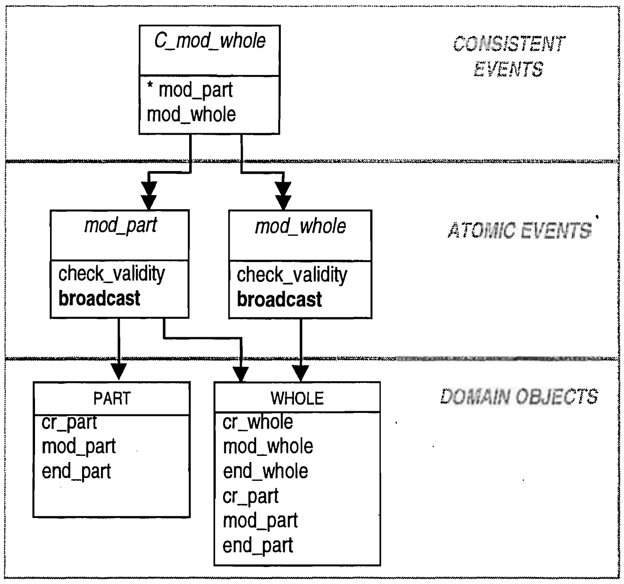


Figure 24. Propagation of modifications to the existence dependent parts

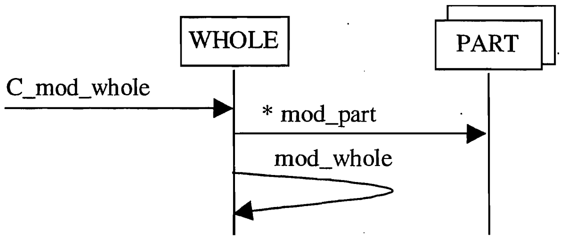


Figure 25. Interaction schema for the propagation of modifications to the existence dependent parts

5.2.3. Creation

The creation of a whole will also create a part if having at least one part is mandatory for the whole. This is represented in Figure 26 and Figure 27. The creation of an existence dependent part is always consistent.

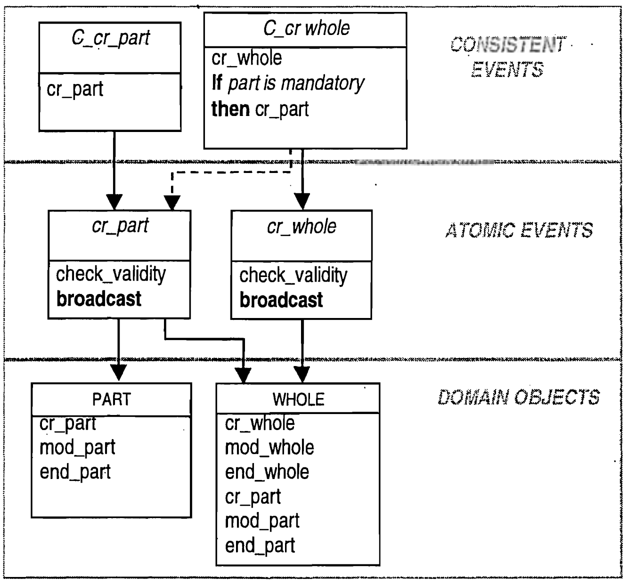


Figure 26. Creation of a whole with mandatory existence dependent part

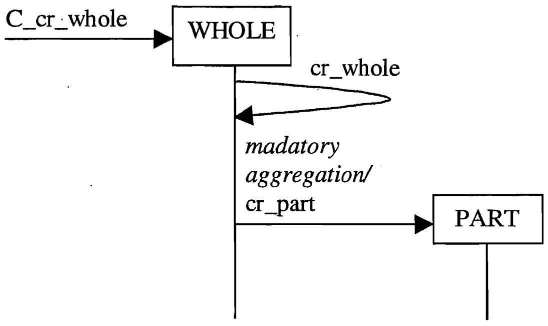


Figure 27. Interaction schema for the creation of a whole with mandatory existence dependent part

5.3. Non-existence dependent but inseparable parts

The object-event table for this kind of aggregations is the same as for separable part (see Table 3). Most creation, updating end deletion rules remain the same. The main difference is that in this case the *end_binding* event cannot be invoked outside the scope of a *C_end_part* or *C_end_whole* event. Indeed, since the part is inseparable from the whole, the termination of the whole-part relationship between a part and its whole can only occur in the context of the termination of the part and/or of the whole. As a result, there is no *C_end_binding* event and the atomic *end_binding* event is always invoked by the *C_end_part* or the *C_end_whole* events.

6. Conclusions

The discussion of the structural aspects of the whole-part relationship in section 3 has only concentrated on three characteristics: separability, existence dependency and shareability. There are of course many more possible characteristics of aggregation.

6.1. Primary Characteristics.

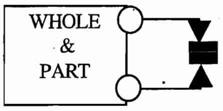
Both in [HEN 99a] and [SAK 98], emergent properties, resultant properties, irreflexivity and anti-symmetry are proposed as primary characteristics of aggregation. In the approach proposed in this paper, the whole can have both emergent and resultant properties. Irreflexivity and acyclicity are part of the formal definition of existence dependency [SNO 98]: in an existence dependency graph an object type is never existence dependent of itself and an existence dependency graph must be acyclic[4] Hence, when parts are existence

[4] 1a) The life span of an object is always embedded in itself. As a result, one could say that an object is existence dependent of itself. However, in the context of object-oriented analysis, it is the relation between *different* objects that is of interest. In this sense, saying that an object is existence dependent of itself does not provide us with additional information.

1b) Assume that an object *type* P would be existence dependent of itself, whereby each occurrence of class P depends on the existence of another occurrence of the same class P. It would then be impossible to create occurrences of class P. Indeed, as the life of the existence dependent object cannot start before the life of its parent, creating the existence dependent object requires the existence of a parent object. But this parent is in turn existence dependent of another object of the same class, which should already exist before the parent is created. As a result, allowing an object type to be existence dependent of itself creates a problem of circular prerequisites. Hence we define that an object type cannot be existence dependent on itself.

dependent on the whole, the aggregation relationship is irreflexive and anti-symmetric. When parts are not existence dependent on the whole, the binding establishes a whole-part relation between objects from two different classes and in a particular direction. As a result irreflexivity and anti-symmetry are respected. Transitivity is not necessarily true.

When the whole and its parts belong to the same object class, we have a homeomerous (and recursive) aggregation. Because an object type cannot be existence dependent on itself, this type of whole-part relationship is always modelled using a binding object type. In this case, irreflexivity and anti-symmetry must be enforced by setting pre-conditions for the *cr_binding* event as shown in Figure 28.



```
Class WHOLE&PART
    binding_as_whole: set(BINDING)
    binding_as_part: set(BINDING)
...
end class WHOLE&PART

Class BINDING
    whole: WHOLE&PART
    part: WHOLE&PART

cr_binding (whole, part: WHOLE&PART) is
precondition:
    irrefexivity : whole ≠ part
    anti-symmetry: not exists b in part.binding_as_whole
                 where b.part = whole
...
end class BINDING
```

Figure 28. Specification of a homeomerous aggregation with shareable parts

6.2. Secondary characteristics

Encapsulation can be enforced by defining the proper consistent events. *Life-time binding* is implied by the (non)-existence dependent nature of the whole part relationship. For existence dependent parts only the lifetime bindings given in Figure 1 are possible. The choice between these possibilities is further

2) Similarly, allowing cycles in the existence dependency graph leads to circular prerequisites as well. Hence we require the existence dependency graph to be acyclic.

determined by the consistent events. For non-existence dependent parts, all we can say is that the existence of a binding between a part and a whole implies that there is some overlap between their lifetime. As the existence dependency relation is transitive the existence dependent nature of parts is transitive as well. This does however not mean that the *transitive* interpretation of the aggregation is meaningful. When parts are not existence dependent on the whole, transitivity is not necessarily true. The *configurational* nature of an aggregation cannot be specified in terms of existence dependency: it must be specified by means of attributes, methods, pre- and post- conditions, and invariants. *Immutability* is a characteristic from the whole that says that parts cannot be removed nor been replaced by an equivalent one without destroying the whole. Existence dependency is a characteristic of the parts that implies that parts are inseparable from the whole. Existence dependency does not necessarily also imply immutability. Even when a part is existence dependent and mandatory, in a *mutable* aggregation it can be exchanged for an equivalent object (that is, from the same class). The original part will die because it cannot be separated from the whole and is immediately replaced by a new existence dependent part. Immutability must be enforced by a suitable definition of the behavioural aspects of the aggregation, that is, by limiting the consistent events to those that leave the aggregation unchanged (in a similar way as discussed in paragraph 5.3).

This discussion of secondary characteristics of aggregation is not complete. However, it largely demonstrates that the proposed concepts can be used to characterise many flavours of whole-part relationships. The simplicity of the concepts used to characterise aggregation, namely existence dependency, and atomic and consistent events, makes them easy to use and easy to define in a precise and formal way. The concept of existence dependency is formalised in [SNO 98] where it is used as core concept to define arbitrary associations between objects. The concepts of atomic and consistent events are defined in detail in [DED 95, SNO 99], together with a CSP-like process algebra that formalises the concepts of object life cycle and object interaction. More importantly, the concept of existence dependency also allows checking semantic integrity between the structural and behavioural aspects of object types.

When analysts model systems, they have to capture the relevant aspects of the real world into information systems models. The difficulty of capturing sometimes very complex reality with only a few modelling concepts has lead to the definition of new modelling concepts with richer semantics. Unfortunately, although their definition seems intuitively clear, they are often very poorly defined. As a consequence, their interpretation varies from person to person. Hence, conceptual models that use such concepts are ambiguous and unprecise. In this paper we have used a set of very simple concepts to define the complex principle of aggregation. We believe that it is possible to model everything, using only these concepts and the concept of generalisation/specialisation. By representing groups of elements with a single icon, it is possible to hide the increased size of the resulting information models. The main advantage of this approach is that the semantics of the richer concepts, aggregation in this paper, are better defined because they can now be inferred from the formal definition of the core concepts.

In [BRU 98] Brunet argues that aggregation is an unnecessary concept because it can be expressed with more fundamental concepts such as composition and inheritance (the used notion of composition is very similar to the notion of existence dependent parts). In this paper we have proven that many flavours of aggregation can indeed be expressed with existence dependency only. Without saying that aggregation is an unnecessary concept, we would at least like to say that it is *not* a primary concept. It rather is a higher level concept the semantics of which can be specified by means of core (or primary) concepts such as existence dependency.

7. REFERENCES

[BAE 86] BAETEN, J.C.M., *Procesalgebra: een formalisme voor parallelle, communicerende processen.* Kluwer programmatuurkunde, Kluwer Deventer, 1986

[BRU 98] BRUNET J., "An Enhanced Definition of Composition and its use for Abstraction", in C. Rolland, G. Grosz (eds.), *OOIS'98*, Springer Verlag London, pp. 11-19

[COO 94] COOK S., DANIELS J., *Designing object systems: object-oriented modelling with Syntropy*, Prentice Hall, 1994

[DED 95] DEDENE G., SNOECK M., "Formal deadlock elimination in an object oriented conceptual schema", *Data and Knowledge Engineering*, 15 (1-30), 1995.

[DOG 90] DOGAC A., OZKARAHAN E., CHEN P., An integrity system for a relational database architecture, in F. H Lochovsky, *Entity Relationship Approach to Database Design and Querying*, Proc. of the Eight International Conference on Entity-Relationship Approach, Toronto, Canada, 18-20 October, 1989,North-Holland, 1990, pp. 287 - 301

[HEN 97] HENDERSON-SELLERS B., "OPEN relationships - compositions and containments", *JOOP/ROAD*, 10(7), pp. 51-55,72

[HEN 99a] HENDERSON-SELLERS B., BARBIER F., "What Is This Thing Called Aggregation ?", *Proceedings of the 29th International Conference on Technology of Object-Oriented Language-Europe*, 7-10 June, Nancy, France.

[HEN 99b] HENDERSON-SELLERS B., BARBIER F., "A Survey of th UML's Aggregation and Composition Relationships", *L'objet*, 5(3-4), 1999, pp.339-366.

[HOA 85] HOARE, C. A. R., *Communicating Sequential Processes*. Prentice-Hall, 1985

[KIL 94] KILOV H., ROSS J., *Information Modeling, An Object-Oriented Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1994

[KOL 97] Kolp M., Pirotte, A, "An Aggregation Model and its C++ Implementation", In M.E. Orlowska and R. Zicari, editors, Proc. of the 4th Int. Conf. on Object-Oriented Information Systems, OOIS'97, pages 211-224, Brisbane, Australia, November 1997

[MIL 80] MILNER R., *A calculus of communicating systems*. Springer Berlin, Lecture Notes in Computer Science, 1980

[ODE 94] ODELL J., "Six different kinds of composition", *JOOP* 6(8), 1994, pp.10-15

[PUT 88] PUT F., "Introducing dynamic and temporal aspects in a conceptual (database) schema", doctoral dissertation, Faculteit der Economische en Toegepaste Economische Wetenschappen, K.U.Leuven, 1988, 415 pp.

[SAK 98] SAKSENA M., FRANCE R., LARONDO-PETRIE M., "A characterisation of aggregation", in C. Rolland, G. Grosz (eds.), *OOIS'98*, Springer Verlag London, pp. 11-19

[SNO 98] SNOECK M., DEDENE G., "Existence Dependency: The key to semantic integrity between structural and behavioural aspects of object types", *IEEE Transactions on Software Engineering* , Vol. 24, No. 24, April 1998, pp.233-251

[SNO 99] SNOECK M., DEDENE G., VERHELST M., DEPUYDT A.-M., *Object-oriented Enterprise Modelling with MERODE*, Leuven University Press, 1999