Existence Dependency: The key to semantic integrity between structural and behavioural aspects of object types

Monique Snoeck* and Guido Dedene

Dept. of Aplied Econ. Sciences- KULeuven
Naamsestraat 69, 3000 Leuven - Belgium
email: monique.snoeck / guido.dedene@econ.kuleuven.ac.be

This paper has been published in IEEE Transactions on Software Engineering (Vol. 24, NO. 4, April 1998)[◊]

ABSTRACT

In object-oriented conceptual modelling, the Generalisation/Specialisation hierarchy and the Whole/Part relationship are prevalent classification schemes for object types. This paper presents an object-oriented conceptual model where, in the end, object types are classified according to two relationships only: existence dependency and generalisation/specialisation. Existence dependency captures some of the interesting semantics that are usually associated with the concept of aggregation (also called composition or Part Of relation), but in contrast with the latter concept, the semantics of existence dependency are very precise and its use clear cut. The key advantage of classifying object types according to existence dependency are the simplicity of the concept, its absolute unambiguity and the fact that it enables to check conceptual schemes for *semantic* integrity and consistency.

We will first define the notion of existence dependency and claim that it is always possible to classify objects according to this relationship, thus removing the necessity for the Part-Of-relation and other kinds of associations between object types. The second claim of this paper is that existence dependency is the key to semantic integrity checking to a level unknown to current object-oriented analysis methods. In other words: existence dependency allows to track and solve inconsistencies in an object-oriented conceptual schema.

Index Terms: Software Engineering, conceptual model, object oriented analysis, existence dependency, aggregation, composition, quality, consistency checking

* At the time of publication I was working at Dépt. d'Informatique -ULB, Bd. Du Triomphe - CP 212, 1050 Brussels - Belgium. Currently I am working at the KULeuven.

1. INTRODUCTION

One of the main tasks in the object-oriented analysis process is the elaboration of the “real world” model. This model describes the Universe of Discourse at the conceptual level, without taking functional requirements in account yet. This model is also called enterprise model [8] or entity object model [16]. Typical kinds of components are classes, attributes, methods and relationships between classes. These relationships or associations organise object types (or classes) into classification schemes. Most developers will agree that the “A Kind Of” (or Generalisation/Specialisation) lattice and the “A Part Of” (or aggregation) lattice are two primary ways of organising objects. Nearly every object-oriented analysis method has special notations to denote these two association lattices [3, 5, 6, 10, 16, 21, 22]. There are of course many other kinds of relationships that can be defined and these are in general captured under the common denominator “associations”. The concepts of Generalisation/Specialisation and Part-Of both reflect very natural and intuitive classification principles. These concepts are considered crucial to object-oriented conceptual modelling because of their ability to reduce the complexity of conceptual schemes. However, as pointed out in [7], one of the major problems with the use of the Part-of relation is that its semantics are insufficiently defined. Many variants of the concept of aggregation exist and definitions are mostly subject to different interpretations. The classification principle *existence dependency,* presented in this paper, captures the most interesting features of the principle of composition and has the advantage that its semantics are simple and clear.

In addition to the simplicity and total absence of ambiguity, a very important motivation for classifying object types according to existence dependency is the ability to allow for quality control at a very high level. In *object-oriented* conceptual modelling, both the structure and the behaviour of object types have to be modelled. Usually different techniques are used to capture both kinds of aspects. For example, a large number of object-oriented analysis (OOA) methods use an Extended Entity Relationship-like technique together with the concepts of Generalisation/Specialisation and “Part Of” for specifying static aspects. Finite State Machines and Event Trace Diagrams are used for capturing dynamic aspects. Some of these structure and behaviour modelling techniques have overlapping semantics. This means that the same aspects may be modelled several times in different schemes. For example, the Generalisation/Specialisation-lattice should have an influence on how behaviour should be modelled. If we assume that the technique of Finite State Machines is used for behaviour modelling, then these are examples of relevant questions:

- Does a specialisation type inherit the state machine of the generalisation type?
- Can it refine this state machine by adding, removing or redefining states, transitions or events ?
- Can it restrict the behaviour of the generalisation type or extend it or both ?
- Are the events of the specialisation type specialisations of the events of the generalisation type ?
- Can a specialisation type override properties of the generalisation ?

Many current OOA-methods do not answer these questions in a very precise or formal way. For example, in OOSA the life-cycle of a subtype corresponds to *a part* of the life-cycle of its supertype [23]. This definition violates the broadly accepted notion of inheritance where subtypes inherit data *and behaviour* of their supertypes.

It is clear that some kind of consistency checking between subschemes is required to ensure the quality of the conceptual schema. This consistency checking can vary from a simple syntactic correspondence to a full semantic match between subschemes. In [24] it was demonstrated how consistency between the “A Kind Of”-lattice and behaviour modelling can be ensured. In this paper we will demonstrate how the Existence Dependency lattice can serve as a starting point to derive overlapping semantics between static and dynamic schemes in general and to define schema constraints that will ensure consistency.

Classifying object types according to existence dependency thus solves two problems at once. First, it is a better alternative for the sometimes confusing concepts of aggregation and composition. Secondly, it allows for semantic integrity control of object-oriented domain models to a level unknown by current object-oriented analysis methods.

The paper is organised as follows. Section 2 defines the concept of existence dependency and demonstrates that it is always possible to classify object types according to this relationship. Existence dependency is then compared in more detail with the concept of aggregation (Section 2.4). Finally, the paper presents the formal definition of the dynamic aspects of a conceptual model with an explicit existence dependency relation

(Section3) and demonstrates how this relation can be used as a starting point for semantic consistency checking between static and dynamic aspects of object types (Sections 3.3 and 3.6).

Note that this paper presents a conceptual model where object types are classified according to existence dependency only in order to simplify the presentation of the results. In practical situations, object types can be classified according to Generalisation/Specialisation as well. For a formal definition of the latter relationship type and its consequences for modelling behavioural aspects, the reader is referred to [24].

2. EXISTENCE DEPENDENCY

2.1 What is existence dependency ?

The concept of existence dependency 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 it is ended. 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 (ED) relation is a partial ordering on objects and object types which is defined as follows:

Definition 1.

Let P and Q be object types. P is existence dependent of Q (notation: $P \leftarrow Q$) if and only if the life of each occurrence p of type P is embedded in the life of one single and always the same occurrence q of type Q. p is called the *existence dependent object* (P is the existence dependent object type) and is existence dependent of q, called the *parent* (Q is the parent object type).

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

If each object of a class A always refers to minimum one, maximum one and always the same occurrence of class B, then A is existence dependent of B.

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

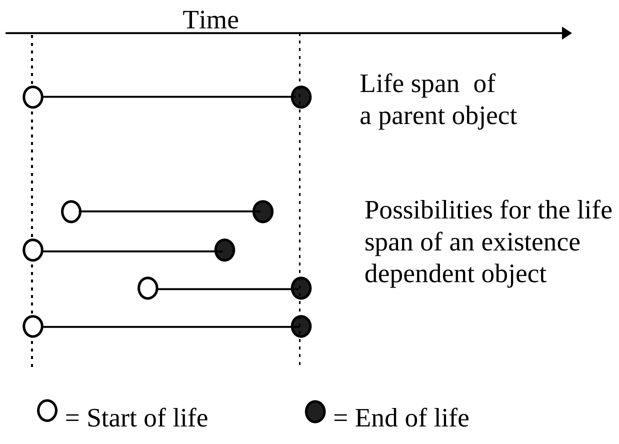


Fig. 1: life span of parent and existence dependent object

Example 1

The life span of a loan of a copy (of a book) is always embedded in the life span of the copy that is on loan. Indeed, we can not have a loan for a copy if the copy doesn’t exist. And the life cycle of the copy cannot end as long as the life cycle 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 of the object type COPY.

The notion of existence dependency is very similar to the concept of a dependency association as described in [17]. However, a major difference is that when an object is dependent from two parent instances, these parent instances need *not* to be of the same type. A loan, for example depends on the existence of a copy instance and of a member instance.

2.2 The Existence Dependency Graph (EDG)

2.2.1 Basic Definition

The existence dependency (ED) relationship is a partial ordering on object types as defined in the previous section. The following definition establishes what we understand by a syntactically correct existence dependency graph.

Definition 2

Let $\mathcal{M}$ be the set of object types in the conceptual schema.

The existence dependency graph (EDG) is a relation that connects object types of $\mathcal{M}$ and satisfies the following restrictions:

1) An object type is never existence dependent of itself
2) The EDG is acyclic.

The two restrictions are motivated as follows:

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 of itself.

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.

Later in this paper, when formulating consistency checking rules, additional arguments that motivate these restrictions will be given.

Informal definitions like the above one have the disadvantage that often different interpretations are possible. For this reason, for each definition a formal equivalent is given in Appendix A.

Graphical representation

A parent object type is placed above the existence dependent object type and the two are connected with a line. Example 2 is represented in Fig. 2.

Example 2

In a library environment, the entity type BOOK captures the common features of similar copies, that is to say, the combination of authors, title, text, publisher, number of pages, and so on. The library can possibly keep zero, one or many physical copies of a single book. Copies are always the physical realisation of a book. As in addition each copy is the realisation of exactly one book for its whole life, the object type COPY is existence dependent of the object type BOOK. MEMBERS can borrow COPIES. But members are not existence dependent of copies, nor are copies existence dependent of members. There is however a relationship between MEMBER and COPY: some copies are borrowed by a member. This relationship is modelled by the object type LOAN, which is existence dependent of the object types that are involved in the relationship, namely COPY and MEMBER. Indeed, a loan always refers to exactly one and always the same member and copy. For this small library example, the set of object types and the existence dependency relation are as follows:

$\mathcal{M}$ = {LOAN, BOOK, MEMBER, COPY}

LOAN ← COPY

LOAN ← MEMBER

COPY ← BOOK

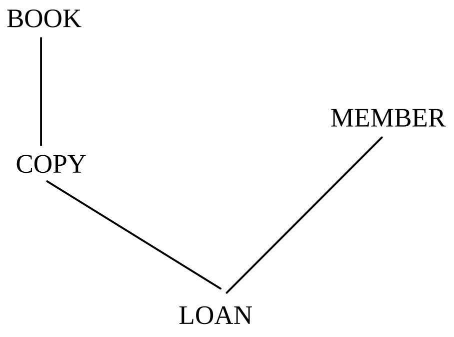


Fig. 2. Existence Dependency Graph for the library

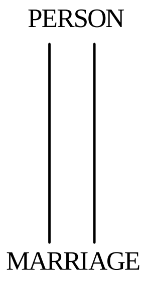


Fig. 3 A marriage depends on two persons

Note that an object type can be existence dependent from the same object type in two different ways. For example, the existence of a marriage[1] depends both on the existence of a husband and on the existence of a wife and thus there will be two lines between MARRIAGE and PERSON (see Fig. 3).

2.2.2 Cardinality of Existence Dependency

The existence dependency graph also defines the cardinality of the existence dependency relationship. This cardinality defines how many occurrences of the existence dependent object type can be dependent of one parent object at one point in time.

Notation

P (1)← Q if P ← Q and an occurrence q of Q can have at most one existence dependent occurrence of P *at one point in time*.

P (n)← Q if P ← Q and an occurrence q of Q can have more than one existence dependent occurrence of P *at one point in time*[2].

Graphical representation

The cardinalities are written next to the line that connects the parent and the existence dependent object type. The cardinality of one is written as a ‘1’ next to the line representing the existence dependency. A cardinality of Many is written as a ‘M’ next to the line representing the existence dependency. Note that the ‘M’ stands for ‘Many’ and that if an ‘M’ is appearing in several places in a diagram, this does not mean that the cardinalities of all these existence dependencies are equal in number.

Example 3

In the library example, at one point in time a copy can be involved in at most one loan, a member can have several loans going on and a book can have several physical copies. Therefor:

1. Many designers would define marriage as a relationship rather than as an object type. But as will be seen further on, non-existence dependent relationships always give rise to a new object type that is the instantiation of this relationship.
2. Note 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.

LOAN (1)← COPY and LOAN (n)← MEMBER
COPY (n)← BOOK
Fig. 4 shows the Existence dependency graph with cardinalities.

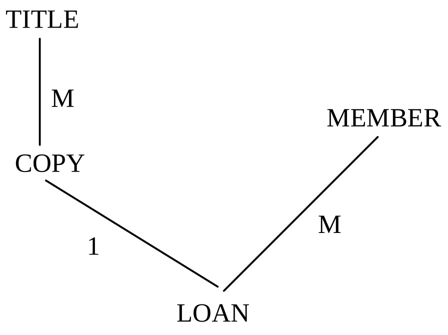


Fig. 4. EDG with cardinalities for the library

Note that the cardinality of the reference from the existence dependent object type to the parent object type is always the same: each existence dependent object always refers to exactly one parent object. As a result, if the existence dependency relation is modelled as a classical binary link between object types, it is a one to many or one to one link with cardinalities and multiplicities as denoted in Fig. 5, where the UML notations are used to denote cardinalities and multiplicities [20].

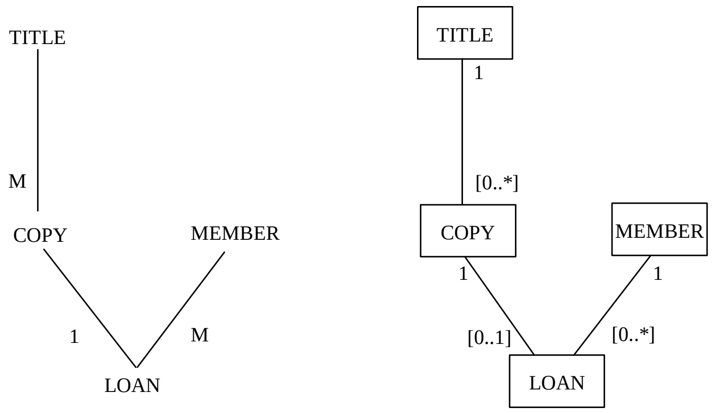


Fig. 5 EDG with equivalent UML diagram

In order to correctly reflect the semantics of existence dependency, the UML diagram needs an additional constraint: it is required that the reference from the existence dependent type to the parent type cannot be re-assigned.

Fig. 6 gives the representation of the existence dependency relationship according to the ER-notation [4]. Existence dependency is equivalent to the concept of a weak relationship that is in addition mandatory for the weak entity type (which is indicated by the black dot).

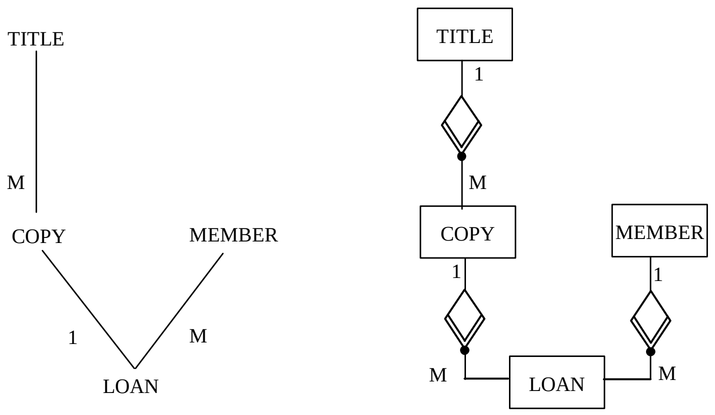


Fig. 6 Existence dependency expressed in the notation of the ER-model

For the life cycles of objects, a multiplicity of *one* implies that the parent object can have only one existence dependent object of a certain type at one point in time. However, there can still be many existence dependent objects of that type consecutively. For example, in the library example, a copy can be involved in at most one loan at any one time, but possibly in many loans consecutively. On the other hand, a member can be involved in many loans at the same time. Fig. 7 gives a graphical representation of the relationship between life cycles of a copy and its related loans and between the life cycle of a member and those of the related loans.

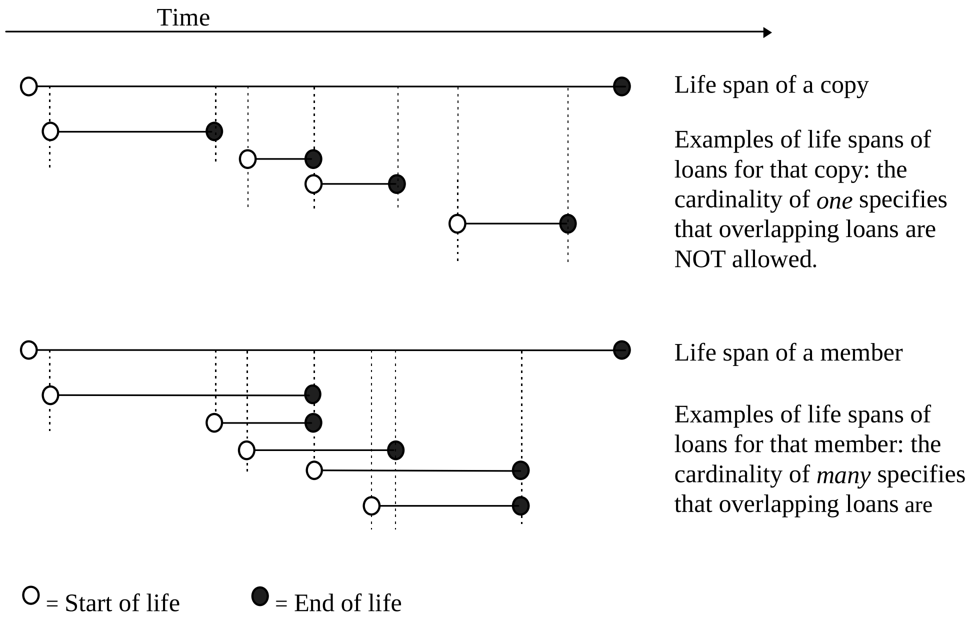

Time
Life span of a copy
Examples of life spans of loans for that copy: the cardinality of *one* specifies that overlapping loans are NOT allowed.
Life span of a member
Examples of life spans of loans for that member: the cardinality of *many* specifies that overlapping loans are
○ = Start of life
● = End of life

Fig. 7. Examples of life spans for a copy, a member and loans

2.3 Existence dependency versus (traditional) relationship types

Most object-oriented analysis methods have an Entity-Relationship like technique for modelling static aspects. In the conceptual model proposed in this paper, it is the existence dependency graph that fulfils this purpose: all object types have to be related according to existence dependency[3]. At first sight it seems not so obvious that organising object types according to existence dependency is always possible. However, by means of a few examples we will demonstrate that this is in fact pretty straightforward.

3. In reality object types can also be classified according to Generalisation/Specialisation. However, as this paper focuses on existence dependency, we assume a domain where there is no need for classification according to the IS-A relationship.

Binary Relationships. Imagine a conceptual model with two related object types. Either the relationship between the object types is existence dependent or it is not. In the first case, one object type is existence dependent of the other. For example, ORDER and ORDERLINE are related to each other by means of an existence dependent relationship with cardinality many: each order can have many existence dependent order lines. If the relationship is not existence dependent, then the relationship should be instantiated to a new object type that models the duration of the relationship. For example, assume an enterprise model where EMPLOYEE is related to DEPARTMENT. The relationship type between EMPLOYEE and DEPARTMENT is a non existence dependent one to many relationship type: each department has many employees and each employee is assigned to at most one department. However, the existence of an employee does not depend on the existence of a department and conversely, the existence of a department does not depend on the existence of an employee. Even if the relationship is mandatory for employees, that is, an employee must always be related to a department, there is no existence dependency because we can assume that an employee can change from department. As the relationship that models the allocation of employees to departments does not express existence dependency, the existence dependency graph will contain a third object type ALLOCATION that models this relationship: it is a kind of contract object type that models everything that can happen during the time an employee is allocated to a department. The new object type ALLOCATION is existence dependent of both EMPLOYEE and DEPARTMENT, with cardinality one and many respectively. That is: an allocation always refers to exactly one and the same employee and as an employee can be allocated to at most one department at the time, each employee is referred to by at most one allocation object at a time. An allocation always refers to exactly one and always the same department and a department can be referred to by many allocations at one point in time, namely one per employee allocated to that department. Fig. 8 shows the Object-Relationship diagram (drawn according to the UML notations) and the resulting existence dependency graph.

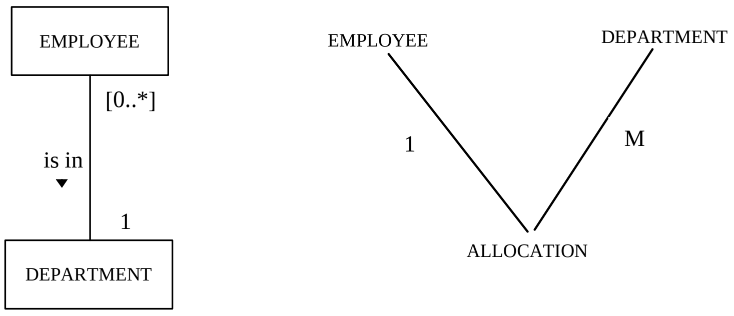


Fig. 8 Object-Relationship Model and Existence Dependency Graph for EMPLOYEE and DEPARTMENT

The same reasoning applies to one-to-one and many-to-many relationships. For example, if in a network management model client computers can be connected to many server computers and server computers serve many client computers, the connection must be modelled as a separate object type. This is illustrated in Fig. 9.

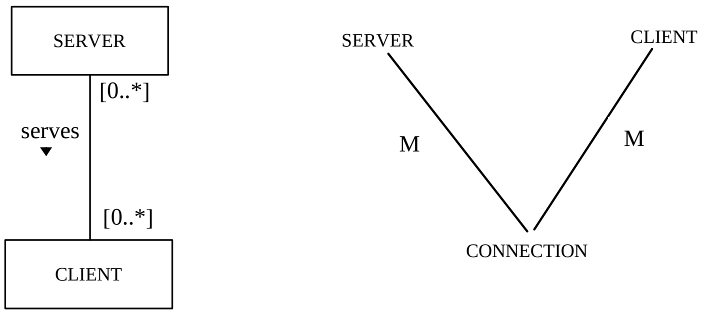


Fig. 9 Converting a many to many relationship type to an existence dependent object type

Unary Relationships. Unary relationship types never express existence dependency, because an object can not be existence dependent of itself, nor of an object of the same type. (If this was possible, we would have an infinite chain of existence dependent objects, with no top parent object. It would thus be impossible to create the first occurrence of that type, because this would require an already existing parent occurrence, which is in contradiction with the fact that we are creating the first occurrence). Unary relationship types are thus always converted to an existence dependent object type. For example, if marriage is modelled as a unary relationship from person to person, the existence dependency graph will contain two object types, namely person and marriage (see Fig. 10). Another example is a model for an organisational hierarchy of departments: the hierarchy relationship should be modelled as a separate object type (Fig. 11).

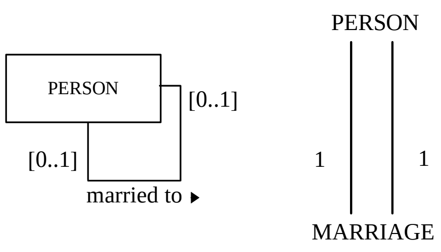


Fig. 10 Unary Relationship type: Marriage

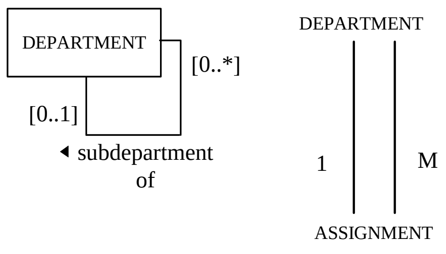


Fig. 11 Unary Relationship type: Sub-department-of

N-ary relationships. This kind of relationships will be converted to a new object type. For example, assume a university model where professors teach courses to student classes in a particular class-room. This can be modelled as a relationship type of arity four in an Object-Relationship diagram. In the existence dependency graph, the relationship is modelled as the contract object LECTURE between PROFESSOR, STUDENT CLASS, ROOM and COURSE (see Fig. 12).

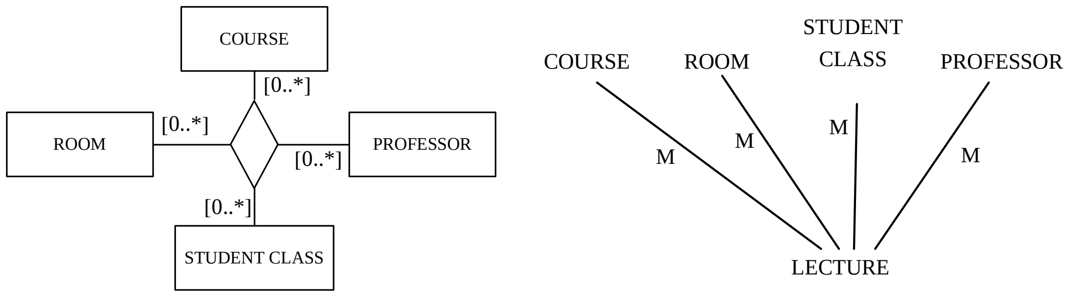


Fig. 12 Lectures at the university

Note that with this conversion of relationships to existence dependent object types we assume that relationships are unchangeable: when a link is changed, a new relationship instance is created. For example, if an employee is allocated to a new department, this means that the old allocation object is destroyed and a new allocation object is created.

2.4 The difference between existence dependency and the Part Of-relation or Aggregation

Although the Part-Of relation is in essence an association between objects like any other association, many designers of OOA methods estimate that it deserves special attention and notation [10, 22, 16, 21, 5]. This is probably due to the fact that the notion of 'Part-Of' embodies some aspects of existence dependency[4] and propagation of events[5]. However, existence dependency and the part-of relation are not equivalent concepts: some part-of relations are existence dependent and some are not. For example, wheels are part of a car, but if a

4. See for example [21], p. 38: "The existence of a component object may depend on the existence of the aggregate object of which it is part. ... In other cases, component objects have an independent existence, ..."
5. See for example [21], p. 60: "Propagation is the automatic application to a network of objects when the operation is applied to some starting object. For example: moving an aggregate moves its parts; the move operation propagates to the parts. Propagation of operations to parts is often a good indicator of aggregation."

car is disassembled, the wheels still exist as objects. Order lines, on the contrary can usually not exist independent of the order of which they are a part. Similarly, if the parts are not existence dependent of the aggregate, propagation of events is not straightforward. Disassembling a car ends the life of a car-object, but does not end the life of the constituent parts. On the other hand, deleting an order usually implies the deletion of the order lines.

As pointed out in [7], the Part-Of relation is quite underdefined. The use of aggregation is not clear cut and the semantics of the Part-Of relationship are not precisely defined. It is not always clear whether parts are existence dependent of the aggregation or not, when and which operations should be propagated, and whether the Part-Of relation is transitive or not [3, 5, 6, 16, 21, 22]. Sometimes the definitions in different methods contradict each other.

An example of a confusing or incomplete definition is the concept of “composition” as defined in the Unified Modelling Language [20]. The Notation Guide says: “Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end (the "component") may not exceed one (it is unshared). The aggregation is unchangeable (once established the links may not be changed). Parts with multiplicity > 1 may be created after the aggregate itself but once created they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.” This definition doesn’t say anything about parts with multiplicity ≤ 1. Are they always created at the same time the aggregate is created ? Can they die before the aggregate is destroyed ? Can you have parts with multiplicity 0 ?

A much better definition of composite objects in the context of object-oriented databases can be found in [18]. This paper makes a difference between dependent and independent composites. The semantics of existence dependency are similar to the semantics of a dependent component: existence dependency implies that the existence of the existence dependent object depends on the existence of the parent. However, in [16] it is not explicitly stated that the reference from parent to component cannot be reassigned. In addition, whereas in [18] dependency implies a cascading delete of components when the root object of a composite is deleted, we assume that ending the life of a parent object is invalid as long as there are existence dependent objects for this parent.

The existence dependency relation proposed in this paper is a valuable alternative for the Part-Of relation. A Part-Of relation with existence dependent parts can simply be replaced by existence dependency: in case of existence dependent components, the existence dependency relation is identical to the Part-Of relation. Fig. 13 illustrates this with an example. The Part-Of relation is drawn according to the notation of OMT [21]: it is drawn as a line between part and whole with a diamond on the side of the whole; the black dot denotes a cardinality of Many.

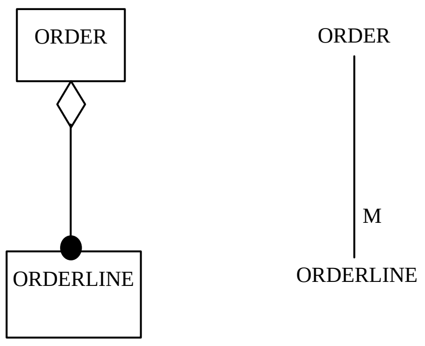


Fig. 13. Existence dependent parts: OMT Object diagram and EDG

In case of non-existence dependent parts, the involvement of the part in the aggregate is modelled as a separate object type. This new object type is a kind of *contract* between the part and the aggregate for the duration of the part-of relationship. The object diagram in Fig. 14 states that a PC consists of one monitor, one system box, one keyboard and zero to many external drives.

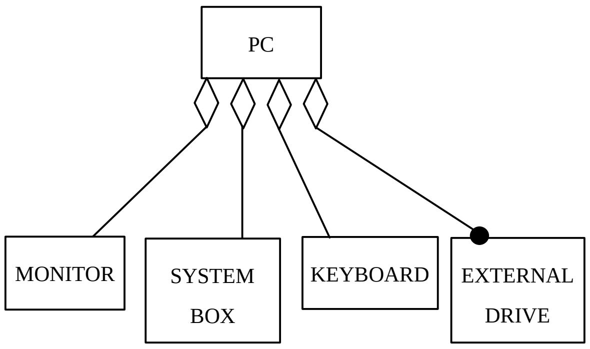


Fig. 14. Non existence dependent parts: OMT Object diagram

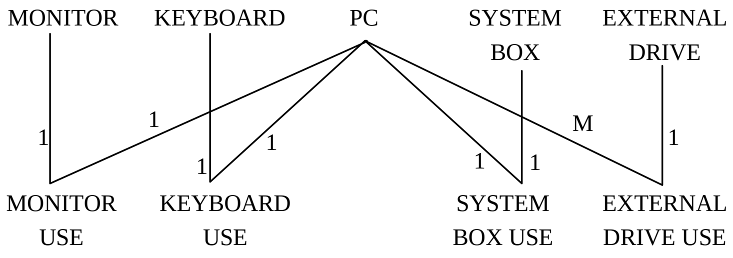


Fig. 15. Non existence Dependent parts: EDG

Fig. 15 shows the equivalent EDG. The cardinalities in this graph express the fact that each component can be in use by at most one PC at one point in time and that a PC consists of one monitor, one keyboard, one system box and zero to many external drives. Reuse of components is possible.

These examples demonstrate that existence dependency is a valuable alternative for the sometimes confusing concept of aggregation: its semantics embodies the interesting features of the Part-Of relation and its precise definition allows for an unambiguous use of the concept. Compared to the Part-Of relation, the Existence Dependency has the advantage that

- its semantics are simple and easy to formalise,
- the use of this relationship is clear cut,
- existence dependency allows for consistency checking at the side of the dynamic model by considering existence dependent objects as contracts,
- the semantics of the concurrency aspects are well-defined [25, 9].

The last two points will become clear when the modelling of behaviour and subsequent consistency checking have been explained.

2.5 A final consideration

Obviously, classifying object type according to existence dependency increases the number of object types in the domain model because all non-existence dependent relationships are turned into an object type which is existence dependent of the object types participating to the relationship. However, this is experienced as a positive effect as volume is easier to deal with than logical complexity. And that is exactly what existence dependency does: it reduces complexity by making the relationships between object types simple and easy to understand. An existence dependency graph is much easier to understand than a semantically rich object-relationship diagram. Note that this does not mean that classical modelling techniques cannot be used during the requirements analysis process. But in the end, the analyst has to come up with a domain model were all object types are arranged in an existence dependency graph (in combination with Generalisation/Specialisation). As will be seen in the next sections, this will ease the task of behaviour modelling and allow to ensure semantic integrity between the structural and behavioural aspects of object types.

In addition, nothing should prevent the software developer of merging classes together at implementation time. In the example of the employees and departments (Fig. 8), the object type ALLOCATION that results from the relationship type that models the allocation of employees to a department, could well be merged with the object type EMPLOYEE. However, this kind of efficiency improvements are to be done at design time.

3. MODELLING DYNAMIC ASPECTS OF OBJECT TYPES

The concept of existence dependency implies some operational semantics: as the life of an existence dependent object is to be embedded in the life of its parent, some constraints are put on the timing of the creation, modification and ending of objects. In object-oriented analysis, behaviour of objects is encapsulated within the object type. In addition, objects can interact with each other through messages. It is clear that the definition of behaviour and its implied semantics should be compatible with the operational semantics implied by existence dependency. This is also true for a concept such as composition: the concept might be less confusing if the relation between the semantics of structural composition and the behavioural aspects of composition were explicitly stated.

This section will shortly present some techniques for modelling the behavioural aspects of object types and will then explain how semantic integrity can be ensured between this dynamic model and the EDG.

3.1 Object Interaction Modelling

In object-oriented analysis, communication between objects is usually based on the concept of message passing. The conceptual schema proposed in this paper uses interaction by means of common event types rather than communication by means of message passing. An analysis model is supposed to specify what the information system should do while the design model specifies how things should be done. When interaction between objects is modelled, message passing is much closer to the how than to the what. For this reason, in the conceptual model proposed in this paper, objects do not communicate with each other by means of message passing. The formalism of synchronous participation to events is used instead. For example, assume a library environment with an object type MEMBER and an object type COPY. Copies can be borrowed, which is modelled by the event type *borrow*. This might be modelled by specifying a message that is sent from COPY to MEMBER on occurrence of a *borrow* event. Or it might seem more natural to have MEMBER send a message to COPY. Rather than debating on the direction of the message, it is much more essential to model the essence of this type of communication. In reality, none of the objects is sending a message to the other. What really happens is that when a borrow event occurs, two objects *are involved in* this event: one member and one book. For this reason we say that objects do not communicate by means of message passing but that they communicate by jointly participating in events. For the library example this means that both the member and copy object types are provided with a method called 'borrow' and it is agreed that upon occurrence of a *borrow* event both methods will be executed simultaneously. In this way, object types synchronise on common events. This way of communication is similar to communication as defined in the process algebras CSP [13] and ACP [2]. Message passing is more similar to CCS [19].

This communication concept gives rise to what one could call simultaneous polymorphism: a method takes several forms (with possibly different interfaces) in different classes, namely, in each of the classes that is involved in the event. As opposed to classical polymorphism, a consequence of inheritance, no choice is made between the different forms of the method, but all methods are executed simultaneously.

3.2 The Object Event Table (OET)

Modelling interaction by means of common event types requires the identification of relevant business event types. The Object Event Table matches object types against event types. The list of object types are those that appear in the Existence Dependency Graph. The Object Event Table (OET) contains one row per event type and one column per object type. Each cell in the table indicates whether an object type is involved in an event type or not. In this way, a subset of event types is assigned to each object type. This subset is called the *alphabet* of an object type P and is denoted $S_A P$. It contains all the event types that are relevant for that particular object type. In addition, the alphabet of P is partitioned in three mutually disjoint sets: c(P), m(P) and e(P) with

c(P) = {The set of event types that create an occurrence of type P} $\in S_A P$
m(P) = {The set of event types that modify an occurrence of type P} $\in S_A P$
e(P)= {The set of event types that end an occurrence of type P} $\in S_A P$

c(P) and e(P) are never empty: per object type there is at least one event type that creates occurrences of this object type and there is at least one event type that ends[6] occurrences of this object type. As there are not always modifying event types, m(P) possibly is empty.

Definition 3

The Object Event Table is a table with one row per event type and one column per object type. Each cell contains either a blank, a 'C', an 'M' or an 'E', which stands for “creates occurrences”, “modifies occurrences” and “ends occurrences” respectively.

Per column, there is at least one row with a ‘C’ and one row with an ‘E’.
c(P), m(P) and e(P) are the sets of all event types for which there is respectively a ‘C’, ‘M’ or ‘E’ in the column of P.

Each event type must be relevant for at least one object type: so on each row there is at least one column with a ‘C’, an ‘M’ or an ‘E’.

Graphical representation

The OET is drawn as a matrix containing one row per event type and one column per object type. A 'C', 'M' or 'E' on a row-column point of intersection indicates that this particular event type is an element of respectively c(P), m(P) or e(P), where P is the object type corresponding to the column. Fig. 16 is a graphical representation of Example 4.

Example 4

Let us assume that for the library example the universe of event types is
{enter, leave, acquire, classify, borrow, renew, return, sell, lose}
The partitions of the alphabets of the object types COPY, MEMBER and LOAN are as follows:

S_ACOPY = {acquire, classify, borrow, renew, return, sell, lose}
with
c(COPY) = {acquire}
m(COPY) = {classify, borrow, renew, return}
e(COPY) = {sell, lose}

6. *Ending* an occurrence is not necessarily the same as physically destroying or deleting object occurrences. An ending event brings an object to a final state. Objects that are in a final state cannot be subject to further changes and can, depending on the implemented archiving, deletion and backup policies, be removed from the database. Ending an object only says something about the permissibility of a physical removal: it is always allowed to physically remove objects that are in a final state. Analogous concepts are also used in temporal databases where information is not removed at its end of validity but is retained for querying purposes.

S_AMEMBER = {enter, borrow, renew, return, lose, leave} with
c(MEMBER) = {enter}
m(MEMBER) = {borrow, renew, return, lose}
e(MEMBER) = {leave}

S_ALOAN = {borrow, renew, return, lose}
with
c(LOAN) = {borrow}
m(LOAN) = {renew}
e(LOAN) = {return, lose}

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

Fig. 16. OET for the library example

The object event table is the basis for the definition of abstract data types for object classes: the abstract data type will contain a method for each 'C', 'M' and 'E' in the classes' column.

Note that only those participations should be indicated where **exactly one** occurrence of the object type participates in events of the given type. In the given example, in a 'borrow' event there is exactly **one** copy, **one** loan and **one** member that are involved. Participation of multiple occurrences is usually not correct[7].

3.3 Existence Dependency Graph versus Object Event Table

The existence dependency graph and dynamic schema (object event table and sequence constraint specifications) are dual perspectives of the same reality and thus must be consistent with each other. Therefore the semantics of the existence dependency graph must also be a subset of those of the dynamic schema. This means (among other things) that for each object in the Existence Dependency graph there is one column in the Object Event Table (and vice versa). An additional requirement is that a parent object type has to participate in all event types in which one of its existence dependent object types participates:

Propagation rule
If P is existence dependent of Q, the alphabet of P must be a subset of the alphabet of Q.

This can be explained as follows. Existence dependent objects should not participate in any event without the parent object having knowledge of this event. By including the alphabet of the existence dependent object type in the alphabet of the parent object type, all possible places for information gathering and constraint definition are identified. For example, the *borrow* method of the class COPY is the right place to count the number of times a copy has been borrowed and to implement a rule such as 'When a copy has been borrowed 500 times, check if it still is in good condition. If not, the copy should be taken out of circulation and sent to the book binder'. Obviously, not all object-event methods will have a meaningful content. At implementation time, empty methods can be removed to increase efficiency.

In addition, by including the event types of the existence dependent objects in the alphabet of the parent, sequence constraints that concern event types of different existence dependent objects can be specified as part of the behaviour of the parent. For example, assume a library where books can be reserved. This can be modelled by means of an additional object type RESERVATION that is existence dependent of COPY and MEMBER. Sequence constraints such as 'a reservation can only be made for copies that are on loan' relate to more than one object type, namely LOAN and RESERVATION. They can be specified as part of the behaviour of a common parent of these object types, COPY in the given example.

Additional restrictions can be put on the subsets of the alphabet. An existence dependent object cannot be created before its parent exists nor can it exist after its parent has been ended. Creating an existence dependent object means that either the parent is created at the same time (e.g. creating the first order line creates the order) or that the parent object type already exists (e.g. opening an account for an existing customer). In the latter case,

7. An exception to this is when each occurrence (of the same type) needs a different interpretation of the event. For example, in a sales transaction two owners are involved: a buyer and a seller. This gives rise to the definition of aliasing event types [25].

the creation of an existence dependent object modifies the state of the parent. As the alphabet of the existence dependent object is a subset of the alphabet of the parent object type, this means that the set of creating event types of the existence dependent object is a subset of the creating and modifying event types of the parent. So $c(P) \subseteq [c(Q) \cup m(Q)]$, when $P \leftarrow Q$. Modifying an existence dependent object always modifies the state of the parent, so $m(P) \subseteq m(Q)$. Finally, ending an existence dependent object also modifies the state of the parent. If the last existence dependent object is ended, then the parent can be ended at the same time. As a result: $e(P) \subseteq e(Q) \cup m(Q)$. 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 parent object type.
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 parent object type.
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 parent object type.

Example 5

LOAN is existence dependent of COPY and MEMBER. The OET in Fig. 16 satisfies both the Propagation and the Type of Involvement rule. The alphabet of LOAN is a subset of the alphabet of both MEMBER and COPY. The event type *borrow* creates a loan, modifies a member and modifies a copy. So for the 'C' in the LOAN column, we have a 'M' in the MEMBER and COPY column. The event type *renew* modifies the state of a loan, a member and a copy. And thus for the 'M' in the LOAN column, we have 'M' in the MEMBER and COPY column. Finally, *return* and *lose* are both ending the life of a loan and are both modifying the state of a member. For the 'E's in the LOAN column we have 'M's in the MEMBER column. *Return* is modifying the state of a copy and *lose* is ending the life of a copy. So for the 'E's in the LOAN column we respectively have an 'M' and 'E' in the COPY column. As a result, the object event table satisfies the type of involvement rule and the propagation rule

The propagation rule and type of involvement rule together ensure that the life span of the existence dependent object is embedded in the life span of the parent.

As explained in the previous section, non existence dependent relationship types are always turned into object types existence dependent of the object types that participate in the relationship. As a consequence of the propagation rule, the alphabet of a relationship object type always is a subset of the alphabet of the entity object types it relates. Moreover, when two (or more) object types share a number of common event types, it makes sense to demand that this relationship between object types be modelled by a common existence dependent object type that has the role of a 'contract'[8]. Possibly, the shared event types can be spread across more than one existence dependent object type. We call this the **contract rule**.

Contract rule

When two object types share two or more event types, the common event types must be in the alphabet of one or more common existence dependent object types.

Example 6

Besides borrowing books, it is also possible to reserve books that are not on shelf. If a member changes his/her mind and decides not to fetch the copy, (s)he can cancel the reservation. The events 'reserve', 'cancel' and 'fetch' are added to the object event table (see Fig. 17). The shaded area shows the common event types of COPY and MEMBER. Some of the events are also in the alphabet of the existence dependent object type LOAN, but 'reserve' and 'cancel' do not appear in the alphabet of a common existence dependent object type. According to the contract rule, the two event types should either be included in the alphabet of LOAN, or they should be included in the alphabet of a new object type, existence dependent of both MEMBER and COPY. The latter solution is to be preferred, because a loan can occur without a reservation and a reservation can occur without being followed by a loan. The correct object event table is as in Fig. 18.

8. The notion of contract will be further elaborated when talking about sequence constraints.

	MEMBER	COPY	LOAN
enter	C		
leave	E		
acquire		C	
classify		M	
borrow	M	M	C
renew	M	M	M
return	M	M	E
sell		E	
reserve	M	M	
cancel	M	M	
fetch	M	M	C
lose	M	E	E

Fig. 17 New OET for the library example

	MEMBER	COPY	LOAN	RESER-VATION
enter	C			
leave	E			
acquire		C		
classify		M		
borrow	M	M	C	
renew	M	M	M	
return	M	M	E	
sell		E		
reserve	M	M		C
cancel	M	M		E
fetch	M	M	C	E
lose	M	E	E	

Fig. 18 Correct OET for the library example

Contracts between objects are thus the counterpart of relationships between objects in an Object-Relationship Diagram: common event types between objects always indicate the presence of at least one relationship between these objects. On the other hand, when in the static schema a relationship is modelled between two object types, then either this relationship expresses existence dependency or either the relationship should be instantiated to a new object type that is existence dependent of the object types participating to the relationship. In the first case, the alphabet of the existence dependent object type is a subset of the alphabet of its parent and in the second case the alphabet of the new relationship object type will be the set of event types common to all its parent object types.

Note that the contract rule is only applicable in case of **two or more** common event types. If there is only one common event type (e.g. ‘fetch’ is common to RESERVATION and LOAN), it would be wrong to model an additional object type: a single event type is a contract with no duration and at least two event types are necessary to determine the life span of a candidate object type. An additional object type cannot be the result of only one event type.

3.4   Techniques for modelling sequence constraints

In general, events are not allowed to occur in a random order during the life cycle of an object. For example, a book can not be returned if it has not previously been borrowed. In order to specify this kind of constraints, each object is equipped with a ‘life cycle state’ variable and a description of allowed sequences of event types. The latter are described by means of a regular expression, a Jackson Structure diagram [15] or a Finite State Machine. From a mathematical point of view, these three techniques are perfectly equivalent. Fig. 19 depicts how sequence, choice (exclusive and exhaustive selection) and iteration are modelled in each of the first three techniques. In this paper we will use any of these techniques, but any other technique that is mathematically equivalent to Regular Expressions would do as well.   Harel Statecharts [11] are an example of such a technique that resolves some of the modelling problems that are attributed to the use of Finite State Machines, but that is still mathematically equivalent to Regular Expressions.

In order to simplify expressions, a special event type “do nothing” is provided. In a regular expression this event is denoted by ‘1’, in a finite state machine by a transition labelled with an ‘&’ and in a JSD diagram by a box labelled with an underscore. The initial state of a finite state machines is depicted as a circle with an incoming arrowhead and final states are drawn as double circles.

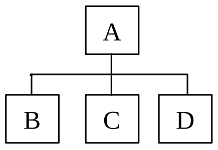
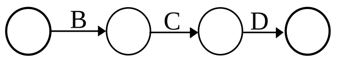
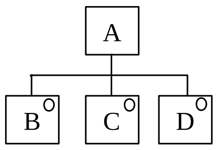
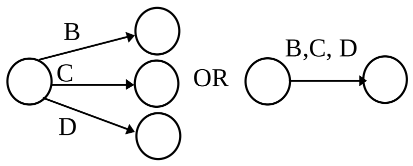
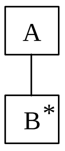
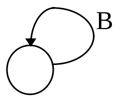
A is a sequence of first B, then C and then D.	A = B.C.D		
A is a choice between B, C and D.	A = B + C + D		
A is an iteration of B.	A = B*		

Fig. 19 Natural Language, Regular Expression, JSD and Finite State Machine specification for sequence, selection and iteration

Note that in fact the object event table already specifies some sequence constraints. The default assumption is that objects first must be created before they can be modified and that an ending event is always the last event in an object's life. But sometimes it is necessary to put additional constraints on sequences of event types.

Example 7

Imagine a library where all available books can be searched for by means of an on-line catalogue. The regular expression for the definition of the object type COPY could be as follows:

COPY = acquire.classify.(borrow.(renew)*.return)*.[1 + (borrow.(renew)*.lose)].
declassify. remove_copy

The equivalent finite state machine and JSD-diagram are given in Fig. 20.

This specification should be read as:

In the context of a library, the existence of a copy starts with its acquisition. The copy is then classified, this is, registered in the catalogue. It can then be borrowed and returned to the library many times consecutively. Loans can be renewed and the copy can possibly be lost in stead of being returned to the library. Finally the copy is removed from the catalogue (declassify) and the set of existing copies (remove_copy).

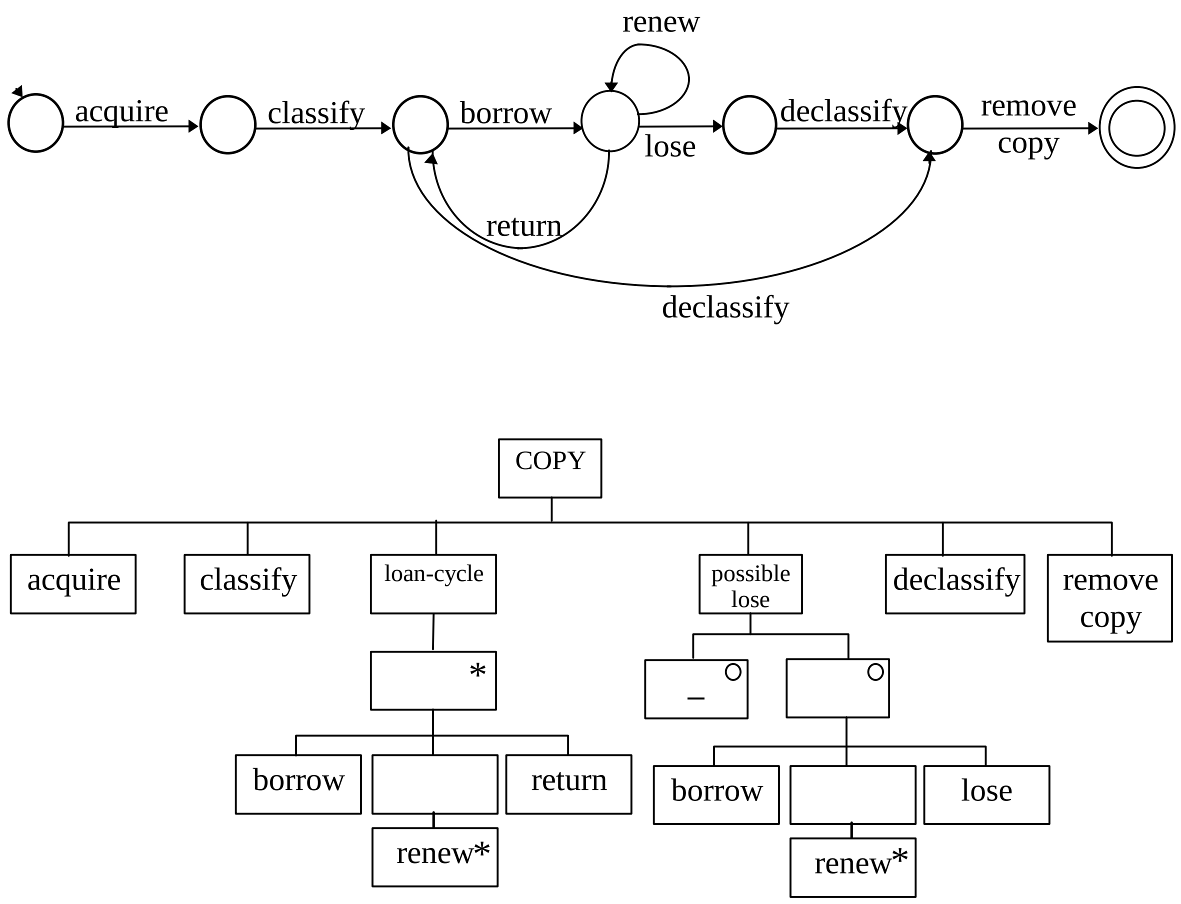


Fig. 20 Finite State Machine and JSD diagram for COPY

Example 8

Fig. 21 shows the JSD-diagrams for the library example of Fig. 18. The conversion from JSD diagram to regular expressions is pretty straightforward: each box is either an event or a sub-expression between brackets. Sequence is indicated by periods, selection by a '+' sign and iteration by a '*'. The equivalent regular expressions are as follows:

COPY = acquire . classify . (borrow + fetch + renew + return)* . (sell + lose)
RESERVATION = reserve . (cancel + fetch)
MEMBER = enter . (borrow + fetch + renew + return + lose)* . leave
LOAN = (borrow + fetch) . (renew)* . (return + lose)

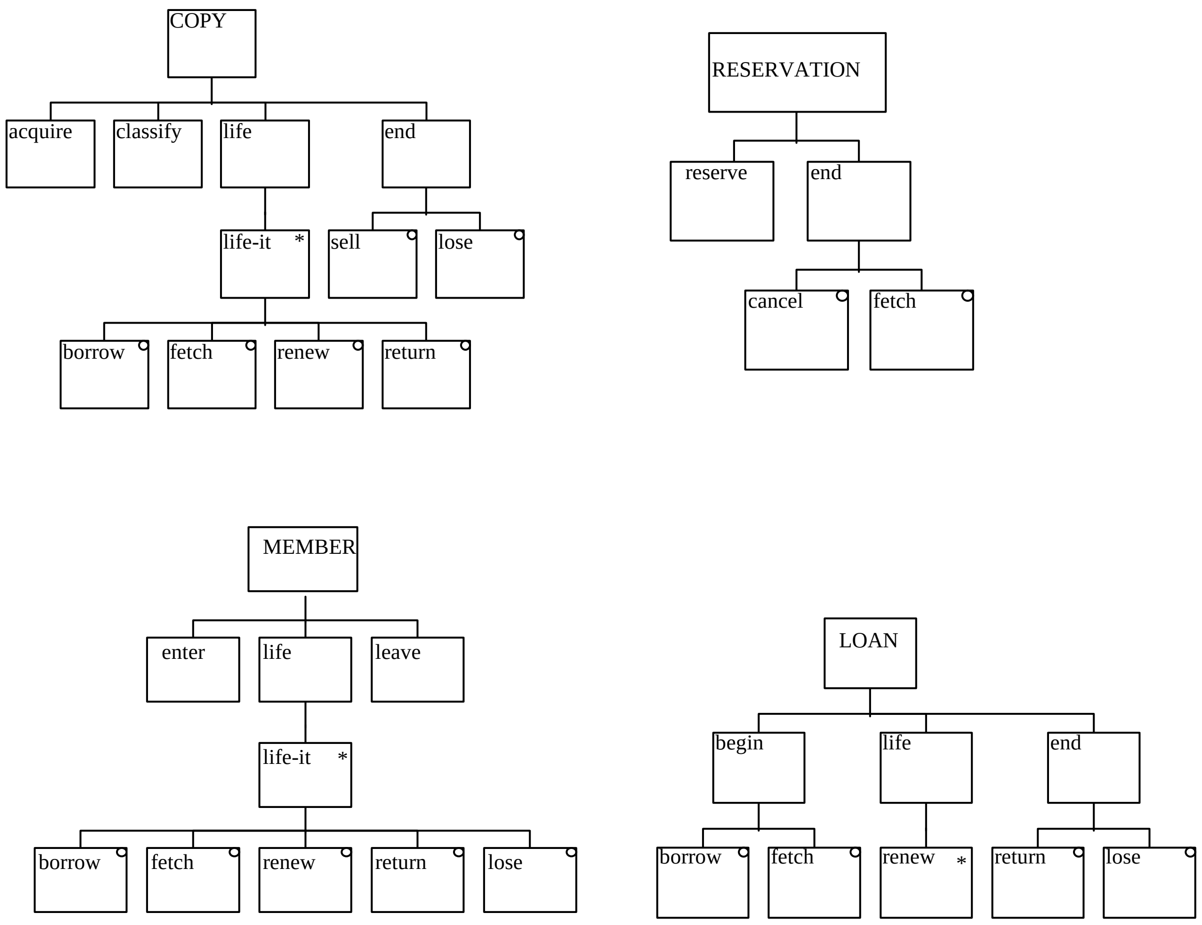


Fig. 21 JSD diagrams for the library example

Conversion of JSD diagrams and regular expressions to Finite State Machines is less straightforward. Algorithms for these kinds of transformations can be found in any book on compiler theory, e.g. [14, 1].

3.5 Formalising communication

As said before, it is important to guarantee consistency between static and dynamic characteristics of object types. As a result, the semantics of existence dependency have their consequence on the specification of sequence constraints. More especially, as the alphabet of an existence dependent object type is a subset of the alphabet of its parent object type (propagation rule), parent object and existence dependent object will have to synchronise on all events in which the existence dependent object participate because of the principle of communication by means of common event types. It is thus necessary to investigate the life cycle of parent and existence dependent object type for possible contradictions.

In order to gain more insight in the paradigm of communication by means of common event types and in order to be able to check sequence constraints for consistency with the existence dependency graph, some more detailed and formal definitions on sequence constraints and parallel composition are required. However, formal definitions will be avoided as much as possible. For a summary of the main definitions we refer to the process algebra of MERODE as described in [9]. Full formal definitions can be found in [25].

As defined in the previous paragraph, the set of event types an object type is involved in, is called the **alphabet** of an object type. The finite state machine of an object type defines a set of valid scenarios over this alphabet. A scenario is a sequence of event types, e.g. borrow^renew^return. If iteration operators are used in the life cycle expression, the number of valid scenarios for an object type is infinite.

Example 9

The set of valid scenarios defined by RESERVATION (Example 8) is
{reserve^cancel, reserve^fetch}
The set of valid scenarios for LOAN is infinite:
{borrow^return, borrow^lose, fetch^return, fetch^lose, borrow^renew^return, borrow^renew^lose, fetch^renew^return, fetch^renew^lose, borrow^renew^renew^return, borrow^renew^renew^lose, …}

Communication by means of common event types imposes some restrictions on the set of allowable scenarios for an object when it runs in parallel with other objects. Calculating the result of parallel composition is done by comparing the sets of scenarios. However, as object types can have different alphabets and as synchronisation is required for common event types only, comparison must be made by taking account of these common event types only. Therefor a projection operator '\B' is defined (B will be the set of common event types) that replaces each event type that is not in B by the "do-nothing" event type. The projection can be applied to individual scenarios as well as to definitions of sequence constraints (JSD-diagrams, FSMs or regular expressions). When it is applied to scenarios, the "do nothing" event types can just be dropped from the scenario. When applied to regular expressions, the resulting expression can be simplified by using the following axioms:

– for any event type a, choosing between a and a is just a: a + a = a
– the "do nothing" event type can be dropped in a sequence: 1 . a = a = 1 . a
– "do nothing" is always included in an iteration (namely, do it zero times), so (a + 1)* = a*

Example 10

Given the object definitions of Example 8, the following is a valid scenario for copy:
acquire^classify^borrow^renew^return^borrow^renew^renew^lose

Now let B = {borrow, renew, return}. If we apply the projection \B we obtain the scenario:
1^1^borrow^renew^return^borrow^renew^renew^1
= borrow^renew^return^borrow^renew^renew

If the same projection is applied to the regular expression for COPY the following expression is obtained:

COPY \ {borrow, renew, return}
= [acquire . classify . (borrow + fetch + renew + return)* . (sell + lose)] \ {borrow, renew, return}
= 1 . 1 . (borrow + 1 + renew + return)* . (1 + 1)
= (borrow + renew + return)*

Some readers might have wondered why in the library example (Example 8), the sequence constraints for COPY do not include the constraints that apply to a loan, for example that a *return* event should always be preceded by a *borrow* or *fetch* event. The reason for this lies in the fact that COPY and LOAN objects will synchronise on common events.

The enterprise model specifies object *classes*, but the actual system will consist of object *occurrences*. These objects run concurrently and have to synchronise on events in which they jointly participate. In order to be able to calculate the behaviour of a system composed of many concurrent objects, a parallel-operator || can be defined. This operator expresses the fact that when two objects run concurrently, only those scenarios are valid where both objects agree on the sequence of common events.

Example 11

Suppose we have the following definition for the object types COPY and LOAN:
COPY = acquire.classify.(borrow + renew + return)*.(sell + lose)
LOAN = borrow.(renew)*.(return + lose)

The existence dependency graph specifies that a copy can have at most one loan at one point in time. Hence, during its life a copy will participate zero, one or more times *consecutively* to a loan, which is an *iteration* of loan (see also Fig. 7). To find out what the set of accepted scenarios will be, we thus must calculate the parallel composition of COPY and the iteration of LOAN. The behaviour of a copy that can be on loan zero, one or more times consecutively is:

COPY || (LOAN)*
= COPY || (borrow.(renew)*.(return + lose))*
= acquire.classify.[borrow.(renew)*.return]*.[sell + borrow.(renew)*.lose]

Fig. 22 depicts the same expression calculated by means of finite state machines. An algorithm to calculate the result of the parallel composition of Finite State Machines can be found in [1, 14].

The sequence constraints of the calculated expression are exactly expressing what we understand by a copy than can be borrowed many times consecutively. For this reason it is not necessary to include the sequence constraints of LOAN in the definition of COPY: they will automatically be enforced when COPY and LOAN objects run concurrently.

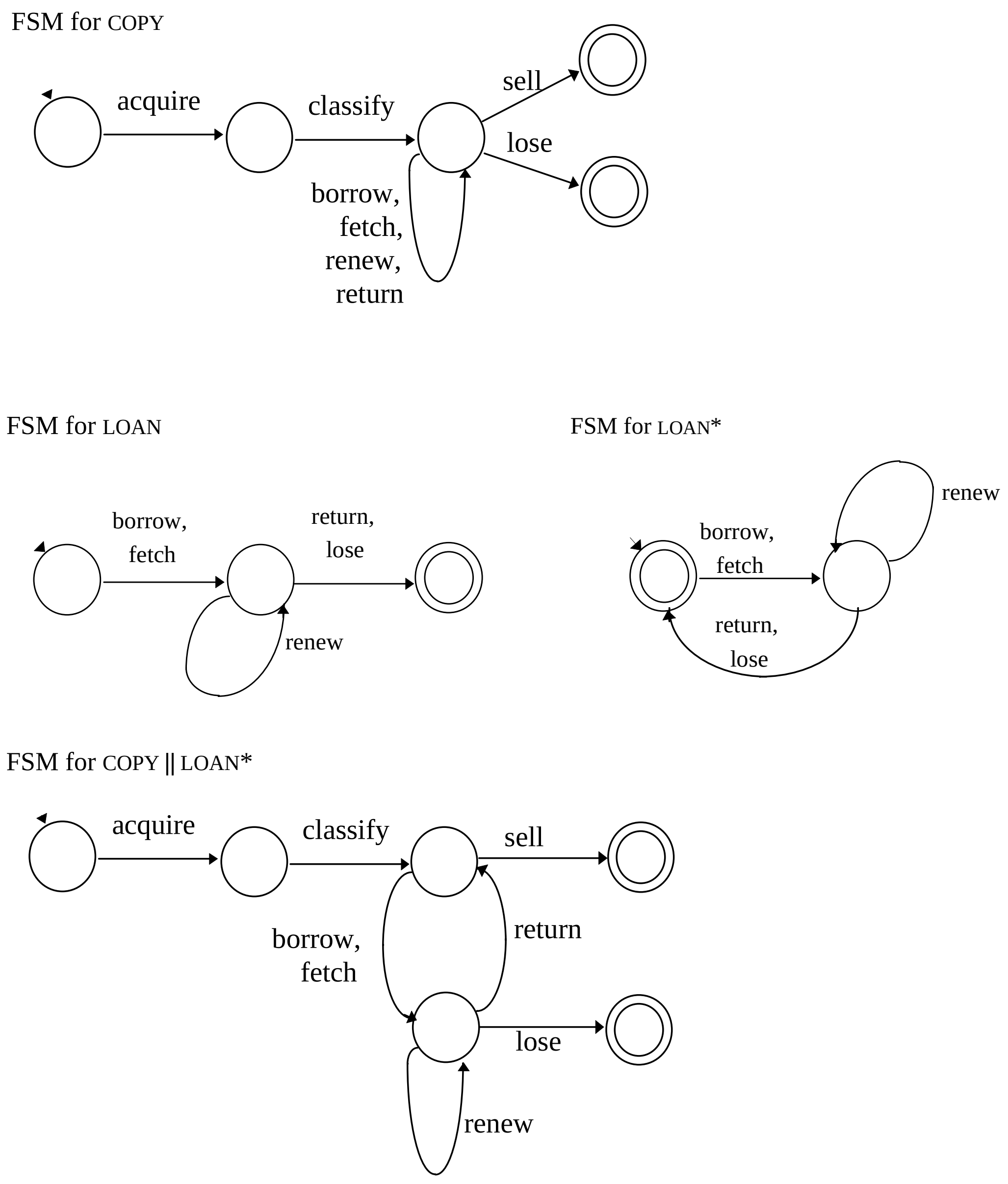


Fig. 22 Calculating the behaviour of a copy on loan.

3.6 Consistency checking with the OET and the EDG

The essence of consistency checking of sequence constraints is based on two principles: propagation of events and existence dependent objects as contracts between parent object types.

Propagation of events. Event types of existence dependent object types are propagated to the parent object type (see Propagation Rule).

Existence dependent objects as contracts. When two object types have common event types, at least one existence dependent object type must be modelled that has the common event types in its alphabet. This existence dependent object type acts as a contract between the parent object types: the contract ensures that the participating object types agree on the allowed behaviour (sequences of events).

In order to understand the consequences of this notion of contract for sequence constraints, let us imagine an enterprise with only two object types, a parent object type Q and an existence dependent object type P (e.g. ORDER and ORDER_LINE). According to the definition of the ||-operator, both object types will run concurrently and such a system will only accept sequences of events that satisfy the sequence constraints of both P and Q. As a result, any scenario of an existence dependent object that is not acceptable from the point of view of its parent object, will *always* be rejected. It can thus be removed from the life cycle definition of the existence dependent object type. It thus seems sensible to demand that a parent object type Q can accept all scenarios of its existence dependent object type P. We say that the existence dependent object type P must have more stringent sequence constraints than or must be *more deterministic than* the parent object type Q. The difference between the general case and the case in which P is more deterministic than Q is depicted in Fig. 23.

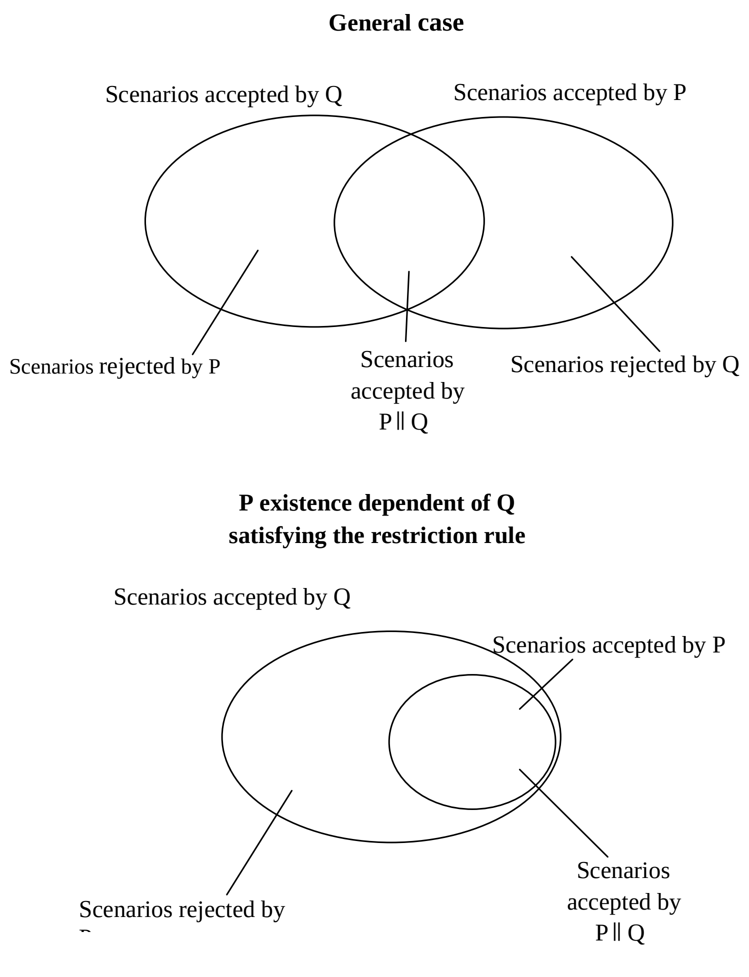


Fig. 23 Restriction rule for existence dependent object types

As a result, when an object type has more than one parent object type, the set of scenarios accepted by the existence dependent object type is a subset of the intersection of the sets of scenarios accepted by all its parent object types. In this sense, the existence dependent object type acts as a contract between the parent object types: for the common event types it defines the set of scenarios that will be accepted by all participants. For the library example LOAN is a contract between MEMBER and COPY (see Fig.24).

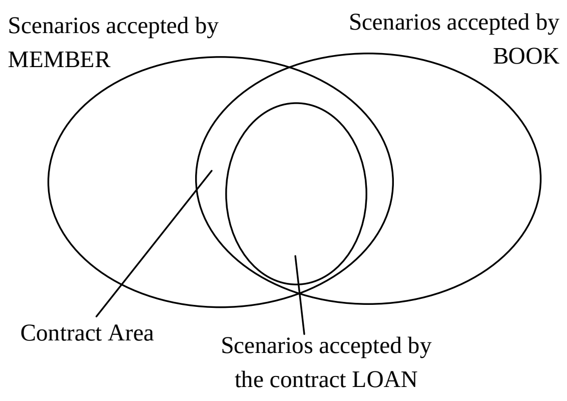

Scenarios accepted by MEMBER
Scenarios accepted by BOOK
Contract Area
Scenarios accepted by the contract LOAN

Fig.24. loan as a contract between member and copy

How can this relationship *more deterministic than* be checked ? The basic idea is to project the sequence constraints of the parent object type on the set of common event types (that is, the alphabet of the existence dependent object type) and see if these sequence constraints are less stringent than those of the existence dependent object type

Definition 4

An object type P is **more deterministic than** an object type Q (which is written as $P \leq Q$) if the alphabet of P is a subset of the alphabet of Q and if the scenarios of P are all acceptable for Q.

Example 12

With the sequence constraints as in Example 8 and LOAN existence dependent of COPY and MEMBER, we have:

COPY = acquire.classify.(borrow + fetch + renew + return)*.(sell + lose)
MEMBER = enter.(borrow + fetch + renew + return + lose)*.leave
LOAN = (borrow + fetch).(renew)*.(return + lose)

The alphabet of LOAN is a subset of the alphabet of COPY. In order to compare the scenarios defined by COPY and LOAN, the expression of COPY is projected on the alphabet of LOAN:

COPY$\backslash S_A$LOAN
= acquire.classify.(borrow + fetch + renew + return)*.(sell + lose)$\backslash S_A$LOAN
= (borrow + fetch + renew + return)*.(1 + lose)

Apparently, the scenarios defined by LOAN are all acceptable for COPY.
As a result, LOAN $\leq$ COPY: LOAN is more deterministic than COPY.

The same reasoning applies to MEMBER:

MEMBER$\backslash S_A$LOAN
= enter.(borrow + fetch + renew + return + lose)*.leave$\backslash S_A$LOAN
= (borrow + fetch + renew + return + lose)*

Which is also less deterministic than LOAN and thus LOAN $\leq$ MEMBER.

The full set of schema constraints for the specification of behaviour takes not only existence dependency into account, but also the elements specified in the OET. The OET defines the alphabet of an object and partitions this alphabet in creating, mutating and ending event types. These elements have an influence on what can be considered as a valid behaviour definition of an object type. In the first place, there is the alphabet rule:

Alphabet rule

The structure diagram or expression that defines the behaviour of an object type P must contain all and only the event types for which there is a ‘C’, ‘M’ or ‘E’ in the column of P in the OET .

As already said, the partitioning of the alphabet in creating, modifying and ending event types imposes a default life cycle. This default life cycle must be respected by the sequence constraints:

Default life cycle rule

The events in c(P), m(P) and e(P) must appear as creating event types, modifying event types or ending event type respectively in the sequence restriction of the object type P. This means that the sequence constraints must be more deterministic than the default life cycle of create, modify, end.

When object types have a default life cycle it is thus in fact not really necessary to specify the sequence constraints explicitly by means of a FSM, a JSD-diagram or any other technique.

As said before, the existence dependency relation defines a partial order on $\mathcal{M}$. From the point of view of object behaviour, the partial order **more deterministic than** (≤) can be defined as the dual counterpart of the existence dependency relation. This means that

Restriction rule

If P is existence dependent of Q, then P must be more deterministic than Q.

Note that the restriction rule contains a one direction implication. It might indeed happen that P is more deterministic than Q (P ≤ Q) without P being existence dependent on Q.

Example 13

DRAWING = paint.(give_away + receive)*.throw_away
CHILD = (...).(paint + give_away + receive + throw_away)*.(...)

Then DRAWING ≤ CHILD, but a drawing is never existence dependent of a CHILD. As DRAWING and CHILD have more than two event types in common, we need at least one existence dependent object type in which all these event types are involved. In this example this is the object type

PROPERTY_OF = (paint + receive).(give_away + throw_away)

Note that if the sequence constraints imposed by each object type are written as a regular expression or an equivalent graphical representation (such as a Finite State Machine), checking the contract can be done automatically by a CASE-tool.

The combination of the propagation rule and the restriction rule implies that, in case cycles would be allowed in the existence dependency graph, all object types involved in a cycle would have identical alphabets and identical sequence restrictions. These object type could then well be collapsed into a single object type. This is an additional motivation to require the existence dependency graph to be acyclic.

Appendix B presents a comprehensive example that illustrates the process of developing a domain model and that demonstrates how the consistency checking rules help to spot errors in a schema.

4. CONCLUSION

This paper has proposed a new object type classification concept called existence dependency. The existence dependency graph establishes the structural relationships between object types. Its semantics are clear and unambiguous. In addition, classifying object types according to existence dependency allows to formulate a number of semantic integrity rules for modelling the behaviour of object types. A domain model built with the three presented techniques (EDG, OET and sequence constraints) is consistent if all the presented rules, namely the Propagation Rule, the Type of Involvement Rule, the Alphabet Rule, the Default Life Cycle Rule and the Restriction Rule are satisfied. The possibility to check a domain model for semantic integrity between structural and behavioural aspects is a major advancement in object-oriented analysis practice as the level to which current methods allow for this type of consistency checking is unfortunately rather low.

Proposing a conceptual modelling approach where object types can only be classified according to generalisation/specialisation and existence dependency is in sharp contrast with the current trend to provide analysts with techniques with vast semantic richness and as much expressiveness as possible. However, as researchers we should be aware that semantically rich techniques may be much more difficult to use. In addition, they are difficult to define in a perfectly unambiguous way. As a result, schemes built according to semantic rich techniques often need an accompanying interpretation to be understandable by users and, what is more of a problem, the interpretation of a single schema may vary from person to person. With existence dependency, users have a simple technique at hand which application is so clear cut and unambiguous that discussions about possible interpretations become obsolete.

We are aware that during the creative process of developing an enterprise model, it might be useful to use an ER-like approach that allows for more types of associations between objects. However, the final product should be expressed in terms of object types, existence dependency and generalisation/specialisation only. Only in this way it is possible to ensure the syntactical and semantic integrity of domain models.

Finally, existence dependency is a concept that can perfectly be integrated in existing methods, for example by grafting it onto UML [20] or integrating it into OPEN [12].

ACKNOWLEDGEMENT

This research was done while Monique Snoeck was working in the Department of Applied Economic Siences at the Katholieke Universiteit Leuven, Belgium.

REFERENCES

1. Aho, Alfred V., Ullman Jeffrey D., *The theory of Parsing, Translation and Compiling. Volume I : Parsing,* Prentice Hall, Series in Automatic Computation, 1972, 542 pp.
2. Baeten, J.C.M., *Procesalgebra,* Kluwer programmatuurkunde, 1986
3. Booch, G., *Object Oriented Analysis and Design with Applications*. Second Edition, Benjamin/Cummings, Redwood City, CA, 1994.
4. Chen, P.P., The Entity Relationship Approach to logical Database Design, *QED information sciences,* Wellesley (Mass.),1977
5. Coad P., and Yourdon, E. *Object-Oriented analysis*. Prentice Hall, Englewood Cliffs, N.J., 1991.
6. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. Jeremaes, P., *Object-Oriented Development, The FUSION Method*. Prentice Hall, Englewood Cliffs, N.J., 1994.
7. de Champeaux D., Lea D., Faure P., *Object-Oriented System Development,* Addison Wesley Publishing Company, 1993
8. Dedene G., Snoeck M., M.E.R.O.DE.: A Model-driven Entity-Relationship Object-oriented DEvelopment method, ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 3 (1994) 51-61.
9. Dedene G., Snoeck M., Formal deadlock elimination in an object-oriented conceptual schema, *Data and Knowledge Engineering*, Vol. 15 (1995) 1-30
10. Embley, D.W., Kurtz, B.D., and Woodfield, S.N. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Yourdon Press, Prentice Hall, Englewood Cliffs, N.J., 1992.

11. Harel, D., On visual formalisms, Communications of the ACM, Volume 31, No 5, May 1988, pp. 514-530
12. Henderson-Sellers, B., Graham, I.M., Firesmith, D., Edwards, J.M., Due, R.,Page-Jones, M., Reenskaug, T., Selic, B., Whitehead, K. and Yourdon, E., The OPEN methodology, Object Magazine (Nov 1996)
13. Hoare C. A. R., *Communicating Sequential Processes* (Prentice-Hall International, Series in Computer Science, 1985).
14. Hopcroft John E., Ullman Jeffrey D., *Formal languages and their relation to automata,* Addison Wesley Publishing Company, 1969
15. Jackson, M. A., *System Development*, Prentice Hall Englewood Cliffs (N.J.), 1983, 418 pp.
16. Jacobson Ivar et al., *Object-Oriented Software Engineering, A use Case Driven Approach*, Addison-Wesley, 1992
17. Kilov Haim, Ross James, Information Modeling: an object oriented approach, Prentice Hall, Englewood Cliffs, 1994
18. Kim Won, Bertino Elisa, Garza Jorge F., Composite Objects Revisited, ACM SIGMOD 89, pp. 337-347
19. Milner R., *A calculus of communicating systems* (Springer Berlin, Lecture Notes in Computer Science, 1980).
20. Rational Software Corporation, The Unified Modeling Language, Version 1.0.1, 19 March 1997, http://www.rational.com/uml
21. Rumbaugh, J., Blaha M., Premerlani, W., Eddy, F., Lorensen, W., *Object Oriented Modelling and Design*, Prentice Hall International, 1991
22. Shlaer, S., Mellor, S.J., *Object-Oriented Systems Analysis: Modelling the World in Data*, Prentice Hall, 1988
23. S. Shlaer, S.J. Mellor, *Object Lifecycles: Modeling the World in States* (Prentice Hall, Englewood Cliffs, New Jersey, 1992).
24. Snoeck M., Dedene G., Generalization/Specialization and Role in Object Oriented Conceptual Modeling, *Data and Knowledge Engineering,* 19(2), 1996
25. Snoeck M., *On a Process Algebra Approach to the construction and analysis of M.E.R.O.DE.-based conceptual models*, PhD. Dissertation (Katholieke Universiteit Leuven, Faculty of Science & Department of Computer Science, May 1995)

APPENDIX A

Formal definition of the existence dependency graph

Let $\mathcal{M}$ be the set of object types in the conceptual schema.

The existence dependency graph (EDG) is a relation $\leftarrow$ which is a bag[9] over $\mathcal{M} \times \mathcal{M}$ such that $\leftarrow$ satisfies the following restrictions:

1) An object type is never existence dependent of itself: $\forall P \in \mathcal{M}: (P,P) \notin \leftarrow$
2) Existence dependency is acyclic. This means that:
   $\forall n \in \mathbb{N}, n \geq 2, \forall P_1, P_2, ..., P_n \in \mathcal{M}$:
   $(P_1,P_2), (P_2,P_3),..., (P_{n-1},P_n) \in \leftarrow \Rightarrow (P_n,P_1) \notin \leftarrow$

Formal Definition of the Object Event Table

Let A be the universe of relevant event types. Then

$T \subseteq \mathcal{M} \times A \times \{' ', 'C', 'M', 'E'\}$ such that

$\forall P \in \mathcal{M}, \forall a \in A$:

$(P,a,' ') \in T$ or $(P,a,'C') \in \mathcal{T}$ or $(P,a,'M') \in \mathcal{T}$ or $(P,a,'E') \in T$

$\forall P \in \mathcal{M}: c(P) = \{a \in A \mid (P,a,'C') \in T\}$

$m(p) = \{a \in A \mid (P,a,'M') \in T\}$

$e(P)= \{a \in A \mid (P,a,'E') \in T\}$

c(P), m(P), e(P) are pairwise disjoint

$c(P) \cup m(P) \cup e(P) = S_AP$

$c(P), e(P) \neq \varnothing$

$\cup \{S_AP \mid P \in \mathcal{M}\} = A$

Propagation rule

If P is existence dependent of Q, the alphabet of P must be a subset of the alphabet of Q.

Formally: $P \leftarrow Q \Rightarrow S_AP \subseteq S_AQ$.

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 parent object type.
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 parent object type.
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 parent object type.

Formally: $P \leftarrow Q \Rightarrow c(P) \subseteq c(Q) \cup m(Q)$ and $m(P) \subseteq m(Q)$ and $e(P) \subseteq e(Q) \cup m(Q)$

Contract rule

When two object types share two or more event types, the common event types must be in the alphabet of one or more common existence dependent object types.

Formally: $\forall P, Q \in \mathcal{M} : \#(S_AP \cap S_AQ) \geq 2$ and $\neg(S_AP \subseteq S_AQ$ or $S_AQ \subseteq S_AP)$ $\Rightarrow \exists R_1, R_2, ... R_n \in \mathcal{M}: \forall i \in \{1,...,n\}: R_i \leftarrow P,Q$ and $S_AR_1 \cup ... \cup S_AR_n = S_AP \cap S_AQ$

Formal Definition of scenarios, projection and parallel composition

Let A be the universe of event types.

R(A) is the set of all regular expressions over A where e is a regular expression over A if and only if (a) e = 0 or (b) e = 1 or (c) $\exists a \in A$: e = a or (d) $\exists$ e', e" $\in$ R(A) such that e = e' + e" or e = e'.e" or e = (e’)*

9. Bags can contain the same element more than once (as opposed to sets).

Each regular expression defines a set of scenarios, called its *language*

(1) A^* is the set of scenario's over A. A^* is defined by

(a) $1 \in A^*$

(b) $\forall a \in A: a \in A^*$

(c) Let $s, t \in A^*$, then $s \wedge t \in A^*$

(d) $\forall s \in A^*: 1 \wedge s = s = s \wedge 1$

(2) The regular language of a regular expression is a subset of A^* defined by

$L(1) = \{1\}$

$\forall a \in A : L(a) = \{a\}$

$\forall e, e' \in R^*(A)$: $L(e + e') = L(e) \cup L(e')$, $L(e.e') = L(e).L(e')$, $L(e^*) = L(e)^*$

where $L(e).L(e') = \{ s \wedge t \mid s \in L(e) \text{ and } t \in L(e')\}$

and $L(e)^* = \{1\} \cup L(e) \cup L(e).L(e) \cup L(e).L(e).L(e) \cup L(e).L(e).L(e).L(e) \cup \ldots$

Let $B \subseteq A$. Then

$1\backslash B = 1$

$\forall a \in A : (a\backslash B = 1 \Leftrightarrow a \notin B)$ and $(a\backslash B = a \Leftrightarrow a \in B)$

$\forall s, t \in A^* : (s \wedge t)\backslash B = s\backslash B \wedge t\backslash B$,

$\forall e, e' \in R^*(A) : (e + e')\backslash B = e\backslash B + e'\backslash B$, $(e.e')\backslash B = e\backslash B . e'\backslash B$, $(e^*)\backslash B = (e\backslash B)^*$

Let P, Q be object types

The alphabet of P || Q is the union of the alphabets: $S_A (P \| Q) = S_A P \cup S_A Q$

The behaviour of P || Q is defined by an expression $e'' \in R^*(A)$ such that

$L(e'') = \{ s \in (S_A P \cup S_A Q)^* \mid s\backslash S_A P \in L(P) \text{ and } s\backslash S_A Q \in L(Q)\}$

Formal definition of *more deterministic than*:

$P \leq Q$ if and only if $S_A P \subseteq S_A Q$ and $L(P) \subseteq L(Q\backslash S_A P)$

Alphabet rule:

The structure diagram or expression that defines the behaviour of an object type P must contain all and only the event types for which there is a 'C', 'M' or 'E' in the column of P in the OET.

Formally: $\varphi(S_R P) = S_A P$.

where $\varphi : R^*(A) \rightarrow \mathcal{P}(A): e \rightarrow \varphi(e)$ such that $\varphi(a) = \{a\}$

$\varphi(e + e') = \varphi(e) \cup \varphi(e')$

$\varphi(e . e') = \varphi(e) \cup \varphi(e')$

Default life cycle rule

The events in c(P), m(P) and e(P) must appear as creating event types, modifying event types or ending event type respectively in the sequence restriction of the object type P. This means that the sequence constraints must be more deterministic than the default life cycle of create, modify, end.

Formally: $\forall P \in \mathcal{M}: S_R P \leq (\Sigma\, c(P)).(\Sigma\, m(P))^*.(\Sigma\, d(P))$

Restriction rule

If P is existence dependent of Q, the P must be more deterministic than Q

Formally: $P \leftarrow Q \Rightarrow P \leq Q$

APPENDIX B: CONCEPTUAL SCHEMA FOR PROJECT ADMINISTRATION

The EDP-department of a large company consists of several groups. Each development project has one group responsible for it. People from different groups can be assigned full-time or part-time to the same project. In order to keep track of the development cost of information systems, each member of the development staff has to register the number of hours (s)he worked for a particular project. A person can only register working hours for projects (s)he is assigned to. When a project comes to an end, all assignments are closed as well. Finished projects and closed assignments can be kept for a while for cost analysis purposes.

Fig. 25 shows an ER-schema for this case-study. As explained in section 2.3, all relationships that do not express existence dependency have to be converted to object types existence dependent of the object type participating to the relationship. Fig. 26 shows the resulting existence dependency graph.

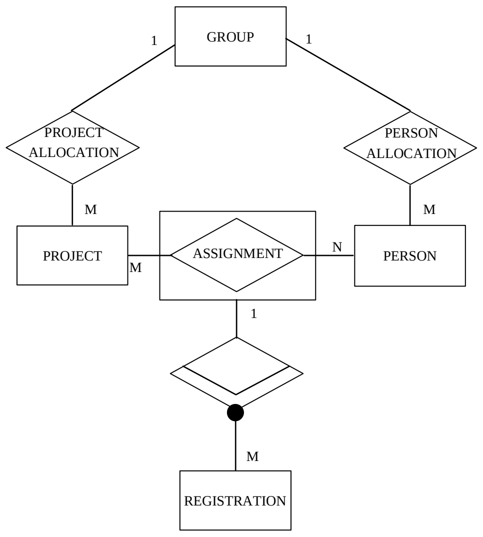


Fig. 25. ER-schema for the project administration

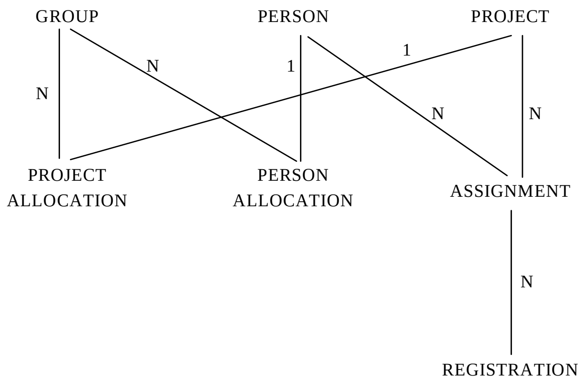


Fig. 26. Existence Dependency Graph for the project administration

	GROUP	PROJECT	PERSON	PROJECT ALLOCATION	PERSON ALLOCATION	ASSIGNMENT	REGISTRA-TION
cr_group	C						
end_group	E						
cr_project		C					
close_project		M	M			M	
end_project		E					
cr_person			C				
end_person			E				
proj_alloc	M	M		C			
proj_dealloc	M	M		E			
pers_alloc	M		M		C		
pers_dealloc	M		M		E		
assign		M	M			C	
close_assign		M	M			M	
end_assign		M	M			E	
register		M	M			M	C
end_registr		M	M			M	E

Fig. 27 OET for the project administration

The Object event table contains one column per object type in de EDG and one row per relevant domain event type. Each object type should have at least one creating and one ending event type. In addition, the Propagation Rule and Type of Involvement Rule tell us how to fill in the OET. Participation to event types has to be propagated from REGISTRATION to ASSIGNMENT, from ASSIGNMENT to PROJECT and PERSON, from PERSON_ALLOCATION to PERSON and GROUP and finally from PROJECT_ALLOCATION to PROJECT and GROUP. Fig. 27 shows the resulting OET.

Finally, sequence constraints can be specified: each object type defines a regular expression that contains exactly these event types in which the object type is involved according to the OET (Alphabet Rule). In addition, the type of involvement for each event type must be respected (Default Life Cycle Rule). Fig. 28 shows possible sequence constraints.

GROUP : cr_group.(proj_alloc + proj_dealloc + pers_alloc + pers_dealloc)*.end_group

PROJECT : cr_project . (proj_alloc+ proj_dealloc + assign + close_assign + end_assign
+ register + end_registr)*.close_project .end_project

PERSON : cr_person .(close_project + pers_alloc + pers_dealloc + assign + close_assign + end_assign +
register + end_registr)* . end_person

PROJECT_ALLOCATION : proj_alloc.proj_dealloc

PERSON_ALLOCATION : pers_alloc.pers_dealloc

ASSIGNMENT : assign.(register + end_registr)* .(close_assign + close_project).(end_registr)*.
end_assign

REGISTRATION : register.end_registr

Fig. 28. Sequence constraints for the project administration

When checking the specifications against the Restriction Rule, an error is found: some scenarios of ASSIGNMENT are not conform to the sequence restrictions imposed by PROJECT. Indeed, the sequence restrictions of PROJECT require each individual scenario to end with a 'close_project' event:

PROJECT\ S_AASSIGNMENT = (assign + close_assign + end_assign + register + end_registr)*.close_project

So the 'close_project' cannot be followed by 'end_registr' or 'end_assign' events as allowed by the sequence restrictions of ASSIGNMENT . Even worse, the specification of PROJECT requires the end_assign event to precede the close_project event, while the specification of ASSIGNMENT requires the opposite: close_project must precede end_assign. The conceptual schema thus contains conflicting sequence restriction, which will result in a deadlock at execution time.

The conceptual schema can be corrected by removing the close_project from the sequence restrictions of ASSIGNMENT. The correct solution defines the sequence restriction of ASSIGNMENT as:

ASSIGNMENT = assign.(register + end_registr)* .close_assign.(end_registr)*. end_assign

As PERSON had acquired the close_project event from ASSIGNMENT because of the Propagation Rule, the close-project event must also be removed from the alphabet of PERSON. The fact that all assignments referring to a project must be ended before that project is closed is in fact already modelled by the sequence restrictions of PROJECT. Indeed, these restrictions say that an "close_assign" can not follow a "close_project" even type. As a result, correct event handling will ensuring that all assignments are ended before a project is closed.