DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

ONDERZOEKSRAPPORT NR 9640

Existence Dependency: Conceptual modelling by contract

by

M. Snoeck

G. Dedene

Katholieke Universiteit Leuven

Naamsestraat 69, B-3000 Leuven

ONDERZOEKSRAPPORT NR 9640

Existence Dependency: Conceptual modelling by contract

by

M. Snoeck

G. Dedene

D/1996/2376/40

Existence Dependency: Conceptual modelling by contract.

Dr. M. Snoeck
Prof. dr. G. Dedene

Katholieke Universiteit Leuven
Dept. of Applied Economic Sciences
Naamsestraat 69, 3000 Leuven, Belgium
Tel. 32 16 32 66 11
Fax. 32 16 32 67 32
Email: {Monique.Snoeck, Guido.Dedene}@econ.kuleuven.ac.be

Abstract

In Object Orientation, the Generalisation/Specialisation hierarchy and the Whole/Part relationship are prevalent classification schemes for object types. This paper presents a new classification scheme for object types, called "existence dependency". Existence dependency captures some of the interesting semantics that are usually associated with the concept of aggregation (Part Of relation). In fact, the semantics of existence dependency are hidden in the semantics of the Entity Relationship model, but have never been explicitly named. We will demonstrate how the explicit classification of object types according to the existence dependency relation allows for formal and automatic consistency checking between static and dynamic aspects of object types that goes far beyond mere syntactical consistency.

1. INTRODUCTION

One of the main tasks in conceptual modelling is the identification of components in the universe of discourse. 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 [2, 4, 6, 9, 15, 17, 19]. Especially the Generalisation/Specialisation lattice is considered crucial to object oriented conceptual modelling due to its ability to reduce the complexity of conceptual schemes. There are of course many other kinds of relationships that can be defined and these are in general captured under the common denominator "associations". This paper presents a new classification principle, called *existence dependency*, that is important enough to deserve special attention and notation. We will demonstrate how existence dependency is present in every data model where at least two object types are somehow related. As a result, existence dependency can be considered even more fundamental than the "A Kind Of" and "A Part Of" associations.

However, the most important motive for the explicit modelling of this relation is its ability to allow for quality control at a very high level. In object oriented conceptual modelling static and dynamic aspects of the Universe of Discourse are modelled with equal emphasis. Usually different techniques are used to capture different aspects. For example, a large number of Object Oriented Analysis (OOA) methods use an Extended Entity Relationship-like technique and "A Kind Of"- and "A Part Of"-lattices for specifying static aspects. Finite State Machines and Event Trace Diagrams are used for capturing dynamic aspects. Some of these techniques have overlapping semantics. This means that the same aspects may be modelled several times in different schemes. For example, the "A Kind Of"-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 inherit the state machine of the generalisation ?
- Can it refine this state machine by adding, removing or redefining states, transitions or events ?
- Can it restrict the behaviour of the generalisation or extend it or both ?
- Are the events of the specialisation specialisations of the events of the generalisation ?
- Can a specialisation 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 [20]. This definition violates the broadly accepted notion of inheritance where subtypes inherit data *and behaviour* of their supertype.

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 [21] 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.

The paper is organised as follows. The next section defines the concept of Existence Dependency and motivates the statement that this concept is present in every data model or object model. We will then proceed to the formal definition of a conceptual model with an explicit existence dependency relation and demonstrate how this relation can be used as a starting point for consistency checking between static and dynamic aspects of object types. Finally Existence Dependency is compared to Generalisation/Specialisation and to the concept of aggregation.

2. ON THE UBIQUITY OF EXISTENCE DEPENDENCY

2.1. Existence Dependency: Definition

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 destroyed. 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 <- 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 *marsupial* (P is the marsupial object type) and is existence dependent of q, called the *mother* (Q is the mother object type).

Example 1

The life span of a loan of a book is always embedded in the life span of the book that is on loan. Hence the object type LOAN is existence dependent of the object type BOOK

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.

2.2. The Entity Relationship model and Existence Dependency

Existence Dependency is a key concept in the Entity Relationship model. P.P. Chen uses the concept of existence dependency in his definitions of "weak entity type" and "ID dependency" ([3], p. 24):

Weak entity type. The existence of a weak entity depends on the existence of other entities. The 'E' in the relationship box indicates that it is an "existence dependent" relationship.

ID dependency. In case of ID dependency, entities are identified by their relationships with other entities.

The weak entity type is thus existence dependent of the other entity types involved in the existence dependent relationship. ID dependency implies existence dependency, but the reverse is not necessarily true. Fig. 1. gives an example of a weak entity type and an ID-dependent entity type.

Example 2

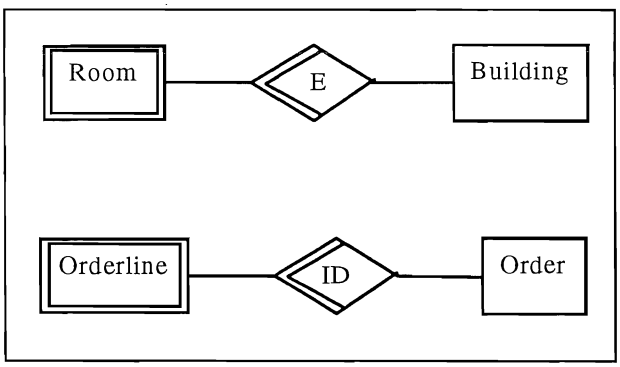

Fig. 1. Weak entity type and ID dependent entity type.

ROOM is a weak entity type, existence dependent on BUILDING. Indeed, the existence of a room depends on the existence of a building. ORDERLINE is ID dependent of ORDER because an orderline is identified by means of the order it is part of. In both cases the relationship that connects the two entity types is a *weak relationship*[1].

Note that ID-dependency is in fact a matter of attribute and primary key definition. One can perfectly choose to identify orderlines by means of a unique number, not including the order identification. As a result, in a non-attributed Entity Relationship schema, the distinction between ID-dependency and Weak Entity types is irrelevant.

The semantics of the ER-model have been defined by means of insertion, deletion and updating rules [4, 9, 10]. The first definitions [4, 9] were quite imprecise in that they did not make a clear difference between mandatory (or total) and weak relationships. A total relationship implies some 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 inserted if at the same time a relationship occurrence is inserted that relates this EMPLOYEE-occurrence to some PROJECT-occurrence. Reversely, 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, EMPLOYEE is existence dependent of PROJECT. As pointed out in [10, 17], an important difference between a mandatory and a weak relationship lies in the updating rules. In the next example (Fig. 2) in case (a) an ORDERLINE is always connected to exactly one *and always the same* ORDER. In case (b) the relationship is not weak, but, as indicated by the black dot, participation to the relationship is mandatory for ORDERLINE. This implies that the existence of an ORDERLINE depends on the existence of *an* ORDER. But it is not necessarily the same order throughout the whole lifetime of the ORDERLINE. As a result, ORDERLINES can be moved to an other ORDER if necessary.

1. In [9] a difference is made between a weak relationship and a weak entity, but semantics are very unprecise. A later publication [10] provides more precise semantics and defines that a *weak relationship* connects a *weak entity* to a strong entity.

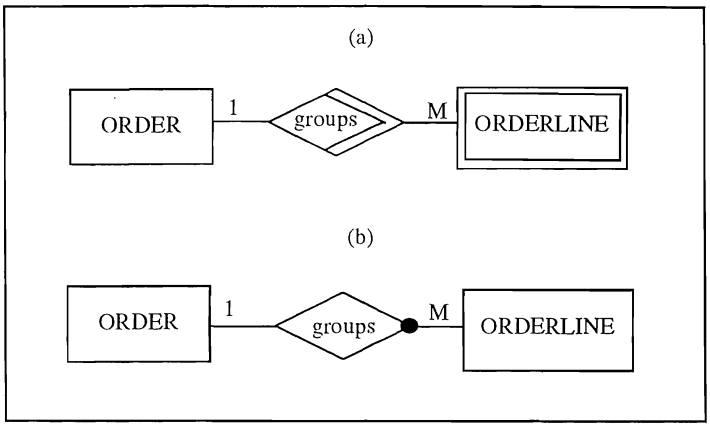

Fig. 2. Orders and Orderlines

Although not mentioned as such, the concept of existence dependency is also present in the concept of regular relationships. Relationships can be considered as object types on their own and in the definition of P.P. Chen, relationships are identified by using the identifiers of the entities involved in the relationship [3]. By definition, if relationships are considered as object types, they are ID dependent (and thus existence dependent) of the involved entity types.

More in general, the concept of relationships and the insertion, deletion and updating rules [4, 9, 10] imply the following three assumptions:

Assumption 1: The relationship cannot be created if the participating entities do not exist. A relationship is thus always created after (or at the same time) that the participating entities are created.

Assumption 2: The relationship exists at most as long as the participating entities exists. It can be destroyed earlier but not later than the involved entities.

Assumption 3: A relationship is always connected to the same entities. This means that if a connection is changed (at the level of occurrences) we have a new relationship object[2].

In fact, these assumptions are equivalent to the statement that when relationships are considered as objects, they are existence dependent of the entities they connect. This is illustrated by means of a library example.

Example 3.

In a library, books can be borrowed by members. Fig. 3 depicts a possible object type diagram for a small library. According to assumption 1 and 2, a loan relationship can only be created if the corresponding book exists and a loan-object cannot continue to exist if the book to which it refers is destroyed. According to assumption 3, as long as a loan exists, it always refers to the same book. Indeed, a different book implies a different loan. The life of a loan object thus is always embedded in the life of one particular book object. Therefore, the object type LOAN is existence dependent of the object type BOOK. The same line of reasoning applies to LOAN and MEMBER and thus the object type LOAN is existence dependent of the object type MEMBER as well.

2 . This follows from the fact that relationships are viewed as tuples [4] and are identified by means of the identifiers of the participating entities.

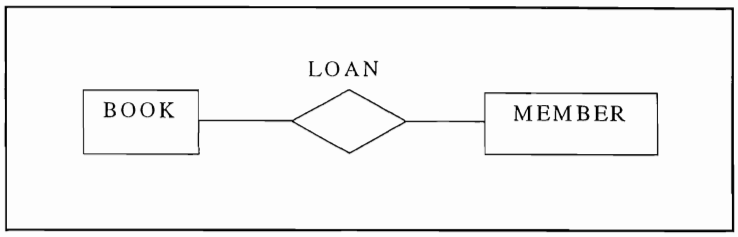

Fig. 3. Library administration

3. EXISTENCE DEPENDENCY: THE KEY TO CONSISTENCY CHECKING IN CONCEPTUAL SCHEMES

This section formally defines a conceptual schema with an explicit existence dependency graph. The conceptual model presented here consists of four submodels: an entity-relationship model, an existence dependency graph (EDG), a behaviour model and an object-event table (OET). Consistency checking between the four schemes is achieved by ensuring consistency between the EDG and each of the other three subschemes. In addition, some consistency checking between the OET and the behaviour schema must be done. The first paragraph of this section defines the existence dependency graph (EDG). The next three paragraphs define the ER-schema, the object event table (OET) and the behaviour schema respectively. For each of these schemes it is explained how consistency with the EDG is ensured. In addition, behaviour must also be checked for consistency with the OET.
Fig. 1 gives an overview of the necessary consistency checking steps.

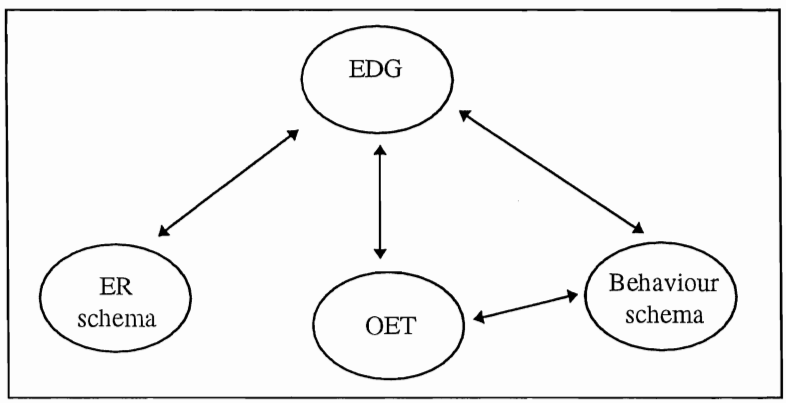

Fig. 4. Consistency checking between the four subschemes

3.1. The Existence Dependency Graph

Basic Definition

The existence dependency (ED) relation is a partial ordering on object types as defined in Definition 1. 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 <- which is a bag[3] over $\mathcal{M} \times \mathcal{M}$ such that <- satisfies the following restrictions:

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

<-- is the non-reflexive transitive closure of <- :

<-- $\subseteq \mathcal{M} \times \mathcal{M}$ such that 1) $\forall P, Q \in \mathcal{M}: (P,Q) \in <- \Rightarrow (P,Q) \in <--$

2) $\forall P, Q, R \in \mathcal{M}: (P,Q) \in <-$ and $(Q,R) \in <-- \Rightarrow (P,R) \in <--$

<- is a *bag* over $\mathcal{M} \times \mathcal{M}$ because an object type can be existence dependent from the same object type in two different ways. For example, the existence of a marriage[4] depends both on the existence of a husband and on the existence of a wife and thus (MARRIAGE, PERSON) will be twice in <-.

GRAPHICAL REPRESENTATION

A mother object type is placed above the marsupial object type and the two are connected with a line. Example 4 is represented in Fig. 5.

Example 4

In a library environment, the entity type BOOK denotes the abstract concept of a book. The library can possibly keep many physical COPIES of a single book. MEMBERS can borrow COPIES. 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

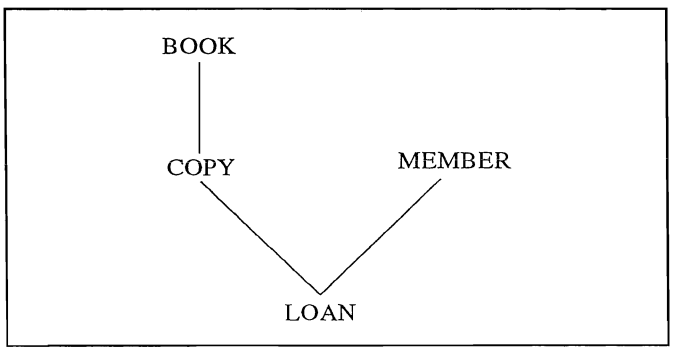

Fig. 5. Existence Dependency Graph for the library

---

3. Bags can contain the same element more than once (as opposed to sets).
4. Many designers would define marriage as a relationship rather than as an object type. But as will be seen in the next section, relationships are always considered as object types, unless they are weak.

Cardinality of Existence Dependency

The existence dependency graph also defines the cardinality of the existence dependency relation. This cardinality defines how many occurrences of the marsupial object type can be dependent of one mother 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*[5].

The precise semantics of these constructs can only be defined in a formal way by means of the object level and have been established in the definition of an instantiation of a conceptual schema in [7, 22]

GRAPHICAL REPRESENTATION

The cardinalities are written next to the line that connects the mother and the marsupial object type.

Example 5

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:

LOAN (1)<- COPY and LOAN (n)<- MEMBER

COPY (n)<- BOOK

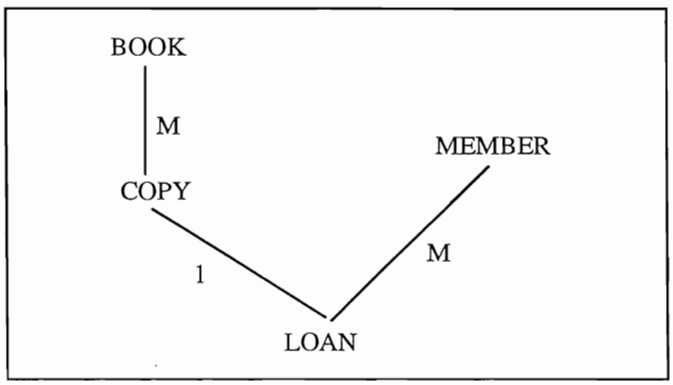

Fig. 6. EDG with cardinalities for the library

5. Note that the clause "at one point in time" is essential in the definition of the cardinalities. Over time, most objects can have many existence dependent objects.

3.2. The ER-model

Basic definition

As pointed out by Webre [23], in spite of the many publications on the ER-model, it is still underdefined. This section defines what is understood by a syntactically correct ER-schema. The semantics will be defined by relating the components of the ER-schema with the Existence Dependency Graph. The definition of the OET and the sequence restrictions will provide the concept of Existence Dependency with operational semantics. For semantics in terms of (temporal) functional dependencies, we refer to [24].

The ER-schema defines entity types and relationship types. In order to allow aggregated relationships, the formal definition makes no difference between entity types and relationship types: every relationship type is possibly an entity type as well. The term association is used to denote the connection between a relationship type and an entity type.

Definition 3
An ER-schema is a tuple $(\mathcal{ER}, \mathcal{A}, \mathcal{W})$ where $\mathcal{ER}$ is a set of entity types and relationship types, $\mathcal{A}$ is a set of associations and $\mathcal{W} \subseteq \mathcal{A}$ is a set of weak associations.

Two functions are defined that determine the elements which are related by an association:
from: $\mathcal{A} \rightarrow \mathcal{ER}$: a $\rightarrow$ from(a)
to: $\mathcal{A} \rightarrow \mathcal{ER}$: a $\rightarrow$ to(a)

E is an *entity type* $\Leftrightarrow \neg(\exists$ a $\in \mathcal{A}$: from(a) = E)
R is a *relationship type* $\Leftrightarrow \exists$ a $\in \mathcal{A}$: from(a) = R)
A relationship type R is *aggregated* $\Leftrightarrow \exists$ a $\in \mathcal{A}$: to(a) = R
A relationship type R is *weak on E* $\Leftrightarrow \exists$ a $\in \mathcal{W}$: from(a) = R and to(a) = E

The following restrictions hold for a valid ER-schema

1) Every association relates exactly two distinct elements of $\mathcal{ER}$:
   $\forall$ a $\in \mathcal{A}$: from(a), to(a) $\in \mathcal{ER}$ and from(a) $\neq$ to(a)

2) Every relationship type relates at least two (not necessarily distinct) elements:
   $\forall$ R $\in \mathcal{ER}$, a $\in \mathcal{A}$: [from(a) = R $\Rightarrow \exists$ b $\in \mathcal{A}$: from(b) = R]

3) A weak relationship type cannot be aggregated:
   $\forall$ R $\in \mathcal{ER}$: [$\exists$ a $\in \mathcal{W}$: from(a) = R $\Rightarrow \neg(\exists$ b $\in \mathcal{A}$: to(b) = R)]

4) A weak relationship type is weak on at most one element:
   $\forall$ R $\in \mathcal{ER}$: [$\exists$ a $\in \mathcal{W}$: from(a) = R $\Rightarrow \neg(\exists$ b $\in \mathcal{W}$: from(b) = R)]

GRAPHICAL REPRESENTATION
Entity types are drawn as rectangles, relationship types as diamonds, aggregated relationship types as diamonds in a rectangle and weak relationship types as diamonds with a double line on the side of the weak entity. Associations are drawn as lines between the elements they relate. The graphical representation of Example 6 is given in Fig. 7.

Example 6

An ER-schema for the library example would be as follows:

$\mathcal{ER}$ = {LOAN, BOOK, MEMBER, COPY, IS_OF}

$\mathcal{A}$ = {borrows, borrowed_by, of, has}

$\mathcal{W}$ = {of}

from(borrows) = LOAN

from(borrowed_by) = LOAN

to(borrows) = MEMBER

to(borrowed_by) = COPY

from(of) = IS_OF

to(of) = COPY

from(has) = IS_OF

to(has) = BOOK

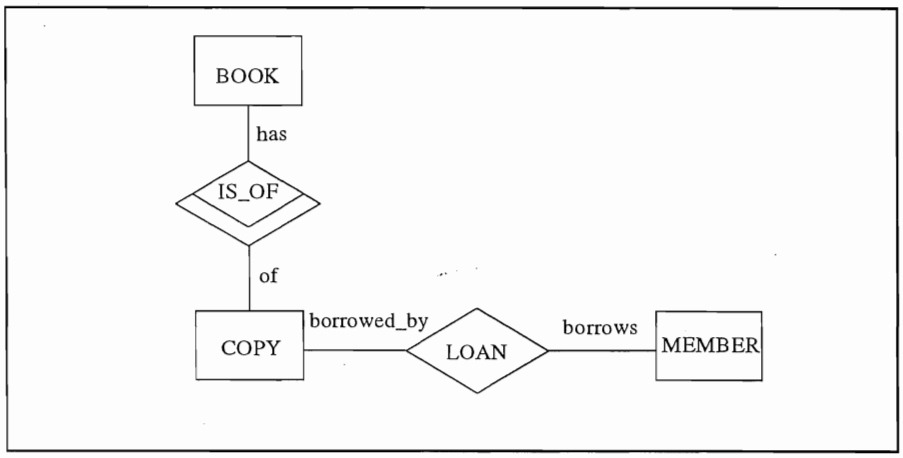

Fig. 7. ER-schema for the library example

Cardinality and Optionality of Associations

Let R be a relationship type that relates the entity type E_1, ..., E_n. This means there are n associations a_1, ..., a_n such that $\forall$ i $\in$ {1, ...,n} from(a_i) = R and to(a_i) = E_i. The association a_i has *cardinality one* if and only if an occurrence of (E_1, ..., E_{i-1}, E_{i+1}, ... E_n) can be associated with at most one occurrence of E_i. In the other case the association a_i has *cardinality many*. In Example 6, a copy can be associated with at most one member and thus the association 'borrows' has cardinality one. But a member can borrow many copies and thus the cardinality of the association 'borrowed_by' is many.

The optionality of an association indicates whether the association is mandatory or not. If an association a is *mandatory*, this means that each occurrence of to(a) must be associated with *at least one* occurrence of from(a). In the other case the association is optional. For example, the association 'of' is mandatory because a copy is always the copy of a book, i.e. a copy is always associated with an occurrence of the relationship IS_OF. The association 'has' is optional because a book might not have a corresponding copy. Note that the semantics of existence dependency imply that a weak association is always mandatory.

GRAPHICAL REPRESENTATION

The cardinality of one is represented by a '1' and a cardinality of many is indicated by a 'N' or 'M' next to the line of the association. When an association is mandatory this is indicated by a black dot on that side of the relationship. Fig. 8 gives the cardinalities and optionalities for Example 6.

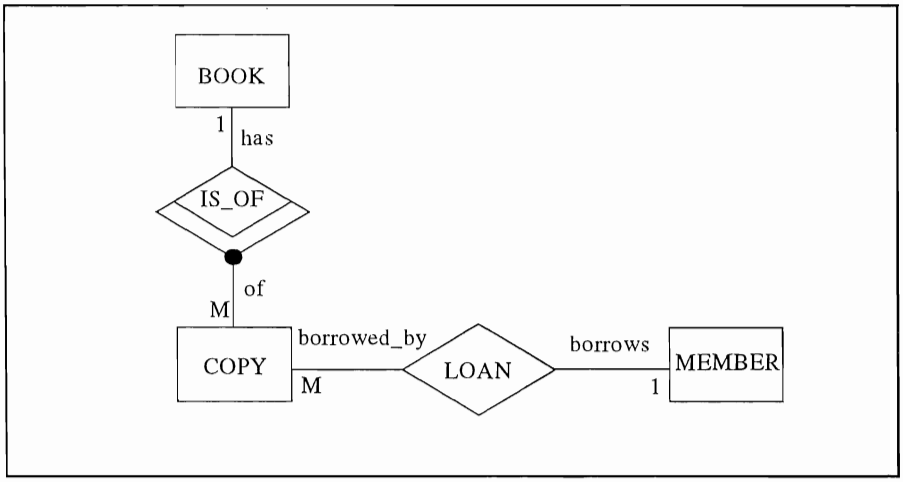

Fig. 8. ER-schema with cardinalities and optionalities

The cardinalities of a ER-schema can be different depending on the time considerations that are taken for information modelling (see next paragraph).

ER-schema versus Existence Dependency graph

As explained in the second section of this paper, the semantics of the existence dependency graph are a subset of the semantics of the ER-schema. As a result, the existence dependency partial ordering can be derived from the ER-schema by means of the following rules:

- Each relationship type R is existence dependent on the participating entity types.
- A relationship type R can be weak on a participating entity type E. According to the semantics of the ER-model this means that E is existence dependent on R. As R is in turn existence dependent on E (previous rule), the life cycles of E and R will be equal. For this reason, R is dropped from the existence dependency graph. The advantage of doing so, is that the ED relation can be defined as a partial order and the ED graph as a Directed Acyclic Graph[6]. When a relationship type R is weak on an entity type E, E is said to be existence dependent on the other participating entity types.

ER and ED Consistency rule

Let ($\mathcal{ER}$, $\mathcal{A}$, $\mathcal{W}$) be an ER-schema and let $\mathcal{M}$ be the set of object types occurring in the EDG. The ER-schema and the EDG are consistent with each other if and only if $\mathcal{M}$ contains all entity types and non-weak relationship types of $\mathcal{ER}$ and the existence dependency relation reflects the relationships as defined in the ER-schema:

$\mathcal{M} = \mathcal{ER} \setminus \{ R \mid \exists b \in \mathcal{W}: \text{from}(b) = R \}$

and

6. Acyclicity is required in order to simplify the deadlock checking procedure [7, 22].

<- is a bag over $\mathcal{M} \times \mathcal{M}$ such that $\forall$ P, Q $\in \mathcal{ER}$:
(P,Q) $\in$ <- $\Leftrightarrow$
P is a non-weak relationship to which Q participates:
$\exists$ a $\in \mathcal{A} \setminus \mathcal{W}$: from(a) = P and to(a) = Q and $\neg(\exists$ b $\in \mathcal{W}$: from(b) = P)
or
there exists a relationship R between P and Q that is weak on P:
$\exists$ a $\in \mathcal{W}$, $\exists$ b $\in \mathcal{A} \setminus \mathcal{W}$, $\exists$ R $\in \mathcal{ER}$: from(a) = R and
to(a) = P and from(b) = R and to(a) = Q

Using this consistency rule to derive an EDG from the ER-schema in Fig. 7 we obtain the following:

Example 7
$\mathcal{M}$ = {LOAN, BOOK, MEMBER, COPY}
IS_OF is not in $\mathcal{M}$ because from(of) = IS_OF and of $\in \mathcal{W}$

The ER-schema further implies the following existence dependencies:
LOAN <- MEMBER and LOAN <- COPY
because LOAN is a non-weak relationship to which MEMBER and COPY participate;
COPY <- BOOK
because copy and book are related by a weak relationship:
of $\in \mathcal{W}$, has $\in \mathcal{A} \setminus \mathcal{W}$, from(of) = IS_OF, to(of) = COPY, from(has) = IS_OF and to(has) = BOOK

This is exactly the EDG of Example 4.

For binary relationship, the cardinalities of the EDG are generally the same as those that are usually modelled in the ER-schema, provided that this ER-schema reflects reality at one point in time. In other words, it must be a *time-sliced* ER-schema. Due to particular information needs and the resulting database design, cardinalities of the ER-schema can be different from the cardinalities of the existence dependency graph.

Example 8
MEMBERs can borrow BOOKS. At a particular point in time a BOOK can be borrowed by at most one MEMBER (Fig. 9 (b)). However, if a history of LOAN is kept to cover information needs, the ER-schema will be as in Fig. 9 (a) because *over time* a book can be borrowed many times. In both cases the cardinality of the existence dependency relation between BOOK and LOAN is as in Fig. 9 (c).

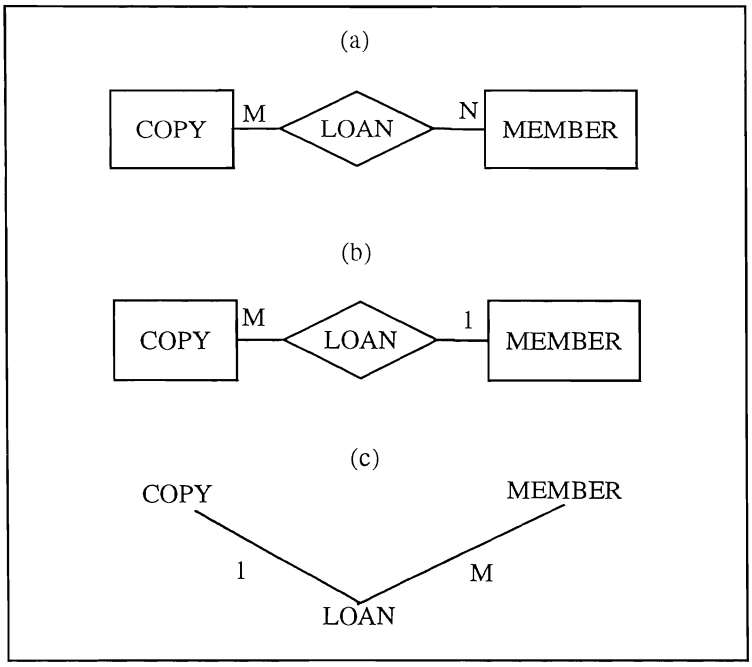

Fig. 9. cardinalities in the ER-schema and the EDG

Although it is clear that the semantics of the existence dependency relationship are part of the semantics of the ER-model, the reverse is not true. An existence dependency graph that classifies object types according to the existence dependency relation cannot capture as much semantics as the ER-model. Fig. 10 shows two ER-diagrams that give rise to the same Existence Dependency graph (Fig. 10(c)). In a hotel room reservation system, a customer makes a reservation for a room type. In Fig. 10(a) a reservation is defined as an existence dependent entity type with two weak relationship types. In this case, a single customer can make several reservations for the same room type. Each reservation uniquely identifies a customer and a room type. In Fig. 10 (b) the reservation is defined as a relationship type between customer and room type. As a result, also in this schema a reservation is an object type that is existence dependent of room type and customer. But Fig. 10 (b) models the additional constraint that a customer can be linked only once to a specific room type: there is only one reservation for a pair (customer, room type) at one point in time. In addition to the fact that each reservation uniquely identifies a customer and a room type, now a (customer, room type) pair also uniquely identifies a reservation.

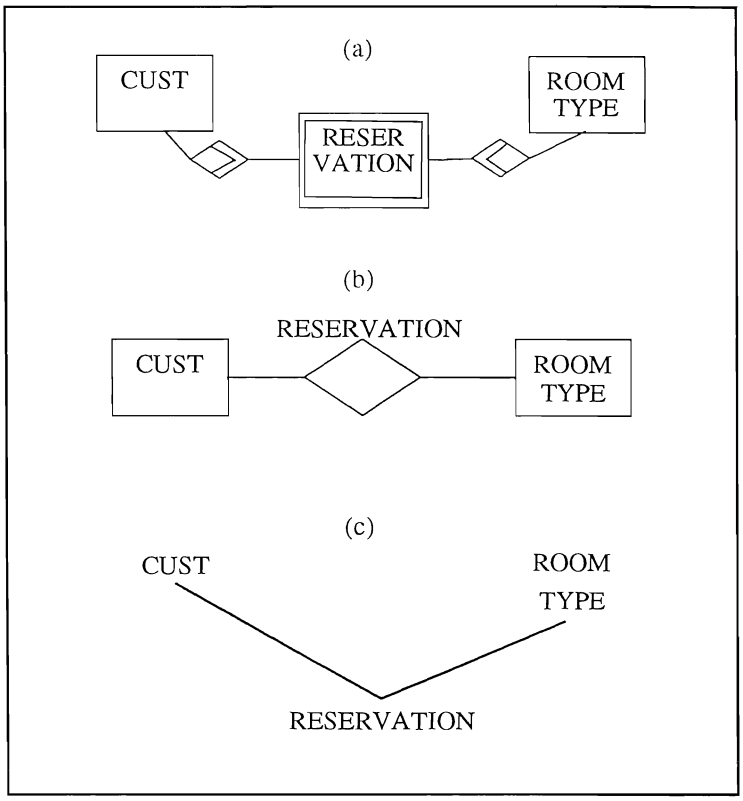

Fig. 10. Hotel room reservation.

3.3. The Object Event Table

Formal Definition

The dynamic schema defines event types and object behaviour. The definition of event types results in a universe A of event types that are relevant for the Universe of Discourse. The Object Event Table 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 A 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 $S_A P$ is partitioned in three mutually disjoint sets: {c(P), m(P), d(P)} is a partition of $S_A P$ with

c(P) = {$a \in A \mid a$ is an event type that creates an occurrence of type P} $\subseteq S_A P$
m(P) = {$a \in A \mid a$ is an event type that modifies an occurrence of type P} $\subseteq S_A P$
d(P) = {$a \in A \mid a$ is an event type that destroys an occurrence of type P} $\subseteq S_A P$
c(P) and d(P) never are empty; m(P) possibly is.

Definition 4

The object-event table $\mathcal{T}$ is a table with one row per event type in A and one column per object type in $\mathcal{M}$. Each cell contains a blank, 'C', 'M' or 'D' such that:
$\mathcal{T} \subseteq \mathcal{M} \times A \times \{' ', 'C', 'M', 'D'\}$ such that
$\forall P \in \mathcal{M}, \forall a \in A$:
(P,a,' ') $\in \mathcal{T}$ or (P,a,'C') $\in \mathcal{T}$ or (P,a,'M') $\in \mathcal{T}$ or (P,a,'D') $\in \mathcal{T}$

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

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

$d(P) = \{a \in A \mid (P,a,'D') \in \mathcal{T}\}$

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

$c(P) \cup m(P) \cup d(P) = S_A P$

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

Each event type must be relevant for at least one object type:

$\cup S_A P = A$

GRAPHICAL REPRESENTATION

The OET is drawn as a matrix containing one row per event type and one column per objet type. A 'C', 'M' or 'D' on a row-column point of intersection indicates that this particular event type is an element of respectively c(P), m(P) or d(P), where P is the object type corresponding to the column. Fig. 11 is a graphical representation of Example 9

Example 9

For the library example the universe of event types is

A = {enter, leave, acquire, catalogue, borrow, renew, return, sell, lose}

The partitions of the alphabets of the object types BOOK, MEMBER and LOAN are as follows:

S_ABOOK = {acquire, catalogue, borrow, renew, return, sell, lose}
with
c(BOOK) = {acquire}
m(BOOK) = {catalogue, borrow, renew, return}
d(BOOK) = {sell, lose}

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

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

	MEMBER	BOOK	LOAN
enter	C		
leave	D		
acquire		C	
catalogue		M	
borrow	M	M	C
renew	M	M	M
return	M	M	D
sell		D	
lose	M	D	D

Fig. 11. OET for the library example

Existence dependency graph versus Object Event Table

The static and dynamic schema 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 ED graph there is one column in the Object Event Table (and vice versa). An additional requirement is that a mother object type has to participate in all event types in which one of its marsupial object types participates:

Propagation rule

$P <\text{-} Q \Rightarrow S_A P \subseteq S_A Q$.

This can be explained as follows. Existence dependent objects can not participate in any event without the mother object having knowledge of this event. By including the alphabet of the marsupial object type in the alphabet of the mother object type, all possible places for information gathering and constraint definition are identified, which does not mean that all object-event routines will have a meaningful content. For example, the *borrow* method of the class MEMBER is the right place to count the number of books a member has in loan and to check a rule such as 'a member can have at most 5 books 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 lent 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'. At implementation time, empty methods can be removed to increase efficiency.

In addition, by including the event types of the marsupials in the alphabet of the mother, sequence restrictions that concern event types of different marsupials can be specified as part of the behaviour of the mother. 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. They can be specified as part of the behaviour of a common mother of these object types, COPY in the given example.

An additional restriction can be put on the subsets of the alphabet: a marsupial cannot be created before its mother exists nor can it exist after its mother has been destroyed:

Type of involvement rule

$P <\text{-} Q \Rightarrow c(P) \subseteq c(Q) \cup m(Q)$ and $m(P) \subseteq m(Q)$ and $d(P) \subseteq d(Q) \cup m(Q)$

The propagation rule and type of involvement rule together ensure that the life span of the marsupial is embedded in the life span of the mother.

The propagation rule has as a consequence that the alphabet of a relationship type always is a subset of the alphabet of the entity 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 marsupial object type that has the role of a 'contract'[7]. Possibly, the shared event types can be spread across more than one existence dependent object type, such that:

Contract rule

$\forall P, Q \in \mathcal{M}: \#(S_A P \cap S_A Q) \geq 2 \Rightarrow \exists R_1, R_2, \ldots R_n \in \mathcal{M}$:
$\forall i \in \{1,\ldots,n\}: R_i <\text{-} P,Q$ and $S_A R_1 \cup \ldots \cup S_A R_2 = S_A P \cup S_A Q$

Common event types between objects thus always indicate the presence of at least one relationship between these objects.

---

7. The notion of contract will be further elaborated when talking about sequence restrictions.

3.4. The Behaviour Schema

A complete definition of behaviour contains more than participation in events only. First, each object type can impose sequence restrictions on the event types in its alphabet by means of a Finite State Machine, a Jackson Structure diagram [14] or a regular expression. From a mathematical point of view, these three techniques are equivalent [7, 22]. Secondly, events are the basis for interaction between objects. The conceptual schema proposed in this paper has no separate object interaction model: for the purpose of conceptual modelling we use communication by means of common event types rather than communication by means of message passing. For example, assume an object type MEMBER, an object type COPY and an event type *borrow*. In stead of specifying a message from COPY to MEMBER (or inversely from MEMBER to COPY) that is triggered by a borrow event, both object types are provided with a method called 'borrow' and it is agreed that object types have to synchronise on common events. This way of communication is similar to communication as defined in the process algebras CSP [12] and ACP [1]. Message passing is more similar to CCS [16].

Consistency checking between the EDG and the behaviour schema is not possible without a rigorous definition of the concepts in use. For the formalisation of sequence restrictions and communication by means of common event types, we refer to the process algebra of M.E.R.O.DE.[8], an Object Oriented Analysis method that explicitly deals with existence dependency. The basic definitions are summarised in the following paragraph (see [7, 22] for a complete set of definitions).

Basic definitions of the M.E.R.O.DE.- process algebra

If we assume a universe A of relevant event types in the Universe of Discourse, the set of services (methods) delivered by an object type is a subset of A. This subset is also called the alphabet of the object type. Object types are allowed to impose sequence restrictions on the event types in their alphabet by means of a regular expression, a Finite State Machine or a JSD diagram, which are three mathematically equivalent formalisms. As a result, object types can be seen as tuples over $<\mathcal{P}(A), R^*(A)>$, where $R^*(A)$ is the set of all possible regular expressions over the universe of event types A. Formally:

Definition 5

$R^*(A) = \{e \mid e$ is a regular expression over A$\}$ where e is a regular expression over A if and only if

(a) $e = 1$ or

(b) $\exists\, a \in A: e = a$ or

(c) $\exists\, e', e'' \in R^*(A)$ such that $e = e' + e''$ or $e = e'.e''$ or $e = (e')^*$

The symbol '1' stands for 'do-nothing'.
In fact, iteration is a meta-syntactical operator as it can be defined by means of selection and sequence:

Definition 6

$$e^* = \sum_{i \in \mathbb{N}} e^i \quad \text{where } e^0 = 1,\ e^1 = e \text{ and } \forall\, n \in \mathbb{N},\ n \geq 2: e^n = e.e^{n-1}$$

The definition of a set of axioms for the selection and sequence operators and a motivation and discussion of these laws can be found in [7, 22].

These definitions can now be used to define the concept of object type:

---

8. Model driven Entity-Relationship Object oriented DEvelopment

Definition 7

Let $\alpha \subseteq A$, $e \in R^*(A)$, then $<\alpha,e>$ is called a tuple over $<\mathcal{P}(A), R^*(A)>$

An object type P is a tuple $<\alpha,e>$ over $<\mathcal{P}(A), R^*(A)>$

Notation: if $P = <\alpha,e>$ then $S_A P = \alpha$, $S_R P = e$

Example 10

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

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

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).

As explained in the introduction, interaction between object types is modelled by means of common event types. The concurrency operator || expresses the fact that object types have to synchronise on common event types. Rather than giving an axiomatic definition, the parallel-operator is defined by means of sets of accepted sequences of event types. Every regular expression defines a Regular Language, this is a set of scenarios over A. A scenario (or sentence) over A is a finite sequence of event types from A, where '^' acts as the concatenation operator and 1 as the empty scenario. A* is the set of all possible scenarios over A.

Definition 8

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

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

The language of an object type is the language of its regular expression: $L(<\alpha,e>) = L(e)$

In order to compare the scenarios of two different object types relative to the common events, a projection operator \B, with $B \subseteq A$, is defined as follows:

Definition 9

Let $B \subseteq A$. Then

$1\backslash B = 1$

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

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

Example 11

Suppose the library example is extended with an object type LOAN that is defined as follows:

LOAN = <{borrow, renew, return, lose}, borrow.(renew)*.(return + lose)>

According to the definition given in Example 10, the following is a possible scenario for the object type COPY:

acquire^classify^borrow^renew^return^declassify^remove_copy

Looking at this scenario of COPY from the point of view of LOAN results in the following:

1^1^borrow^renew^return^1^1 = borrow^renew^return

which is a scenario that is perfectly acceptable from the point of view of LOAN.

When two object types run concurrently, only those scenarios are valid where both object types agree on the sequence of common event types:

Definition 10

Let P, Q ∈ <$\mathcal{P}$(A), R*(A)>
P || Q = <S_AP ∪ S_AQ, e"> with e" ∈ R*(A) such that
L(e") = { s ∈ (S_AP ∪ S_AQ)* | s\$S_A$P ∈ L(P) and s\$S_A$Q ∈ L(Q)}

The regular expression e" always exists as is proved by the theory on Finite State Machines [13, theorem 8-7, p. 252]. This ||-operator can be used to calculate the behaviour defined by composite conceptual schemes [7, 22] as illustrated in the next example.

Example 12

Suppose we have the following definition for the object types BOOK and LOAN:
BOOK = <{acquire, catalogue, borrow, renew, return, sell, lose},
acquire.catalogue.(borrow + renew + return)*.(sell + lose)>
LOAN = <{borrow, renew, return, lose},borrow.(renew)*.(return + lose)>

The behaviour of a book that can be on loan zero, one or more times consecutively is:

S_R(BOOK || (LOAN)*)
= S_R(BOOK || <{borrow, renew, return, lose},(borrow.(renew)*.(return + lose))*>)
= acquire.catalogue.[borrow.(renew)*.return]*.[sell + borrow.(renew)*.lose]

In order to be able to compare the sequence restrictions of two object types, the following order on regular expressions is defined:

Definition 11

∀ e, e' ∈ R*(A) define $e \leq e' \Leftrightarrow e + e' = e'$

Property 1

$\leq$ is a partial order on R*(A)

Proof

$\forall$ e,e',e'' $\in$ R*(A) :

1. $e \leq e$ because $e + e = e$
2. $e \leq e'$ and $e' \leq e \Rightarrow e = e'$
   because $e \leq e' \Rightarrow e + e' = e'$ and $e' \leq e \Rightarrow e' + e = e$
   Thus $e = e' + e = e + e' = e'$
3. $e \leq e'$ and $e' \leq e'' \Rightarrow e \leq e''$
   because $e \leq e' \Rightarrow e + e' = e'$ and $e' \leq e'' \Rightarrow e' + e'' = e''$
   Thus $e + e'' = e + (e' + e'') = (e + e') + e'' = e' + e'' = e''$ ■

The regular expression of an object type determines a set of scenarios (sequences of events) that are accepted by this object type. Intuitively, $e \leq e'$ means that the set of scenarios defined by e' includes the set of scenarios defined by e. In other words, e' accepts all the scenarios of e. In general, object types have different alphabets. In order to analyse the restrictions imposed by object types on their *common* event types only, we need a projection operator that drops irrelevant event types from the behaviour description:

Definition 12

Let $B \subseteq A$, $a \in A$, e, e' $\in$ R*(A). Then define

$1\backslash B = 1$

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

$(e + e')\backslash B = e\backslash B + e'\backslash B$, $(e.e')\backslash B = e\backslash B \,.\, e'\backslash B$ and $((e)^*)\backslash B = (e\backslash B)^*$

This partial order on regular expressions can be used to define a partial order (or taxonomy) on processes:

Definition 13

Let P, Q $\in$ $<\mathcal{P}(A), R^*(A)>$ then define

$$P \leq Q \Leftrightarrow S_A P \subseteq S_A Q \text{ and } S_R P \leq [(S_R Q)\backslash (S_A P)]$$

Property 2

$\leq$ is a partial order on object types.

Proof. This follows from the fact that $\leq$ is a partial order on regular expressions and the fact that $\subseteq$ is a partial order on sets.■

Formal definition of the sequence restrictions schema

Definition 14

The dynamic schema defines sequence restrictions for each object type such that $\forall P \in \mathcal{M}$:
the sequence restrictions are a regular expression over A: $S_R P \in R^*(A)$

Example 13

For the library example the sequence restrictions are as follows:

S_RBOOK = acquire.catalogue.(borrow + renew + return)*.(sell + lose)

S_RMEMBER = enter.(borrow + renew + return + sell + lose)*.leave

S_RLOAN = borrow.(renew)*.(lose + return)

GRAPHICAL REPRESENTATION

When represented in a mathematical way, the sequence restrictions are expressed as regular expressions over A. The equivalent structure diagram is drawn according to the notations of JSD [14]. It is also possible to represent the regular expressions by means of a Finite State Machine.

Consistency checking with the OET and the EDG

The OET defines the alphabet of an object and partitions this alphabet in creating, mutating and destroying 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, the structure diagram or expression that defines the behaviour of an object type P must contain all and only the event types in S_AP, the alphabet of P. Formally:

Alphabet rule

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

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

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

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

The partitioning of the alphabet in creating, modifying and destroying event types imposes a default life cycle which must be respected by the sequence restrictions:

Default life cycle rule

$\forall P \in \mathcal{M}$: $S_R P \leq (\Sigma\, c(P)).(\Sigma\, m(P))^*.(\Sigma\, d(P))$[9]

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 $\leq$ can be defined as the dual counterpart of the existence dependency relation. This means that

Restriction rule

$P <\!\!- Q \Rightarrow P \leq Q$

This can be motivated as follows. In a system composed of a mother object of type Q and a marsupial object of type P, both object types will run concurrently. According to the definition of the ||-operator, such a system will only accept sequences of events that satisfy the sequence restrictions of both P and Q. Namely,

---

9 . if $c(P) = \{a_1,...,a_n\}$ then $\Sigma\, c(P) = a_1 + ... + a_n$

$L(P \parallel Q)\backslash S_A Q \subseteq L(Q)$ and
$L(P \parallel Q)\backslash S_A P \subseteq L(P)$

Any scenario of a marsupial that is not acceptable from the point of view of its mother object, will *always* be rejected and can thus be removed from the life cycle definition of the marsupial. It thus seems sensible to demand that a mother object type Q can accept all scenarios of its marsupial P and thus that $P \leq Q$. Namely, if $P \leq Q$ then $L(P \parallel Q)\backslash S_A P = L(P)$ ([22], theorem 3.11). The difference between the general case and the case in which P and Q satisfy the restriction rule is depicted in Fig. 12. In other words, the life cycle of a marsupial must be more deterministic than the life cycle of the mother object type.

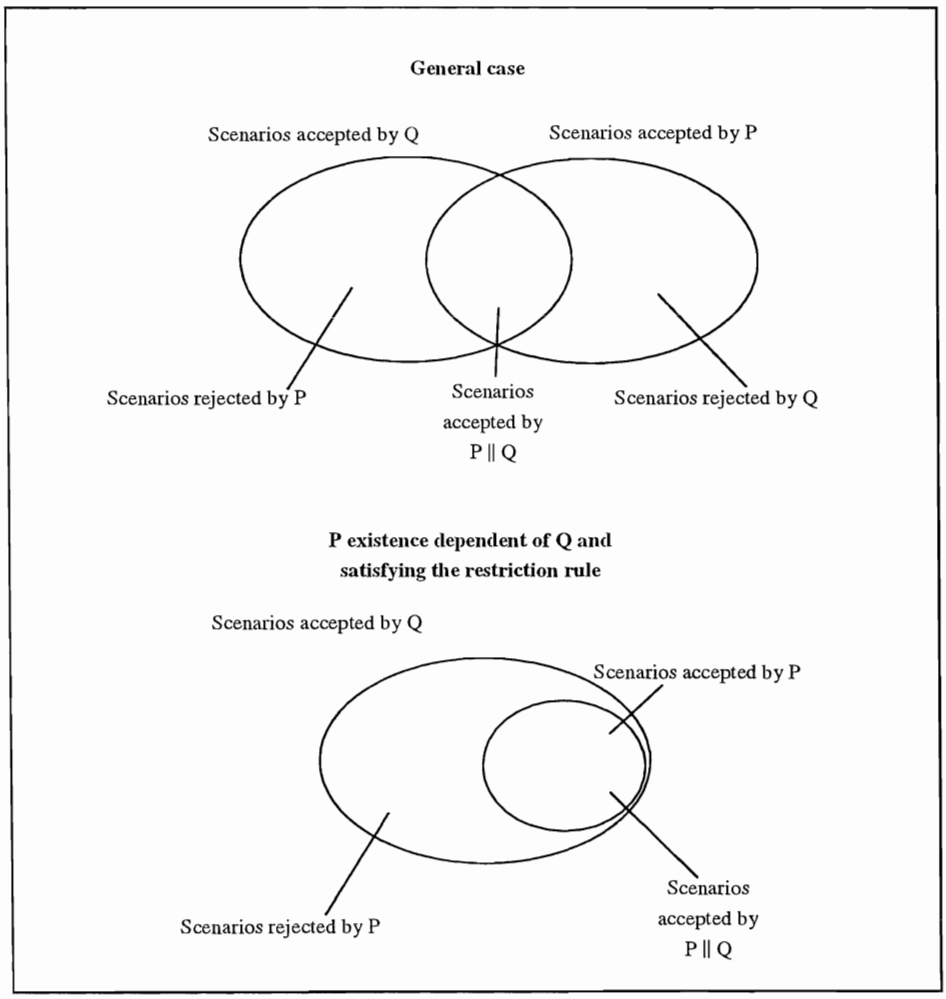

Fig. 12. restriction rule for existence dependent object types

Example 14

With the sequence restrictions as in Example 13 and LOAN existence dependent of BOOK and MEMBER (LOAN ≤ BOOK, MEMBER), we have:

S_RBOOK$\backslash S_A$LOAN
= acquire.catalogue.(borrow + renew + return)*.(sell + lose)$\backslash S_A$LOAN
= (borrow + renew + return)*.(1 + lose)

and S_RMEMBER$\backslash S_A$LOAN

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

= (borrow + renew + return + lose)*

Both expressions are less deterministic than S_RLOAN.

As a result, the set of scenarios accepted by a marsupial is a subset of the intersection of the sets of scenarios accepted by all its mother object types. In this sense, the marsupial acts as a contract between the mother 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 BOOK (see Fig. 13).

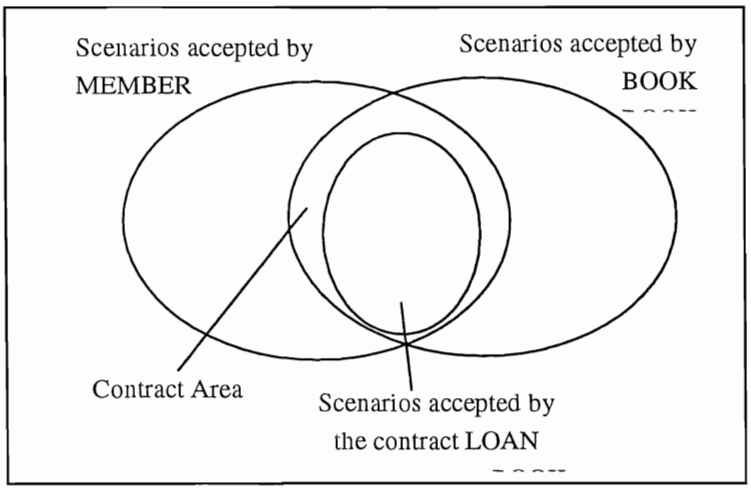

Fig. 13. LOAN as a contract between MEMBER and BOOK

Note that the restriction rule contains a one direction implication. It might indeed happen that $P \leq Q$ without P being existence dependent on Q.

Example 15

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)

To summarise, the complete definition of a conceptual schema is given in Appendix A. Appendix B presents a comprehensive example that illustrates how the restriction rule helps to spot specification errors.

4. COMPARISON WITH OTHER CONCEPTS

Generalisation/Specialisation (A Kind Of -lattice)

Although due to the propagation rule a mother object type acquires all the event types of its marsupials, there is no inheritance relationship between a mother object type and a marsupial object type. Likewise, there is no existence dependency relationship between a generalisation and a specialisation: the life of a specialisation does not depend on the life of a generalisation (see [21] for more details).

As a specialisation object type inherits all the features of the generalisation, it also inherits the marsupials of the generalisation. One might expect that it is then necessary to check the sequence restrictions of the specialisation and those of the *inherited* marsupials for the restriction rule. Fortunately, a proper formalisation of the inheritance relationship (as in [21]) ensures that checking done by comparison with the generalisation object type only is sufficient. More specifically, if G is the generalisation object type and S the specialisation object type it is sufficient to require that $S_RG \leq S_RS\backslash S_AG$ to ensure consistency with the restriction rule. Indeed, let M be a marsupial object type of G and $M \leq G$. We then have $S_RM \leq S_RG\backslash S_AM$ (restriction rule applied to M and G). In addition, if $S_RG \leq S_RS\backslash S_AG$, then also $(S_RG)\backslash S_AM \leq (S_RS\backslash S_AG)\backslash S_AM$. We thus have $S_RM \leq (S_RG)\backslash S_AM \leq (S_RS\backslash S_AG)\backslash S_AM$. As $S_AM \subseteq S_AG$ (propagation rule), $(S_RS\backslash S_AG)\backslash S_AM = S_RS\backslash S_AM$. And thus $S_RM \leq S_RS\backslash S_AM$, which proves the conformity of S and M to the restriction rule.

Note that the requirement that $S_RG \leq S_RS\backslash S_AG$ is somewhat strange as it states that the sequence restrictions of the generalisation type should be more deterministic (or stringent) than the sequence restrictions that specialisation type imposes on inherited event types. This conflicts with the idea of specialisation as 'refinement'. However, it conforms to the idea of specialisation as 'extension': the specialisation can do more than the generalisation. In addition, this requirement ensures universal substitutability of objects [21].

Part Of (A Part Of -lattice) 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 [9, 19, 15, 17, 19, 4]. This is probably due to the fact that the notion of 'part-of' embodies some aspects of existence dependency[10] and propagation of events[11]. 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 car is disassembled, the wheel still exists as an object. Orderlines, on the contrary can usually not exist independent of the order of which they are part-of. Similarly, if the parts are not existence dependent of the aggregate, propagation of events is not

10. See for example [17], 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, ..."
11. See for example [17], 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."

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 orderlines.

As pointed out by de Champeaux et al. [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 [2, 4, 6, 14, 15, 17, 19]. Sometimes the definitions in different methods contradict each other. In Fusion [6] aggregation and relationships are thought of as similar concepts as they both are formed by taking tuples of class instances. In Fusion aggregations can thus be used to model relationships as classes. This contrasts with the OMT method [17] where separate constructs are proposed for "association as a class" and the Part-Of relation.

The existence dependency relation proposed in this paper is a valuable alternative for both the Part-Of relation and the concept of "Association as a Class". The question when to model relationship types as classes is solved by deriving the EDG from the ER-schema: all non-weak relationship types in the ER-schema are object types in the EDG. A Part-Of relation with existence dependent parts can be replaced by weak relationship types. As a result, in case of existence dependent components, the ED relation is identical to the Part-Of relation (see example in Fig. 14, the Part-Of relation is drawn according to the notation of [17]).

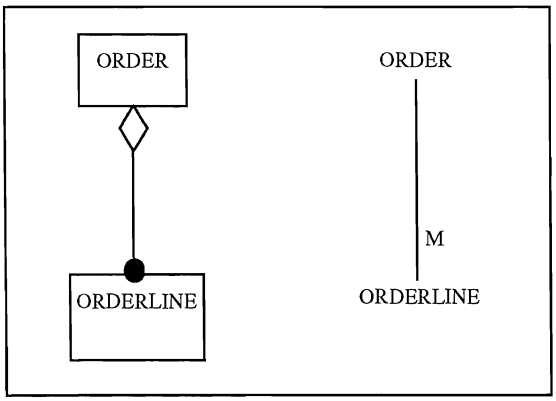

Fig. 14. Existence dependent parts: Object diagram and EDG

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. 15 states that a PC consists of one monitor, one system box, one keyboard and zero to many external drives.

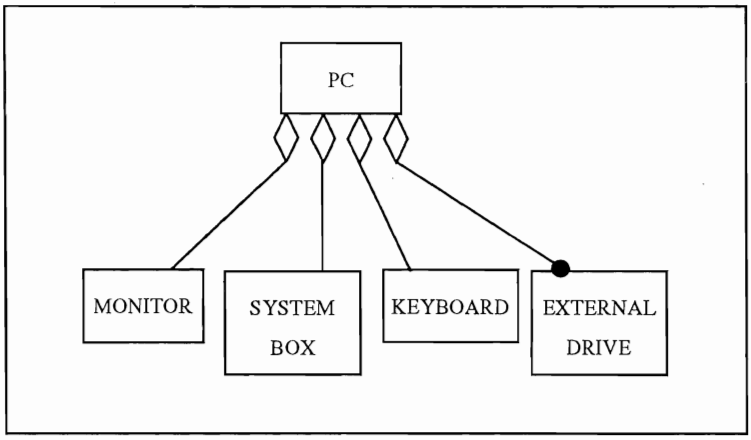

Fig. 15. Non existence dependent parts: Object diagram

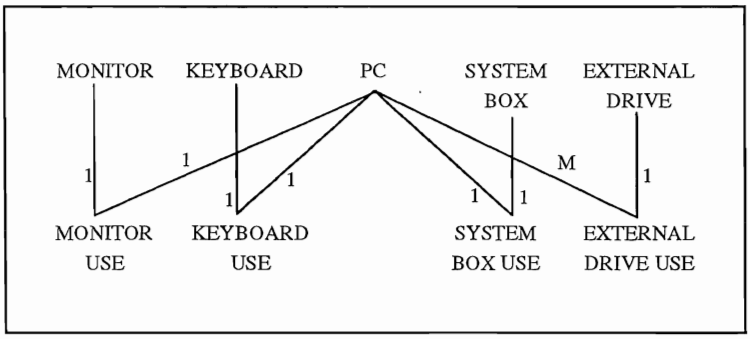

Fig. 16. Non existence Dependent parts: EDG

Fig. 16 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.

Beside the existence dependency considerations, aggregation usually implies some concurrency aspects: the states of the components are also determining the state of the aggregate. In [7, 22] it has been demonstrated how the behaviour of a system composed of many objects related to each other by means of the existence dependency relation can be calculated from individual object behaviour. The principles and the algorithm presented in that paper also apply to a subsystem composed of an aggregate and its parts.

5. DISCUSSION AND CONCLUSION

This paper presented a new classification principle for object types, called Existence Dependency. Existence dependency is present in every data model where at least two classes are related to each other: either one class is existence dependent of the other or the relationship between the two classes is a third object which is existence dependent of the two participating classes.

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

Propagation of events. Event types of existence dependent object types are propagated to the mother object type.

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 mother object types: the contract ensures that the participating object types agree on the allowed behaviour (sequences of events). If the sequence restrictions imposed by each object type on event types 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 classifying object types according to existence dependency, the question whether or not to model a relationship as a class becomes irrelevant. By de facto modelling non weak relationship types as classes, the number of classes in a schema will increase but the logical complexity of individual classes will decrease. This is experienced as a positive effect as volume is easier to deal with than logical complexity. Of course, nothing should prevent the software developer of merging classes together at implementation time. In the example of Fig. 2 (b), the object type that results from the relationship type "groups" can be merged with the object type ORDERLINE by including a (modifiable) reference to ORDER in ORDERLINE.

In addition, existence dependency is a valuable alternative for the vague 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
- it 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 [22, 7]

Acknowledgement
We would like to thank Prof. Verhelst for suggesting this interesting research topic and carefully reading this manuscript.

REFERENCES

1. Baeten, J.C.M., *Procesalgebra,* Kluwer programmatuurkunde, 1986
2. Booch, G., *Object Oriented Analysis and Design with Applications.* Second Edition, Benjamin/Cummings, Redwood City, CA, 1994.
3. Chen, P.P., The Entity Relationship Approach to logical Database Design, *QED information sciences,* Wellesley (Mass.),1977
4. Chen P. P., The Entity-Relationship Model - Toward a Unified View of Data, *ACM Transactions on Database Systems,* Vol. 1, No. 1 (1976) 9-36.
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., Formal deadlock elimination in an object oriented conceptual schema, *Data and Knowledge Engineering,* Vol. 15 (1995) 1-30
9. A. Dogac and P. P. Chen, *Entity-Relationship Model in the ANSI/SPARC Framework,* in P.P. Chen, Entity Relationship Approach to Information Modelling and Analysis, Proc. of the Second International Conference on Entity-Relationship Approach, Washington D.C., October 12- 14 , 1981, North Holland, 1983, pp. 357 - 374
10. A. Dogac, E. Ozkarahan, P. Chen, *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
11. 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.
12. Hoare C. A. R., *Communicating Sequential Processes* (Prentice-Hall International, Series in Computer Science, 1985).
13. Hopcroft John E., Ullman Jeffrey D., *Formal languages and their relation to automata,* Addison Wesley Publishing Company, 1969
14. Jackson, M. A., *System Development,* Prentice Hall Englewood Cliffs (N.J.), 1983, 418 pp.
15. Jacobson Ivar et al., *Object-Oriented Software Engineering, A use Case Driven Approach,* Addison-Wesley, 1992
16. Milner R., *A calculus of communicating systems* (Springer Berlin, Lecture Notes in Computer Science, 1980).
17. F. Put, *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.
18. Rumbaugh, J., Blaha M., Premerlani, W., Eddy, F., Lorensen, W., *Object Oriented Modelling and Design,* Prentice Hall International, 1991
19. Shlaer, S., Mellor, S.J., *Object-Oriented Systems Analysis: Modelling the World in Data,* Prentice Hall, 1988
20. S. Shlaer, S.J. Mellor, *Object Lifecycles: Modeling the World in States* (Prentice Hall, Englewood Cliffs, New Jersey, 1992).
21. Snoeck Monique, Dedene Guido, Generalization/Specialization and Role in Object Oriented Conceptual Modeling, *Data and Knowledge Engineering,* 19(2), 1996

22. Snoeck Monique, *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)
23. Webre N. W., An Extended Entity-Relationship Model and its Use on a Defence Project, in *Entity Relationship Approach to Information modeling and Analysis*, P. P. Chen (ed.), 1983, pp.173-193
24. Wijsen J., *Extending dependency theory for temporal databases*, PhD. Dissertation (Katholieke Universiteit Leuven, Faculty of Science & Department of Computer Science, February 1995)

APPENDIX A

Definition

A conceptual schema is an ER-schema $(\mathcal{ER}, \mathcal{A}, \mathcal{W})$ and a derived set of object types $\mathcal{M} \subseteq <\mathcal{P}(A), R^*(A)>$ such that:

(a) $\bigcup_{P \in \mathcal{M}} S_A P = A$

(b) The ER-schema satisfies the restrictions

1) Every association relates exactly two distinct elements of $\mathcal{ER}$:
$\forall a \in \mathcal{A}$: from(a), to(a) $\in \mathcal{ER}$ and from(a) $\neq$ to(a)

2) Every relationship type relates at least two (not necessarily distinct) elements:
$\forall R \in \mathcal{ER}, a \in \mathcal{A}$: [from(a) = R $\Rightarrow \exists b \in \mathcal{A}$: from(b) = R]

3) A weak relationship type cannot be aggregated:
$\forall R \in \mathcal{ER}$: [$\exists a \in \mathcal{W}$: from(a) = R $\Rightarrow \neg(\exists b \in \mathcal{A}$: to(b) = R)]

4) A weak relationship type is weak on at most one element:
$\forall R \in \mathcal{ER}$: [$\exists a \in \mathcal{W}$: from(a) = R $\Rightarrow \neg(\exists b \in \mathcal{W}$: from(b) = R)]

(c) $\mathcal{M}$ contains all entity types and non-weak relationship types of $\mathcal{ER}$.
$\mathcal{M} = \mathcal{ER} \setminus \{ R \mid \exists b \in \mathcal{W}$: from(b) = R$\}$

(d) The existence dependency graph is derived from the ER-schema by means of the following rules:
<- is a bag over $\mathcal{M} \times \mathcal{M}$ such that $\forall P, Q \in \mathcal{ER}$:
(P,Q) $\in$ <- $\Leftrightarrow$
$\exists a \in \mathcal{A} \setminus \mathcal{W}$: from(a) = P and to(a) = Q and $\neg(\exists b \in \mathcal{W}$: from(b) = P)
or
$\exists a \in \mathcal{W}, \exists b \in \mathcal{A} \setminus \mathcal{W}, \exists R \in \mathcal{ER}$: from(a) = R and
to(a) = P and from(b) = R and to(a) = Q

<- must satisfy the following restrictions:
1) The EDG is fully connected: $\forall P \in \mathcal{M}: \exists Q \in \mathcal{M}$: (P,Q) $\in$ <- or (Q,P) $\in$ <-
2) An object type is never existence dependent of itself: $\forall P \in \mathcal{M}$: (P,P) $\notin$ <-
3) <- is acyclic. This means that:
$\forall n \in \mathbb{N}, n \geq 2, \forall P_1, P_2, ..., P_n \in \mathcal{M}$:
$(P_1,P_2), (P_2,P_3),..., (P_{n-1},P_n) \in$ <- $\Rightarrow (P_n,P_1) \notin$ <-

(e) The object-event table $\mathcal{T}$ is a table with one row per event type in A and one column per object type in $\mathcal{M}$. Each cell contains a blank, 'C', 'M' or 'D'.
$\mathcal{T} \subseteq \mathcal{M} \times A \times \{' ', 'C', 'M', 'D'\}$ such that
1) $\forall P \in \mathcal{M}, \forall a \in A$:
(P,a,' ') $\in \mathcal{T}$ or (P,a,'C') $\in \mathcal{T}$ or (P,a,'M') $\in \mathcal{T}$ or (P,a,'D') $\in \mathcal{T}$
2) $\forall P \in \mathcal{M}$: c(P) = {a $\in$ A | (P,a,'C') $\in \mathcal{T}$}
m(P) = {a $\in$ A | (P,a,'M') $\in \mathcal{T}$}
d(P) = {a $\in$ A | (P,a,'D') $\in \mathcal{T}$}

3) c(P), m(P), d(P) are pairwise disjoint
   $c(P) \cup m(P) \cup d(P) = S_A P$
   $c(P), d(P) \neq \varnothing$
4) Propagation rule and Type of involvement rule
   $P <\text{-} Q \Rightarrow S_A P \subseteq S_A Q$ and $c(P) \subseteq c(Q) \cup m(Q)$
   and $m(P) \subseteq m(Q)$ and $d(P) \subseteq d(Q) \cup m(Q)$
5) Contract rule : $\forall P, Q \in \mathcal{M}: \#(S_A P \cap S_A Q) \geq 2 \Rightarrow \exists R_1, R_2, \ldots R_n \in \mathcal{M}$:
   $\forall i \in \{1,\ldots,n\}: R_i <\text{-} P,Q$ and $S_A R_1 \cup \ldots \cup S_A R_2 = S_A P \cup S_A Q$

(f) $\mathcal{M}$ defines sequence restrictions for each object type such that
1) Alphabet rule : $\forall P \in \mathcal{M}: \varphi(S_R P) = S_A P$
2) Default life cycle rule : $\forall P \in \mathcal{M}: S_R P \leq (\Sigma\, c(P)).(\Sigma\, m(P))^*.(\Sigma\, d(P))$
3) Restriction rule : $\forall P, Q \in \mathcal{M}: P <\text{-} Q \Rightarrow P \leq Q$

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.

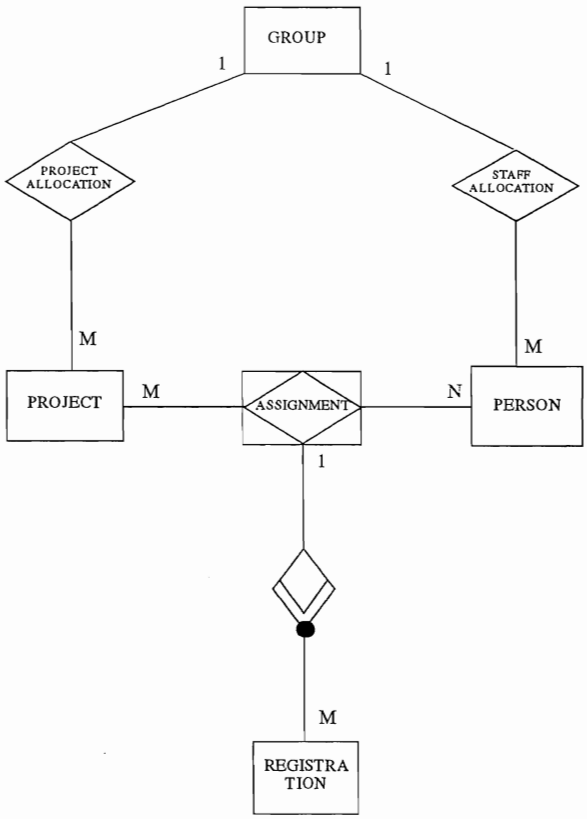

Fig. 17. ER-schema for the project administration

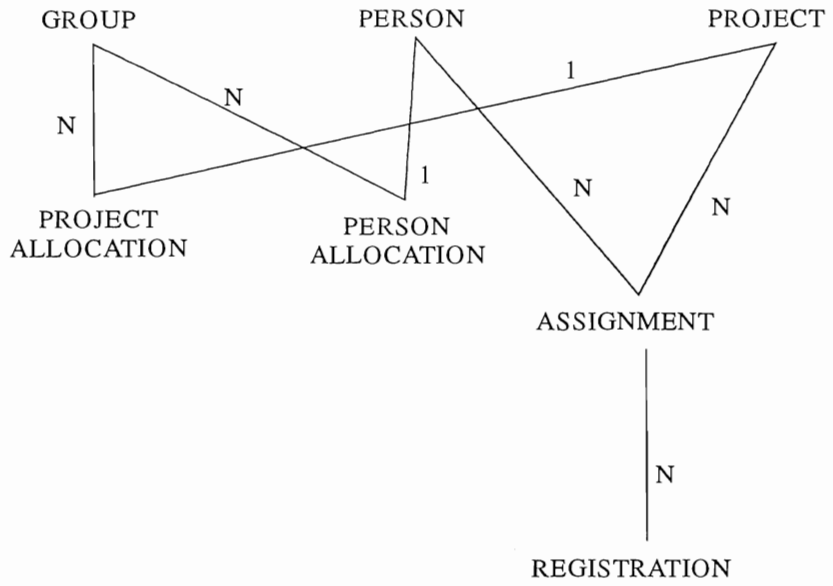

Fig. 18. Existence Dependency Graph for the project administration

	GROUP	PROJECT	PERSON	PROJ-ALLOC.	PERS-ALLOC	ASSIGNMENT	REGISTRATION
cr_group	C						
del_group	D						
cr_project		C					
close_project		M					
del_project		D					
cr_person			C				
del_person			D				
proj_alloc	M	M		C			
proj_dealloc	M	M		D			
pers_alloc	M		M		C		
pers_dealloc	M		M		D		
assign		M	M			C	
end-assign		M	M			M	
del-assign		M	M			D	
register		M	M			M	C
del_registr		M	M			M	D

Fig. 19. Object Event Table for the project administration

S_RGROUP = cr_group.(proj_alloc + proj_dealloc + pers_alloc + pers_dealloc)*.del_group

S_RPROJECT = cr_project . (proj_alloc+ proj_dealloc + assign + end_assign + del_assign + register + del_registr)* .close_project .del_project

S_RPERSON =cr_person .(pers_alloc + pers_dealloc + assign + end_assign + del_assign + register + del_registr)* . del_person

S_RPROJECT_ALLOCATION = proj_alloc.proj_dealloc

S_RPERSON_ALLOCATION = pers_alloc.pers_dealloc

S_RASSIGNMENT = assign.(register + del_registr)* .(end_assign + close_project).(del_registr)*. del_assign

S_RREGISTRATION = register.del_registr

Fig. 20. Sequence restrictions for the project administration

When checking the specifications against the restriction rule (f).3, 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_proj' event:

PROJECT\ S_AASSIGNMENT = (assign + end_assign + del_assign + register + del_registr)*.close_project

So the 'close_proj' cannot be followed by 'del_registr' or 'del_assign' events as allowed by the sequence restrictions of ASSIGNMENT . Even worse, the specification of PROJECT requires the del_assign event to precede the close_project event, while the specification of ASSIGNMENT requires the opposite: close_project must precede del_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 form the sequence restrictions of ASSIGNMENT. The correct solution defines the sequence restriction of ASSIGNMENT as:

S_RASSIGNMENT = assign.(register + del_registr)* .end_assign .(del_registr)*. del_assign

The fact that all the 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 "end_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.