Temporal Features of Class Populations and Attributes in Conceptual Models

(extended version)[1]

Dolors Costal, Antoni Olivé, Maria-Ribera Sancho

Universitat Politècnica de Catalunya, Dept. Llenguatges i Sistemes Informàtics
Jordi Girona Salgado 1-3, Campus Nord, Ed. C-6, E 08034 Barcelona (Catalonia)
e-mail: {dolors|olive|ribera}@lsi.upc.es

Abstract. Constraints play an important role in conceptual modeling. In general, the specification of constraints, both static and transition, must be done in some logic-based language. Unfortunately, the resulting formulas may be complex, error-prone and difficult to read. This explain why almost all conceptual modeling languages have developed a special, easy-to-use syntax (language features) to state the most common constraints. Most features (often with graphical symbols) developed so far are concerned with static constraints (like keys, partitions or cardinalities), and very little work has been done for transition constraints.

In this paper, we identify six temporal features, three related to class populations and three to attributes. The corresponding transition integrity constraints appear in almost any conceptual model and their specification is necessary and important. We believe that our temporal features make their specification simple and practical. We have named each feature, and provide a declarative and procedural formalization for them.

1. Introduction.

A conceptual model consists of two (sub)models: The structural and the behavioural model. The first describes the object types characterizing the objects in the domain, their structural relationships, the object attributes and relationships, the derivation rules defining the population of derived object types, or the values of derived attributes and relationships, and the static and transition integrity constraints. The behavioural model describes the event types, the integrity constraints associated with the events, the effect of these events on the Information Base, and the events that must be generated [Bor85].

This paper focuses mainly on the specification of transition integrity constraints. These constraints are conditions that involve facts of two or more states of the Information Base. Usually, they involve facts of only two consecutive states, constraining the transition between them, but in general the constraints may refer to any number of states [ISO82].

In general, the specification of constraints, both static and transition, must be done in some logic-based language allowing, among other things, the use of connectors and quantifiers. Unfortunately, the resulting formulas may be complex, error-prone and difficult to read. This explain why almost all conceptual modeling languages have

---

[1] This is an extended version of the paper on "Temporal Features of Class Populations and Attributes in Conceptual Models" wich has been accepted for presentation at ER'97 Conference.

developed a special, easy-to-use syntax (language features) to state the most common constraints.

Most features (often with graphical symbols) developed so far are concerned with static constraints, such as keys, inclusion, exclusion, equality, partition [VeV82] and, above all, cardinality constraints [LEW93].

Transition constraints are considered explicitly in some languages [GKB82, Kun84, SFN+84, WMW89, DHR91], usually expressed in temporal or dynamic logic. However, very little work has been done in selecting some common subset of them, and developing the corresponding syntactic features. Almost the only temporal features that we may find in conceptual modeling languages are the possible definition of mutable vs. immutable (or constant) attributes and relationships, and initial values of attributes [HaM81, JSH+96] and state diagrams [RBP+91, CoD94, Rat97]. [BiD94] identifies and analyzes a particular transition constraint, but without special language support. In consequence, the designer is forced to specify many transition constraints with complete formulas, or to leave them unspecified.

In this paper, we identify six new temporal features, three related to class populations and three to attributes. We believe that the corresponding transition integrity constraints appear in almost any conceptual model and that their simple specification is necessary and important. We have named each feature, but we have not attempted to propose a graphical symbol for them. We define the temporal features formally, at two levels: declarative and procedural. In the declarative level, the features are related to permissible changes of the Information Base, independent of the transactions that induce the changes. To define features procedurally, we consider that a transaction consists of a number of primitve structural events, and we determine the conditions such events must satisfy to maintain the Information Base consistent with respect to the temporal features.

Section 2 presents the three temporal features related to class population. We also introduce some simple notation needed for the formalization. Section 3 presents the three temporal features related to attributes and, particularly, we discuss their application to the difficult problem of aggregates/composites. The declarative formalization of the proposed features is given in these Sections, while in Sections 4 and 5 we deal with the procedural formalization: in Section 4 we introduce the primitive structural events that we consider, and in Section 5 we present the conditions transactions must satisfy. Additional detail is given in the Appendix. Section 6 summarizes our conclusions, and points out future research.

2. Temporal features of class populations.

A structural model defines, among other things, a set of classes organized into a class hierarchy through generalization (or specialization). Objects are instances of one or more classes. We assume that, in the general case, an object may change its classes dynamically. The population of a class at a time `t` is defined as the set of objects that are instance of that class at `t`.

2.1 Static features.

The most common static features of class populations defined in structural models are class inclusions, partitions and cardinalities. A specialization such as `class a ISA class b` defines that, at any time, the population of class `a` is a subset of the population of class `b`.

Class partitions, such as the `partition of class a into classes b and c`, define that classes `b` and `c` are specializations of class `a`, and that the population of classes `b` and `c` are disjoint. A partial (or incomplete) partition indicates that, at any time, the union of the populations of both classes is a subset of that of class `a`, while a complete partition defines that, at any time, the union of the populations of both classes is equal to the population of class `a` [MaO95].

Finally, some languages allow defining constraints on the cardinality of classes [EKW92, LEW93]. These constraints can also be considered static features since they must be satisfied at any time.

2.2 Temporal features: general definitions.

Temporal features of class populations define some time-dependent constraints that the populations must satisfy. We will define in this Section three of such features. We introduce first the notation we need for their formalization.

We assume that time is discrete, and that time points are expressed uniformly at some level of abstraction (granularity). The lifespan `ls` of an IS is the temporal interval $ls = (t_i, t_f)$ during which the system exists. Similarly, the lifespan `ols` of an object `o`, $ols(o) = (t_{o,i}, t_{o,f})$ is the temporal interval during which object `o` exists. It is obvious that the lifespan of an object must be included in lifespan `ls`. We assume that once an object ceases to exist in the system, it cannot exist again in the future. We usually do not know in advance the exact values of the above time points, but this is unimportant for our purposes: we only need to assume that such values do exist.

We will denote the starting and ending points of a temporal interval `ti` by the functions `startsAt(ti)` and `endsAt(ti)`, respectively. We also use the predicate `belongsTo(t,ti)` to indicate that time point `t` is included in time interval `ti`.

We use a two-term existence predicate `a(o,t)` to indicate that object `o` is an instance of class `a` at time `t`. Objects can be instances of classes only at times belonging to the lifespan `ls`. In general, an object may be an instance of a class during one or more disjoint and non-consecutive time intervals, called membership intervals. We denote by $mi(o,a) = \{ti_1, ..., ti_n\}$ the set of membership intervals of object `o` in class `a`. It is obvious that the membership intervals of an object `o` in any class are included in object's lifespan `ols(o)`. On the other hand, there is a correspondence between the existence predicate and the membership intervals, which is captured by the following equivalence:

$$\forall O,T(a(O,T) \leftrightarrow \exists TI(TI \in mi(O,a) \wedge belongsTo(T,TI)))$$

2.3 Permanent instances.

The permanent instances feature of a class defines whether or not its instances are permanent. A permanent object is an object that, once created, exists until the end of lifespan `ls`. Possible values for this feature are: `[non-]permanent instances.`

Formally, if class `a` has `permanent instances` then:

$\forall O,T(a(O,T) \rightarrow endsAt(ols(O)) = endsAt(ls))$

The value `non-permanent instances` does not impose constraints on objects' lifespan.

It can be seen that if a class has `permanent instances`, then all its subclasses must have also `permanent instances`.

As an example, consider the classes shown in the structural model of Figure 1. Class `car` would have `non-permanent instances`, if we assume that cars may cease to exist at some time. On the other hand, class `person` could be with `permanent instances`, if we assume that persons, once created, are always known to the system. The five subclasses of `person` would also have `permanent instances`.

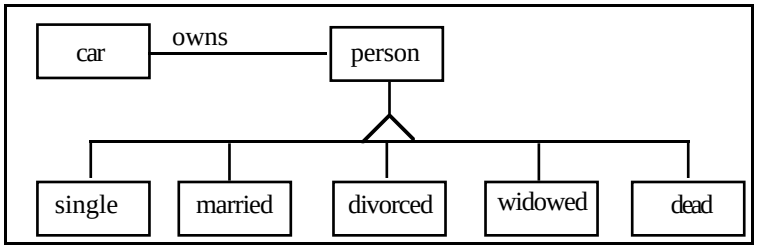

Figure 1. Example of structural model.

2.4 Initial membership.

The initial membership feature of a class defines whether or not potential instances of this class must be (must not be) members at object's creation time. Possible values for this feature are `[always|never|sometimes] initially member.`

If class `a` is `always initially member`, then all objects that are sometime instances of class `a` must be instances of it at creation time. Formally,

$\forall O,T(a(O,T) \rightarrow a(O,startsAt(ols(O)))$

Similarly, if class `a` is `never initially member`, then all objects that are sometime instances of class `a` must not be instances of it at creation time. Formally,

$\forall O,T(a(O,T) \rightarrow \neg a(O,startsAt(ols(O)))$

The value `sometimes initially member` does not impose constraints on membership at object's creation time.

It can be seen that if a subclass of a partition is `always initially member`, then all other subclasses of the partition must be `never initially member`.

Note that this feature is orthogonal to the previous one. We can see four of the six possible combinations in the example of Figure 1. Classes `car` and `person` are

likely to be `always initially member`, because objects that are cars or persons must be instances of their respective class from the beginning of their existence. Similarly, class `single` would also be `always initially member`, since if a person is sometime single then he had to be single when he was created. Classes `married`, `divorced` and `widowed` could be `sometimes initially member`, if we allow that when a person is known for the first time to the system he may be also instance of one of these classes. Finally, class `dead` would be `never initially member`, if we assume that we do not create new persons that are dead at the beginning of their existence in the system.

2.5 Membership intervals.

The membership intervals feature of a class defines some characteristics of the time intervals during which objects may be instances of that class. The three possible values for this feature are: `[single[non-]permanent|multiple] membership intervals.`

If class `a` is `single (permanent` or `non-permanent) membership interval` then `|mi(o,classA)| = 1.` In the other case (`multiple`), the number of time intervals may be greater than 1.

If class `a` is `single permanent membership interval` then its objects remain in the class until the end of their lifespan. Formally, we say that the end of their unique membership interval coincides with the end of their lifespan:

$\forall O, TI(TI \in mi(O,a) \rightarrow endsAt(TI) = endsAt(ols(O)))$

If a class is `single non-permanent membership interval` then its instances may leave that class before the end of their lifespan.

Note that this feature is orthogonal to the previous ones. We can see several combinations in the example of Figure 1. Classes `person` and `car` would be `single permanent membership interval`, since once an object is classified as `person` or `car`, it remains instance of the class until the end of its existence. Similarly, class `dead` would also be `single permanent membership interval` (unfortunately). Class `single` would be `single non-permanent membership interval`, since persons can only be single during a unique time interval, but may leave this class. Finally, classes `married` and `divorced` would be `multiple membership intervals`, since persons may be married or divorced during several disjoint and non-consecutive time intervals.

2.6 Application.

The temporal features described above can be used in any conceptual model. They capture in a simple way important temporal properties of class populations that may help in the definition and understanding of a system's behaviour.

In particular, the features can be useful to characterize (part of) the behaviour of roles, which are used in several languages [Per90, GSR96]. In languages using state diagrams (such as those described in [RBP+91, CoD94, Rat97]), some of the temporal features may be inferred from the diagrams. Each state corresponds to a class. If the initial state is unique, then it is `always initially member`. The other states are

never initially member. The final state is single permanent membership interval. The other states are single non-permanent or multiple membership intervals, depending on whether the objects may be in the corresponding state one or more times.

3. Temporal features of attributes.

3.1 Static features.

A structural model includes the relevant attributes of classes, which, according to the property induction principle, constraint the factual properties that objects may have [GMB94]. Common static features of attributes defined in many conceptual models include single/multivalued attributes, optional/mandatory attributes, functional/total (injective/surjective) attributes [BoC95], participation constraints [LEW93] and other special constraints such as constant, identifier, subset, equality or uniqueness [VeV82].

3.2 Temporal features: general definitions.

Temporal features of attributes define some time-dependent constraints that attribute values must satisfy. We will define in this Section several of such features. We introduce first the notation we need for their formalization.

Let attname be an attribute of class a taking values from class b. We use a three-term predicate attname(o1,o2,t) to indicate that the value of attribute attname for object o1 is object o2 at time t. For multivalued attributes, the meaning is that object o2 is one of the attribute values. Objects o1 and o2 must be instances of their corresponding classes at time t. We formalize this as the Temporal Referential Integrity axiom:

$\forall O1,O2,T(attname(O1,O2,T) \rightarrow a(O1,T) \wedge b(O2,T))$

Note that we do not consider "null" values to be a special kind of attribute value. Our equivalent concept is that an object does not have a value for a given attribute. On the other hand, it is always assumed that only existing objects can have attribute values or can be attribute values of objects. This is captured by the following implications:

$\forall O,T(\neg a(O,T) \rightarrow \neg\exists O1\ attname(O,O1,T))$
$\forall O,T(\neg a(O,T) \rightarrow \neg\exists O1\ attname(O1,O,T))$

We will see that, in many cases, it is interesting to define the temporal features of inverse attributes [HaM81]. We do not need special notation for that, but of course the value of such attributes must be syncronised. Thus, if invattname is the inverse of attname in class b, the following equivalence must hold:

$\forall O1,O2,T(attname(O1,O2,T) \leftrightarrow invattname(O2,O1,T))$

3.3 Initial value.

The initial value feature of an attribute of a class defines whether or not objects of this

class must have (must not have) values for the attribute when the object starts a membership interval in that class. Possible values for this feature are `[always|never|sometimes] initially valued.`

If attribute `attname` of class `a` is `always initially valued`, then all objects of class `a` must have a value for attribute `attname` every time that they start a membership interval in that class. Formally,

$\forall O,TI(TI \in mi(O,a) \rightarrow \exists O1\ attname(O,O1,startsAt(TI))$

Similarly, if attribute `attname` of class `a` is `never initially valued`, then all objects of class `a` cannot have a value for attribute `attname` when they start a membership interval in that class. Formally,

$\forall O,TI(TI \in mi(O,a) \rightarrow \neg\exists O1\ attname(O,O1,startsAt(TI))$

The value `sometimes initially valued` does not impose constraints on attribute values at the beginning of membership intervals.

As an example, consider the following class definitions:

```
class person
   name: string;
   phone: integer;
   worksIn: set of project

class married ISA person (multiple membership intervals)
   spouse: person
```

Attributes `name` and `spouse` are likely to be `always initially valued`. Note that `spouse` must be given a value every time a `person` starts a membership interval in `married`. Attribute `phone` might be `sometimes initially valued`, if we assume that we may or may not know a person's phone number when he is created. Attribute `worksIn` could be `never initially valued`, if we assume that when a person is created he is still not working in any project.

3.4 Existence intervals.

The existence intervals feature of an attribute defines some characteristics of the time intervals during which exist values for that attribute. This feature is mainly useful for single valued attributes, and this is the only case we will consider here. The three possible values for this feature are `[single[non-]permanent|multiple] existence intervals.` This feature is orthogonal to the previous one.

If attribute `attname` of class `a` is `single permanent existence interval`, then it means that once an object `o` of class `a` takes a value for attribute `attname` -at some time `t` belonging to a membership interval `TI`$\in$`mi(o,a)`- it will keep on having some value for attribute `attname` until the end of the membership interval. Formally:

$\forall O,O1,TI,T(TI \in mi(O,a) \wedge belongsTo(T,TI) \wedge attname(O,O1,T) \wedge belongsTo(T1,TI) \wedge T1>T \rightarrow \exists O2\ attname(O,O2,T1))$

In the example given above, `name` could be an attribute with `single permanent existence interval`. Once an object of class `person` has a `name`, it keeps on having some `name` while the object is classified as `person`. Note that the `name`

may change: this feature only requires that some value continue existing for `name`.

A more involved example would be the following attribute:

```
class employee ISA person (multiple membership intervals)
   assignedTo: department (never initially valued,
        single permanent existence interval)
```

In this case, persons may be employees during several membership intervals, and attribute `assignedTo` is never known when a `person` becomes `employee`, but once known, there is some value for it until he ceases to be employed. The `department` may change during an employee's life. Figure 2 shows graphically the relationship between the three intervals. Note that during the second membership interval, the `employee` was never `assignedTo` any `department`.

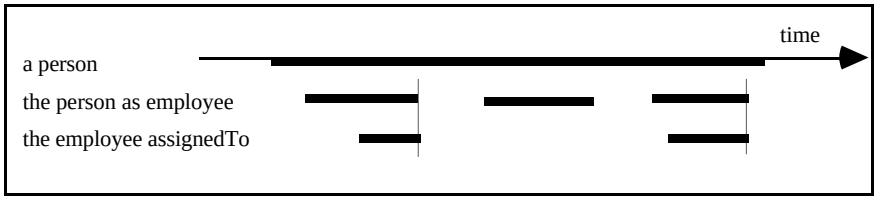

Figure 2. Example of object lifespan, memberhip intervals and attribute existence intervals

If attribute `attname` of class `a` is `single non-permanent existence interval`, then it means that:

- once an object `o` of class `a` takes a value for attribute `attname`, at some time `T` belonging to a membership interval `TI`$\in$`mi(o,a)`,
- and then it ceases to have a value for that attribute during `TI`,
- it cannot have later (during the same `TI`) any value for the attribute.

In other words, there is a single existence interval of the attribute, but it may end before the corresponding membership interval. Formally:

$$\forall O,O1,TI,T(TI\in mi(O,a) \wedge belongsTo(T,TI) \wedge attname(O,O1,T) \wedge belongsTo(T1,TI) \wedge T1>T \wedge \neg\exists O2\ attname(O,O2,T) \wedge belongsTo(T2,TI) \wedge T2>T1 \rightarrow \neg\exists O3\ attname(O,O3,T2))$$

A simple example could be the following attribute:

```
class person (non-permanent instances,
        always initially member, single membership interval)
   mother: person (always initially valued,
        single non-permanent existence interval)
```

In this case, there is a `single membership interval`. During this interval, the value for attribute `mother` exists initially, but once a `person` loses a value for `mother` he cannot regain it later.

The value `multiple existence intervals` does not constrain the intervals of attribute value existence. An example could be:

```
class student
   tutor: person (always initially valued,
        multiple existence intervals)
```

In this case, a `student` may have or may not have, at a given time, a `tutor`.

However, a student must have a tutor when it is known to the system, but he may remain without tutor during some time and have another later.

3.5 Application of initial values and existence intervals to optional attributes.

Before continuing with the last feature, it may be interesting to consider the relationship of the previous features with the well-known "optional/mandatory" static feature of single valued attributes.

The two features described above are orthogonal, and each has three possible values, thus originating nine possible combinations. Mandatory attributes correspond to the combination:
always initially valued/single permanent existence interval,
while the other eight combinations correspond to optional attributes. This means that our features may be helpful in providing additional details on why and when a single valued attribute is optional.

Lack of space prevents us to explain in detail each combination. However, we have already shown before three examples:
- assignedTo is optional only at the beginning of an employee's life.
- mother is optional only at the end of a person's life.
- tutor is mandatory at the beginning of a student's life, and optional later.

3.6 Changeability.

Our last temporal feature (which is also orthogonal to the previous ones) deals with changes of attribute values. A feature similar to this one is provided by SDM [HaM81]. We distinguish here between single and multivalued attributes. For single valued attributes possible values are [non-]modifiable. For multivalued attributes we define two sub-features with possible values [insertions [non-] allowed] and [deletions [non-] allowed].

Formally, if a single valued attribute attname of class a is non-modifiable then:

$$\forall O,O1,O2,TI,T(TI \in mi(O,a) \wedge belongsTo(T\text{-}1,TI) \wedge attname(O,O1,T\text{-}1) \wedge belongsTo(T,TI) \wedge attname(O,O2,T) \rightarrow O1 = O2)$$

Note that this feature only forbids the changes of values of attname. It says nothing with respect to changes from no value for the attribute to some value. The value modifiable does not constrain the changes to single valued attributes.

If a multivalued attribute attname of class a has insertions non-allowed, then the attribute can take values only when an object starts a membership interval in class a. Formally:

$$\forall O,O1,TI,T(TI \in mi(O,a) \wedge belongsTo(T,TI) \wedge attname(O,O1,T) \rightarrow attname(O,O1,startsAt(TI))$$

The value insertions allowed does not constrain insertions of new attribute values.

If a multivalued attribute attname of class a has deletions non-allowed,

then the attribute can take values only when an object starts a membership interval in class a. Formally:

$$\forall O,O1,TI,T,T1(TI \in mi(O,a) \wedge belongsTo(T,TI) \wedge attname(O,O1,T) \wedge belongsTo(T1,TI) \wedge T1 > T \rightarrow attname(O,O1,T1))$$

The value deletions allowed does not constrain deletions of existing attribute values.

The following class definition shows a complete example of this feature:

```
class order
   orderNo: integer; (non-modifiable)
   lines: set of orderLine (insertions non-allowed,
             deletions non-allowed)
```

3.7 Composite temporal features.

In some cases, it may be convenient to give a particular name to a specific combination of values of the three features above, including perhaps values of other features (static or temporal).

For example, a single valued attribute with temporal features:

always initially valued
single permanent existence interval
non-modifiable

could be called a constant attribute.The same name could be given to a multivalued attribute with temporal features:

always initially valued
insertions non-allowed
deletions non-allowed.

Note that, in general, the features of an attribute are independent of those of its inverse. The following example shows that an attribute may be constant while its inverse is not:

```
class person
   parents: set of person inverse of children (constant)
   children: set of person inverse of parents
       (never initially valued, insertions allowed,
       deletions non-allowed)
```

We could give a special name, like fixed, to an attribute which is constant at both sides:

```
class order
   hasLines: set of orderLine inverse of ofOrder (constant)
class orderLine
   ofOrder: order inverse of hasLines (constant)
```

In this example, the association between an order and its orderLine is completely fixed at creation time, and cannot be changed later.

3.8 Application to aggregation/composition.

The aggregation (or composition) abstraction, and the part-of relationships are frequently found in conceptual modeling languages [BoC95]. However, their semantics is not always well-clarified and, as a result, their use becomes problematic [dCF92]. In many cases, aggregation is considered a special form of association and its semantics is either language-dependent [KBG89,CoD94,Rat97] or, worst, left (partially) unspecified.

Aggregation has cognitive, static and temporal aspects. Cognitive aspects have been studied in [WCH87,Sto93,MaO95]. Static aspects include the definition of which is the aggregate and which are the parts. The cardinality constraints associated with part-of links have been studied in [Mots93].

Temporal aspects have been less studied. In this respect, we believe that our temporal features may be helpful in clarifying the temporal behaviour of aggregates. In what follows we analyze two recent interpretations of aggregation, and show that they can be fully characterized by our temporal features.

In Syntropy, "aggregation ... mean life-time dependency; in particular, that life-times of the 'parts' are contained within the life-time of the 'whole'. The 'parts' are permanently attached to the whole, and cannot be removed from it without being destroyed. Conversely, destroying the 'whole' destroys the 'parts'" [CoD94, p.39].

A classical example is the "`division part of company`": "Each division must be associated with a single company, and it must remain associated with that company throughout its life-time. Divisions can be created and destroyed during the life-time of a company, but a division cannot be moved from one company to another. If the company is destroyed, so are the divisions."

Using our temporal features, this semantics is completely captured by:

```
class company (non-permanent instances)
   hasDivisions: set of division inverse of belongsTo
      (insertions allowed, deletions allowed)
class division (non-permanent instances)
   belongsTo: company inverse of hasDivisions (constant)
```

A similar approach is taken by UML: "Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end 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" [Rat97, p.53]. In composition, destroying the whole destroys the parts.

Note that, in both cases, deletion of the whole causes the deletion of the parts. This is a necessary consequence in our features, which is formalized in Section 5. Informally, we may see, in the above example, why the deletion of a company implies necessarily the deletion of its divisions:

1- If the company is deleted at time `t`, then the company cannot have at `t` any value for attribute `hasDivisions`.

2. Therefore, no division may `belongTo` (inverse of `hasDivisions`) at time `t` to the company just deleted.

3. We cannot delete the possible values of attribute belongTo, because there must be always a value for such attribute (single permanent existence interval).
4. Another option would be to modify such values (to another company). But this is not possible, because they are non-modifiable.
5. Therefore, the only option left to maintain consistency is to delete the divisions.

4. Primitive structural events

In the two previous Sections, we have defined six new temporal features for conceptual models, and we have given a declarative formalization to them. We now want to provide a procedural formalization. The way we follow is first (in this Section) to define the primitive structural events that change the Information Base (IB), and then (in the next Section) we study the necessary conditions for a transaction (a set of primitive structural events) to satisfy the transition integrity constraints.

We define five kinds of primitive structural events, depending on their effect on the IB: object insertion, object deletion, attribute insertion, attribute update and attribute deletion. For each one of them we formally specify its effect (denoted by E) in the IB and also an applicability axiom (A) that guarantees that the event is productive [VeF85], that is, it ensures that the intended effect does not hold at previous state.

Let c be a class. We use a two term predicate insert_c(X,T) to denote the object insertion primitive structural event corresponding to c. The effect of an insert_c(x,t) is the addition of instance x in object class c at the time instant t (when the event occurs). As can be expected, this event can only be applied if object x was not an instance of c at previous time. Formally,

insert_c(X,T)
E: $\forall$X,T(insert_c(X,T) $\rightarrow$ c(X,T))
A: $\forall$X,T(insert_c(X,T) $\rightarrow \neg$c(X,T-1))

Note that an insert_c(x,t) does not distinguish between the case when x is a new object in the system and when x is already a known object and the insertion just adds it to class c. Sometimes we need to make such distinction, and we will assume that there is a most general class object, such that object(x,t) is true if x is an existing object at time t.

In a similar way, we use a two term predicate delete_c(X,T) to denote the object deletion primitive structural event corresponding to c. The effect of a delete_c(x,t) is the removal of the instance x from object class c at the time instant t (when the event occurs). This event can only be applied if object x was an instance of c at previous time. Formally,

delete_c(X,T)
E: $\forall$X,T(delete_c(X,T) $\rightarrow \neg$c(X,T))
A: $\forall$X,T(delete_c(X,T) $\rightarrow$ c(X,T-1))

Let attname be an attribute of class c. We use a three-term predicate insert_c_attname(X,Y,T) to denote the corresponding attribute insertion primitive structural event. Its effect consists of the addition of value y for attribute attname of object x at the time instant t (when the event occurs). This event can only be applied if y was not a value for attribute attname of instance x at previous

time. Formally:

insert_c_attname(X,Y,T)

E: $\forall$X,Y,T(insert_c_attname(X,Y,T) $\rightarrow$ attname(X,Y,T))

A: if attname is single valued, then

$\forall$X,Y,T(insert_c_attname(X,Y,T) $\rightarrow \neg\exists$Z attname(X,Z,T-1))

if attname is multivalued, then

$\forall$X,Y,T(insert_c_attname(X,Y,T) $\rightarrow \neg$attname(X,Y,T-1))

Note that if attname is single valued then the insertion of an attribute value at time t is conditioned to the inexistence of any value for that attribute at time t-1.

Attribute update and deletion primitive structural events are defined in a similar way. Its effect and applicability axioms are, hopefully, self-explanatory.

delete_c_attname(X,T)/delete_c_attname(X,Y,T)

E: if attname is single valued, then

$\forall$X,T(delete_c_attname(X,T) $\rightarrow \neg\exists$Y attname(X,Y,T))

if attname is multivalued, then

$\forall$X,Y,T(delete_c_attname(X,Y,T) $\rightarrow \neg$attname(X,Y,T))

A: if attname is singlevalued, then

$\forall$X,T(delete_c_attname(X,T) $\rightarrow \exists$Y attname(X,Y,T-1))

if attname is multivalued, then

$\forall$X,Y,T(delete_c_attname(X,Y,T) $\rightarrow$ attname(X,Y,T-1))

Note that if attname is single valued then predicate delete_c_attname is two-term because it is not necessary to indicate the deleted value.

update_c_attname(X,Y,T)

E: if attname is single valued, then

$\forall$X,Y,T(update_c_attname(X,Y,T) $\rightarrow$ attname(X,Y,T))

A: if attname is single valued, then

$\forall$X,Y,T(update_c_attname(X,Y,T) $\rightarrow \exists$Z attname(X,Z,T-1) $\wedge$ Z$\neq$Y)

Note that update_c_attname has been defined only when attname is single valued.

Finally, we have to define what happens with objects and attributes existing (or not existing) at previous state of the IB that are not affected by the primitive structural events of a transaction. This is accomplished by the following frame axioms [VeF85, BMR95]:

$\forall$X,T ($\neg$c(X,T-1) $\wedge \neg$ insert_c(X,T) $\rightarrow \neg$c(X,T))

$\forall$X,T (c(X,T-1) $\wedge \neg$ delete_c(X,T) $\rightarrow$ c(X,T))

if attname is single valued

$\forall$X,Y,T ($\neg$attname(X,Y,T-1) $\wedge \neg$insert_c_attname(X,Y,T) $\wedge$ $\neg$update_c_attname(X,Y,T) $\rightarrow \neg$attname(X,Y,T))

$\forall$X,Y,T (attname(X,Y,T-1) $\wedge \neg$delete_c_attname(X,T) $\wedge$ $\neg \exists$Z(update_c_attname(X,Z,T)) $\rightarrow$ attname(X,Y,T))

if attname is multivalued:

$\forall$X,Y,T ($\neg$attname(X,Y,T-1) $\wedge \neg$insert_c_attname(X,Y,T) $\rightarrow$ $\neg$attname(X,Y,T))

$\forall$X,Y,T (attname(X,Y,T-1) $\wedge \neg$delete_c_attname(X,Y,T) $\rightarrow$

attname(X,Y,T))

5. Relationship among primitive structural events

As mentioned before, we consider that a transaction consists of a number of primitive structural events. This section explains how to determine the conditions such events must satisfy to maintain the IB consistent with respect to the transition integrity constraints defined by our temporal features.

More specifically, we present a set of theorems defining the relationships between primitive structural events that maintain the IB consistency. If a transaction respects all the conditions imposed by the theorems then the IB satisfies the transition integrity constraints in the resulting state. The complete set of theorems, together with an intuitive explanation about their meaning, can be found in the Appendix. They can be proved from the features definition given in sections 2 and 3 and the primitive structural event axioms given in section 4. For space reasons we are not able to explain all the theorems in detail. Instead, we will show its application to transaction specification.

5.1 Application to transaction specification

Our theorems define constraints on primitive structural events that constitute transactions. These constraints are useful during transaction execution and during transaction definition process. In this last case, they may be applied to assist the process in two manners: *transaction checking* or *transaction repairing*.

Transaction checking consists of: given a transaction specified by the designer, establish whether or not this transaction is consistent with respect to the integrity constraints.

Transaction repairing consists of: given an inconsistent transaction (according to transaction checking) obtain one or more sets of primitive structural events such that once added to it constitute consistent transactions. If no such sets of events exist, the transaction is not repairable.

In the next paragraphs, we illustrate the use of the theorems for transaction checking and transaction repairing by means of several examples. All theorems referred in the analysis of the following examples appear in the Appendix of the paper.

Consider the following class definition:

```
class vendor
    name: string; (always initially valued)
    hasAssigned: set of client; (never initially valued)
```

and a transaction specified with the purpose of inserting a new vendor in the IB:

```
insert_vendor(X,T)
```

Transaction checking establishes that the above transaction is inconsistent. In fact, it violates a single constraint defined by theorem A7 (see appendix). The constraint is applicable to attributes declared as `always initially valued` and when applied to attribute `name` it comes down to:

$\forall$X,T(insert_vendor(X,T) $\rightarrow$ $\exists$Y insert_vendor_name(X,Y,T))

As a consequence, our transaction is inconsistent because it inserts a vendor without

inserting a value for its name, being name an always initially valued attribute.

The same constraint allows to perform transaction repairing. From it, we deduce that the transaction can be repaired by adding to it the primitive structural event insert_vendor_name(X,Y,T).

The above example shows a very simple case of transaction repairing. In other cases, several theorems have to be applied in order to perform the repair. We illustrate this in next example. Consider the previous definition of class vendor and the definition of class client that follows:

```
class client
    name: string; (always initially valued)
    assignedTo: vendor inverse of hasAssigned; (constant)
```

A transaction that inserts a value for attribute hasAssigned to a vendor could be:

```
insert_vendor_hasAssigned(X,Y,T)
```

This transaction violates the constraint specified by theorem A10 which is relevant to this case because attribute assignedTo (inverse of hasAssigned) is constant:

$$\forall X,Y,T(\texttt{insert\_vendor\_hasAssigned}(X,Y,T) \rightarrow \texttt{insert\_client}(Y,T) \wedge \texttt{insert\_client\_assignedTo}(Y,X,T))$$

This constraint means that when a value is inserted for hasAssigned, this value must be inserted as an object in class client and the corresponding value for attribute assignedTo in class client must also be inserted. Thus, to repair the initial transaction we must add to it two primitive structural events obtaining:

```
insert_vendor_hasAssigned(X,Y,T)
insert_client(Y,T)
insert_client_assignedTo(Y,X,T)
```

Our transaction repairing is still not complete. Now, we insert a client in the IB without inserting a value for its name (always initially valued attribute). As in our previous example, theorem A7 of the appendix, deals with this case and from it we deduce that we also have to include the primitive structural event insert_client_name(Y,Z,T) obtaining:

```
insert_vendor_hasAssigned(X,Y,T)
insert_client(Y,T)
insert_client_assignedTo(Y,X,T)
insert_client_name(Y,Z,T)
```

In section 3.8, it has been shown that two recent interpretations of aggregation can be fully characterized by our temporal features. Let's analyze the impact of aggregation on transaction repairing by means of the "division part of company" example introduced in 3.8. Consider a transaction that deletes a company:

```
delete_company(X,T)
```

1- From theorem A17 of the Appendix we have that all existent values of attribute hasDivisions of the company have also to be deleted together with it:

$$\forall X,Y,T,TI(TI \in \texttt{mi}(X,\texttt{company}) \wedge \texttt{belongsTo}(T\text{-}1,TI) \wedge \texttt{delete\_company}(X,T) \wedge \texttt{hasDivisions}(X,Y,T\text{-}1) \rightarrow$$

delete_company_hasDivisions(X,Y,T))

Applying this to transaction repairing, we obtain the new transaction:

delete_company(X,T)
∀Y (hasDivisions(X,Y,T-1) →
delete_company_hasDivisions(X,Y,T))

2- Now, the new transaction violates a constraint that corresponds to theorem A20 which indicates that when a value of an attribute is deleted, either this value is deleted from its class or the inverse attribute value has to be deleted or the inverse attribute has to be updated:

∀X,Y,T(delete_company_hasDivisions(X,Y,T) →
delete_division(Y,T) ∨
delete_division_belongsTo(Y,T)∨
∃Z update_division_belongsTo(Y,Z,T) ∧ X≠Z)

Applying this to transaction repairing, three possible transactions appear:

delete_company(X,T)
∀Y(hasDivisions(X,Y,T-1) →
delete_company_hasDivisions(X,Y,T))
∀Y(hasDivisions(X,Y,T-1) → delete_division(Y,T))

or

delete_company(X,T)
∀Y(hasDivisions(X,Y,T-1) →
delete_company_hasDivisions(X,Y,T))
∀Y(hasDivisions(X,Y,T-1)→delete_division_belongsTo(Y,T))

or

delete_company(X,T)
∀Y(hasDivisions(X,Y,T-1) →
delete_company_hasDivisions(X,Y,T))
∀Y(hasDivisions(X,Y,T-1) →
update_division_belongsTo(Y,Z,T)∧ X≠Z)

3- The first possibility has still to be repaired according to theorem A16 which specifies that the deletion of an object from a class implies the simultaneous deletion of all its attributes:

∀Y,T(delete_division(Y,T)→
delete_division_belongsTo(Y,T))

Then, the transaction must be repaired into:

delete_company(X,T)
∀Y(hasDivisions(X,Y,T-1) →
delete_company_hasDivisions(X,Y,T))
∀Y(hasDivisions(X,Y,T-1) → delete_division(Y,T))
∀Y(hasDivisions(X,Y,T-1) →
delete_division_belongsTo(Y,T))

which is a consistent transaction. This result is coherent with the interpretation taken for aggregation in section 3.8: the deletion of the whole induces the deletion of the parts.

4- The second possibility has also to be repaired according to theorem A3 which specifies that when a `single permanent existence interval` attribute takes a value, it keeps on having value until the end of the object membership interval:

$\forall$`Y,T(delete_division_belongsTo(Y,T)`$\rightarrow$
`delete_division(Y,T))`

Then, the transaction is transformed into:

`delete_company(X,T)`
$\forall$`Y(hasDivisions(X,Y,T-1)` $\rightarrow$
`delete_company_hasDivisions(X,Y,T))`
$\forall$`Y(hasDivisions(X,Y,T-1)`$\rightarrow$`delete_division_belongsTo(Y,T))`
$\forall$`Y(hasDivisions(X,Y,T-1)`$\rightarrow$`delete_division(Y,T))`

which coincides with the transaction obtained in previous step.

5- The third possibility violates the constraint specified by theorem A6 because `belongsTo` is `non-modifiable`:

$\forall$`T (`$\neg\exists$`Y,Z update_division_belongsTo(Y,Z,T))`

As can be seen, this third possibility is not repairable according to the previous constraint.

Our last example illustrates the impact of class hierarchy definitions on transactions. Consider the example described in section 2 (also depicted in figure 1) and consider a transaction that inserts a new person in the IB and inserts it simultaneously as a dead person:

`insert_person(X,T)`
`insert_dead(X,T)`

As `dead` is `never initially member`, according to theorem C4 when an object is inserted in the corresponding superclass it cannot be inserted in the subclass at the same time:

$\forall$`X,T(insert_person(X,T)` $\rightarrow \neg$`insert_dead(X,T))`

Thus, transaction checking would establish that the transaction is inconsistent. From the constraint above, we also conclude that it is not repairable by adding new primitive structural events to it. This means that there is not a way of inserting a new person as a dead person in the IB.

6. Conclusions

We have described six temporal features that correspond to common transition integrity constraints. Three of them (permanent instances, initial membership, membership intervals) are related to classes, constraining the way how their populations can evolve through time. The other three features (initial value, existence intervals, changeability) are related to attributes, constraining the way how attribute values can change through time. We have shown that these features are orthogonal, and that the features of an attribute are orthogonal to those of its inverse. Our features may be helpful in clarifying the meaning of some concepts. In particular, we have shown that they capture in a simple way the meaning of aggregation/composition, as defined in two recent conceptual modeling languages.

The proposed features are language-independent and, thus, they can be used in any conceptual model. The features correspond to transition integrity constraints that

appear in many models, and we have made their specification simple and practical. One possible extension of this work, however, would be the identification and formalization of other important temporal features.

We have formalized the features declaratively and procedurally. The procedural formalization assumes that the Information Base is changed by a set of primitive structural events, and determines the conditions such events must satisfy. The primitive structural events, however, are 'too much' primitive to be practical in behavioural models. In this respect, we would like to determine a set of more complex structural events, consisting of a composition of primitive ones, and meaningful at the behavioural level.

Appendix

Class Theorems

C1. Let c be a class with permanent instances, then:
$\forall T\ (\neg\exists X\ delete\_c(X,T))$
/* a permanent object cannot be deleted */
C2. Let c be a class and let c1 be an specialization of c, then:
$\forall X,T,TI(delete\_c(X,T) \wedge TI \in mi(X,c1) \wedge belongsTo(T-1,TI) \rightarrow delete\_c1(X,T))$
/* the deletion of an object from a class c imposes the deletion of the object from all the subclasses of c*/
C3. Let c be a class and let c1 be an specialization of c, declared as always initially member, then:
$\forall X,T(insert\_c(X,T) \wedge \neg object(X,T-1) \rightarrow insert\_c1(X,T))$
/*the insertion of a new object in a class c imposes its simultaneous insertion in all subclasses of c which are always initially member*/
C4. Let c be a class and let c1 be an specialization of c defined as never initially member then:
$\forall X,T(insert\_c(X,T) \wedge \neg object(X,T-1) \rightarrow \neg insert\_c1(X,T))$
/*if subclass c1 is never initially member then when a new object is inserted in the superclass c it cannot be inserted in subclass c1 at the same time*/
C5. Let c be a class and let c1 be a descendent of c in a specialization hierarchy and declared as single permanent membership interval, then:
$\forall X,T(delete\_c1(X,T) \rightarrow delete\_c(X,T))$
/*if a class c1 is single permanent membership interval then its objects remain in this class until the end of their lifespan. So, the deletion of an object from this class implies the simultaneous deletion of the object from any ascendent of c1*/
C6. Let c1 be an specialization declared as single non-permanent membership then:
$\forall X,T,T1,TI(TI \in mi(X,c1) \wedge T > endsAt(TI) \rightarrow \neg insert\_c1(X,T))$
/*If c1 is single non-permanent membership interval then its objects

have a single membership interval . So, if an object exists in this class during an interval TI it cannot be inserted again in this class after TI (an insertion starts a new membership interval)*/

C7. Let c be a class partitioned into classes c1,..., ci, ...,cn, then:

$\forall X,T(\texttt{delete\_ci}(X,T) \rightarrow \texttt{delete\_c}(X,T) \vee \texttt{insert\_c1}(X,T) \vee ... \vee \texttt{insert\_ci-1}(X,T) \vee \texttt{insert\_ci+1}(X,T) \vee ... \vee \texttt{insert\_cn}(X,T))$

/*The deletion of an object from a subclass implies its deletion from the superclass or its insertion in any other subclass of the same partition*/

Attribute Theorems

A1. Let attname be a single valued attribute defined on c and declared as `always initially valued` and (`single permanent existence interval` or `single non-permanent existence interval`), then:

$\forall X,Y,T(\texttt{insert\_c\_attname}(X,Y,T) \rightarrow \texttt{insert\_c}(X,T))$

/*If an attribute is `always initially valued` and has a single existence interval (either permanent or not) then its unique insertion has to be done at the same time when the object is inserted in the class*/

A2. Let attname be a multivalued attribute defined on c and defined as `insertions non-allowed`, then:

$\forall X,Y,T(\texttt{insert\_c\_attname}(X,Y,T) \rightarrow \texttt{insert\_c}(X,T))$

/*if attname is `insertions non-allowed` then the attribute has to take all its values when the object is inserted in the class*/

A3. Let attname be a single valued attribute defined on c and declared as `single permanent existence interval`, then:

$\forall X,T(\texttt{delete\_c\_attname}(X,T) \rightarrow \texttt{delete\_c}(X,T))$

/*If an attribute is `single permanent existence interval` then once it takes a value, it will keep on having some value until the end of the object membership interval. So, the deletion of the attribute value must be simultaneous with the deletion of the object from the class*/

A4. Let attname be a single valued attribute defined on c and declared as `single non-permanent existence interval`, then:

$\forall X,Y,T,T1,TI(TI \in \texttt{mi}(X,c) \wedge \texttt{belongsTo}(T1,TI) \wedge \texttt{attname}(X,Y,T1) \wedge \texttt{belongsTo}(T,TI) \wedge T1 \leq T \rightarrow \neg\exists Z\, \texttt{insert\_c\_attname}(X,Z,T))$

/*If an attribute has a `single existence interval` then during an object membership interval there is at most one insertion for this attribute.*/

A5. Let attname be a multivalued attribute defined on c and declared as `deletions non-allowed`, then:

$\forall X,Y,T(\texttt{delete\_c\_attname}(X,Y,T) \rightarrow \texttt{delete\_c}(X,T))$

/*If an attribute is `deletions non-allowed` then it will keep on having its values until the end of the object membership interval. So, the deletion of the attribute values must be simultaneous with the deletion of the object from the class*/

A6. Let attname be a singlevalued attribute defined on c and declared as `non-modifiable`, then:

$\forall T\ (\neg\exists X,Y\ \texttt{update\_c\_attname}(X,Y,T)$

/*If an attribute is non-modifiable then an event update_c_attname can not occur*/

A7. Let attname be an attribute defined on c and declared as always initially valued, then:

$\forall X,T(\texttt{insert\_c}(X,T) \rightarrow \exists Y\ \texttt{insert\_c\_attname}(X,Y,T))$

/*If an attribute is always initially valued then when an object is inserted in the class, a value for this attribute must be inserted at the same time*/

A8. Let attname be an attribute defined on c and declared as never initially valued, then:

$\forall X,T(\texttt{insert\_c}(X,T) \rightarrow \neg\exists Y\ \texttt{insert\_c\_attname}(X,Y,T))$

/*If an attribute is never initially valued then when an object is inserted in the class, a value for this attribute can not be inserted at the same time*/

A9. Let attname be a single valued attribute defined on c, then:

$\forall X,Y,T(\texttt{insert\_c}(X,T) \wedge \texttt{insert\_c\_attname}(X,Y,T) \rightarrow \neg\exists Z\ \texttt{insert\_c\_attname}(X,Z,T) \wedge Y \neq Z)$

/*If an attribute is single valued then two different values can not be simultaneously inserted*/

A10.Let attname be an attribute defined on c over an object class c1 and being invattname the inverse attribute of attname. If invattname is constant, then:

$\forall X,Y,T(\texttt{insert\_c\_attname}(X,Y,T) \rightarrow \texttt{insert\_c1}(Y,T)) \wedge \texttt{insert\_c1\_invattname}(Y,X,T))$

/*If invattname is constant then when a value is inserted for attname, this value must be inserted as an object in c1 and the corresponding value for the inverse attribute invattname must be inserted.*/

A11.Let attname be a singlevalued attribute defined on c over an object class c1 and being invattname the inverse attribute of attname. If invattname is constant, then:

$\forall X,Y,T(\texttt{update\_c\_attname}(X,Y,T) \rightarrow \texttt{insert\_c1}(Y,T)) \wedge \texttt{insert\_c1\_invattname}(Y,X,T))$

/*If invattname is constant then when a value is given for attname, this value must be inserted as an object in c1 and the corresponding value for the inverse attribute invattname must be inserted.*/

A12.Let attname be an attribute defined on c over an object class c1 and being invattname the inverse attribute of attname. If invattname is never initially valued, then:

$\forall X,Y,T(\texttt{insert\_c\_attname}(X,Y,T) \rightarrow \neg\texttt{insert\_c1}(Y,T))$

/*If invattname is never initially valued then objects of class c1 must exist before being assigned as values of attribute attname.*/

A13.Let attname be a single valued attribute defined on c over an object class c1 and being invattname the inverse attribute of attname. If invattname is never initially valued, then:

$\forall X,Y,T(\texttt{update\_c\_attname}(X,Y,T) \rightarrow \neg\texttt{insert\_c1}(Y,T))$

/*If invattname is never initially valued then objects of class c1 must exist before being assigned as values of attribute attname.*/

A14.Let attname be an attribute defined on c over an object class c1 and being invattname the inverse attribute of attname. If invattname is single valued and not

constant, then:
$\forall X,Y,T(\texttt{insert\_c\_attname}(X,Y,T) \rightarrow$
$\texttt{insert\_c1\_invattname}(Y,X,T) \vee \texttt{update\_c1\_invattname}(Z,X,T))$
/*If invattname is not constant and single valued then when a value is given for attname, the corresponding value for the inverse attribute invattname must be given through an insertion or through an update.*/
A15.Let attname be an attribute defined on c over an object class c1 and being invattname the inverse attribute of attname. If invattname is multivalued and not constant, then:
$\forall X,Y,T(\texttt{insert\_c\_attname}(X,Y,T) \rightarrow$
$\texttt{insert\_c1\_invattname}(Y,X,T))$
/*Same as before, but multivalued attributes can not be updated*/
A16.Let c be an object class and attname a single valued attribute defined on c, then:
$\forall X,Y,T(\texttt{delete\_c}(X,T) \rightarrow \texttt{delete\_c\_attname}(X,T))$
/*The deletion of an object from a class implies the simultaneous deletion of all its single valued attributes corresponding to this class*/
A17. Let c be an object class and attname a multivalued attribute defined on c, then:
$\forall X,Y,T,TI(TI \in \texttt{mi}(X,c) \wedge \texttt{belongsTo}(T\text{-}1,TI) \wedge \texttt{delete\_c}(X,T) \wedge$
$\texttt{attname}(X,Y,T\text{-}1) \rightarrow \texttt{delete\_c\_attname}(X,Y,T))$
/*Same as before for multivalued attributes*/
A18. Let attname be a single valued attribute defined on c over an object class c1 and let invattname be the inverse attribute of attname and single valued, then:.
$\forall X,Y,T,TI(TI \in \texttt{mi}(X,c) \wedge \texttt{belongsTo}(T\text{-}1,TI) \wedge$
$\texttt{delete\_c1\_invattname}(Y,T) \wedge \texttt{attname}(X,Y,T\text{-}1) \rightarrow$
$\texttt{delete\_c}(X,T) \vee \texttt{delete\_c\_attname}(X,T) \vee$
$\exists Z\ \texttt{update\_c\_attname}(X,Z,T) \wedge Y \neq Z)$
/*When the value of an attribute is deleted, either this value (which is itself an object) is deleted from its class or the inverse attribute value has to be deleted or the inverse attribute value has to be updated*/
A19.Let attname be a single valued attribute defined on c over an object class c1 and let invattname be the inverse attribute of attname and single valued, then:.
$\forall X,Y,Z,T,TI(TI \in \texttt{mi}(X,c) \wedge \texttt{belongsTo}(T\text{-}1,TI) \wedge$
$\texttt{update\_c1\_invattname}(Y,Z,T) \wedge \texttt{attname}(X,Y,T\text{-}1) \rightarrow$
$\texttt{delete\_c}(X,T) \vee \texttt{delete\_c\_attname}(X,T) \vee$
$\exists Z\ \texttt{update\_c\_attname}(X,Z,T) \wedge Y \neq Z)$
/*Same as before, but the attribute is updated*/
A20.Let attname be a single valued attribute defined on c over an object class c1 and let invattname be the inverse attribute of attname and multivalued, then:.
$\forall X,Y,T(\texttt{delete\_c1\_invattname}(Y,X,T) \rightarrow$
$\texttt{delete\_c}(X,T) \vee \texttt{delete\_c\_attname}(X,T) \vee$
$\exists Z\ \texttt{update\_c\_attname}(X,Z,T) \wedge Y \neq Z)$
/*Same as 18, but invattname is multivalued*/
A21.Let attname be a multivalued attribute defined on c over an object class c1 and let invattname be the inverse attribute of attname and single valued, then:.
$\forall X,Y,T,TI(TI \in \texttt{mi}(X,c) \wedge \texttt{belongsTo}(T\text{-}1,TI) \wedge$

delete_c1_invattname(Y,T) ∧ attname(X,Y,T-1) →
delete_c(X,T) ∨ delete_c_attname(X,Y,T))

/*Same as 18, but attname is multivalued*/

A22. Let attname be a multivalued attribute defined on c over an object class c1 and let invattname be the inverse attribute of attname and single valued, then:.

∀X,Y,Z,T,TI(TI∈mi(X,c) ∧ belongsTo(T-1,TI) ∧
update_c1_invattname(Y,Z,T) ∧ attname(X,Y,T-1) →
delete_c(X,T) ∨ delete_c_attname(X,Y,T))

/*Same as 19, but attname is multivalued*/

A23. Let attname be a multivalued attribute defined on c over an object class c1 and let invattname be the inverse attribute of attname and multivalued, then:.

∀X,Y,T(delete_c1_invattname(Y,X,T) →
delete_c(X,T) ∨ delete_c_attname(X,Y,T))

/*Same as 18, but attname and invattname are multivalued*/

References

[BiD94] Bidoit,N.; De Amo,S. "Contraintes dynamiques d'inclusion et schémas transactionnels". Ingénierie des systèmes d'information, Vol.2, No. 1, pp. 83-113.

[BMR95] Borgida,A.;Mylopoulos,J.;Reiter,R. "On the Frame Problem in Procedure Specifications",IEEE Trans. on SE,Oct., pp. 785-798.

[BoC95] Bourdeau,R.H.; Cheng,B.H.C. "A Formal Semantics for Object Model Diagrams", IEEE Tran. on Software Engineering, Vol. 21, No. 10, pp. 799-821.

[Bor85] Borgida,A. "Features of Languages for the Development of Information Systems at the Conceptual Level", IEEE Software, Jan., pp. 63-72.

[CoD94] Cook,S.; Daniels,J. "Designing Object Systems. Object-Oriented Modeling with Syntropy", Prentice Hall.

[dCF92] de Champeaux,D.; Faure,P. "A comparative study of object-oriented analysis methods", JOOP, March/April, pp.21-33.

[DHR91] Dubois,E., Hagelstein,J.; Rifaut,A. "A formal language for the requirements engineering of computer systems", in "From Natural Language Processing to Logic for Expert Systems", Wiley, pp. 269-345.

[EKW92] Embley,D.W.; Kurtz,B.D.; Woodfield,S.N. "Object-Oriented Systems Analysis. A Model-Driven Approach", Prentice-Hall, Inc.

[GKB82] Gustaffson,M.R.; Karlsson,T.; Bubenko jr.J.A. "A Declarative Approach to Conceptual Information Modelling", in Information Systems Design Methodologies: A comparative Review", North-Holland, pp. 93-142.

[GMB94] Greenspan,S.; Mylopoulos,J.; Borgida,A. "On Formal Requirements Modeling Languages: RML Revisited", Proc. 16th. Int. Conf. Software Engineering, Sorrento, Italy, pp. 135-147.

[GSR96] Gottlob,G.; Schrefl,M.; Röck,B. "Extending Object-Oriented Systems with Roles", ACM TOIS, Vol.14,No.3, pp. 268-296.

[HaM81] Hammer,M.; McLeod,D. "Database Description with SDM: A Semantic Database Model", ACM TODS, Vol.6,No.3, September, pp. 351-386.

[ISO82] ISO/TC97/SC5/WG3. "Concepts and Terminology for the Conceptual Schema and the Information Base", ed. J.J. van Griethuysen.

[JSH+96] Jungclaus,R.; Saake,G.; Hartmann,T.; Sernadas,C. "TROLL-A Language for Object-Oriented Specification of Information Systems", ACM TOIS,Vol.14,No.2, April, pp. 175-211.

[KBG89] Kim,W.; Bertino,E.; Garza,J.F. "Composite Objects Revisited", Proc. OOPSLA 89, pp. 337-347.

[Kun84] Kung,C. "A Temporal Framework for Information Systems Specification and Verification", Ph.D Thesis, The University of Trondheim, Norway.

[LEW93] Liddle,S.W.; Embley,D.E.; Woodfield,S.N. "Cardinality constraints in semantic data models", Data&Knowledge Engineering 11 (1993), pp. 235-270.

[MaO95] Martin,J.; Odell,J. "Object-Oriented Methods. A Foundation", Prentice Hall.

[Mots93] Motschnig-Pitrik,R. "The Semantics of Parts Versus Aggregates in Data/Knowlege Modelling", Proc. of the CAiSE'93, LNCS 685, Springer, pp. 352-373.

[Per90] Pernici,B. "Objects with Roles", Proc. ACM Conf. on Office Information Systems, ACM,New York, 205-215.

[Rat97] Rational Software Corporation, "Unified Modeling Language (UML)", Version 1.0, January.

[RBP+91] Rumbaugh,J.; Blaha,M.; Premerlani,W.; Eddy,F.; Lorensen,W. "Object-Oriented Modeling and Design", Prentice Hall.

[SFN+84] Schiel,U., Furtado,A.L., Neuhold,E.J.; Casanova,M.A. "Towards Multi-level and Modular Conceptual Schema Specifications", Information Systems, Vol.9, No.1, pp. 43-57.

[Sto93] Storey, V.C. "Understanding Semantic Relationships", The VLDB Journal, Vol.2,No.4,Oct., pp. 455-488.

[VeF85] Veloso,P.A.S.; Furtado,A.L. "Towards simpler and yet complete formal specifications", In "Information Systems: Theoretical and Formal Aspects", North-Holland, pp. 175-190.

[VeV82] Verheijen,G.M.A.; Van Bekkum, J. "NIAM: An Information Analysis Method", in "Information Systems design Methodologies: A Comparative Review", North-Holland, pp.537-589.

[WCH87] Winston,M.E.; Chaffin,R.; Herrmann,D. "A taxonomy of part-whole relations", Cognitive Science, 11, pp.417-444.

[WMW89] Wieringa,R.; Meyer,J-J.; Weigand,H. "Specifying dynamic and deontic integrity constraints", Data & Knowledge Engineering, 4, pp.157-189.