Temporal Constraints in Non-Temporal Data Modelling Languages

Peter M^{c}Brien

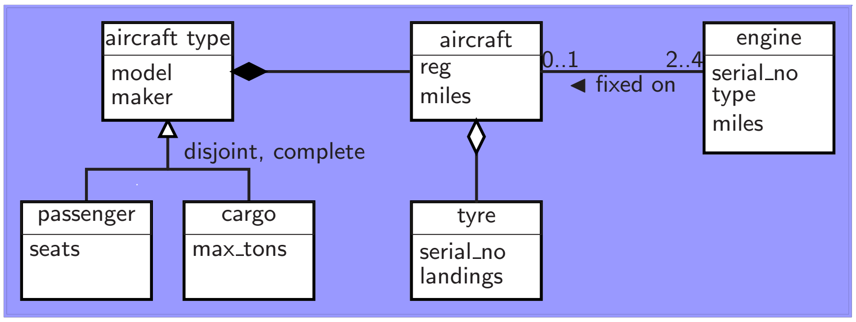
Dept. Computing, Imperial College London, London SW7 2AZ
email: pjm@doc.ic.ac.uk

Abstract. It is common to find that the definition or common usage of a data modelling language causes there to be restrictions placed on the evolution of data values that are associated with schemas expressed in that modelling language. This paper terms these restrictions *temporal constraints*, and defines three types of temporal constraint which are argued to be useful modelling concepts, capturing important real-world semantics about objects and their relationships. By reviewing how these temporal constraints are implied by either the definition or usage of UML and the relational modelling languages, this paper will use the temporal constraints to give precise definitions of modelling concepts that to date have been left only vaguely and partially understood. It will also consider the implementation of these constraints in SQL.
keywords: data modelling, dynamic behaviour, conceptual modelling, temporal constraints

1 Introduction

This paper reviews what will be termed the **temporal constraints** (which are also known as **dynamic constraints**) of data modelling languages, which we define to mean the restrictions that are placed on the evolution of the extent of a schema expressed in a data modelling language. In particular, this paper describes constraints on the evolution of the extent in **transaction time** [1] which may be implemented without the necessity of keeping a transaction time database. Hence we are considering temporal constraints which may be applied in a non-temporal data modelling language.

To illustrate the concept of temporal constraints in non-temporal data modelling languages, consider the UML schema in Fig. 1. A normal interpretation of this schema is that once an instance x has been created of the passenger class, then it will not be possible that later the same x appears as an instance of cargo class. More generally, the normal interpretation of object oriented modelling languages is that an object identifier cannot be associated with two different classes and refer to the same thing. However, this interpretation is not to be found in the definition of the UML modelling language [2], and indeed some research work has been conducted into programming languages which remove this restriction [3]. Another example is found in the UML **association** construct, where the definition of UML makes it unclear if a instance tyre could exist after the deletion of aircraft, and if is does, whether it could then be assigned to another aircraft.



class: ⟨⟨aircraft_type⟩⟩
attribute: ⟨⟨aircraft_type, model⟩⟩
attribute: ⟨⟨aircraft_type, maker⟩⟩
class: ⟨⟨passenger⟩⟩
attribute: ⟨⟨passenger, seats⟩⟩
class: ⟨⟨cargo⟩⟩
attribute: ⟨⟨cargo, max_tons⟩⟩
generalisation: ⟨⟨aircraft_type, {disjoint, complete}, passenger, cargo⟩⟩
class: ⟨⟨aircraft⟩⟩
attribute: ⟨⟨aircraft, reg⟩⟩
attribute: ⟨⟨aircraft, miles⟩⟩

composition: ⟨⟨_, aircraft_type, aircraft, 1..1, 0..N⟩⟩
class: ⟨⟨tyre⟩⟩
attribute: ⟨⟨tyre, serial_no⟩⟩
attribute: ⟨⟨tyre, landings⟩⟩
aggregation: ⟨⟨_, aircraft, tyre, 0..1, 0..N⟩⟩
class: ⟨⟨engine⟩⟩
attribute: ⟨⟨engine, serial_no⟩⟩
attribute: ⟨⟨engine, type⟩⟩
attribute: ⟨⟨engine, miles⟩⟩
association: ⟨⟨fixed_on, aircraft, engine, 0..1, 2..4⟩⟩

Fig. 1. S^{uml}: A UML schema for a database of a aircraft fleet, together with its description as a set of schema objects

Related work will be considered in detail at the end of the paper in Section 4. One contribution of this paper is to define in Section 3 a set of temporal constraints that restrict the evolution of instances of a schema expressed (and stored as) a non-temporal data modelling language. The definitions are made in a manner that allows them to be defined on any data modelling language that fits a certain structure that this paper reviews in Section 2, which has already been shown [4,5] to be sufficient to support the relational, UML, ER, ORM and XML modelling languages. A second contribution of the paper is to discuss the extent to which these temporal constraints are (sometimes rather vaguely) already implied by the definitions of data modelling languages, by discussing in depth how the temporal constraints can be applied to UML and the relational data model.

An advantage gained in defining precisely the temporal modelling constraints and identifying temporal constraints in schemas is that it reveals where there is the possibility of inconsistencies when data is transferred between the schemas of information systems that have been built around different data modelling languages. For example, the relational schema in Fig. 2 would be regarded as equivalent to the UML schema in Fig. 1 under conventional UML to relational mapping approaches [6]. Indeed, at any one time, it will be possible to map instances of one schema into instances of the other schema. However, there are evolutions of the instances of the relational schema that would not be permitted in the instances when mapped into the UML schema. For example, in the relational schema it would be possible to delete an entry x from the passenger table, and insert it into the cargo table, whilst leaving the instance x unchanged in aircraft_type, and hence have the *same* instance of an aircraft type change from passenger to cargo types. As already discussed, this is normally not permitted in object-oriented models.

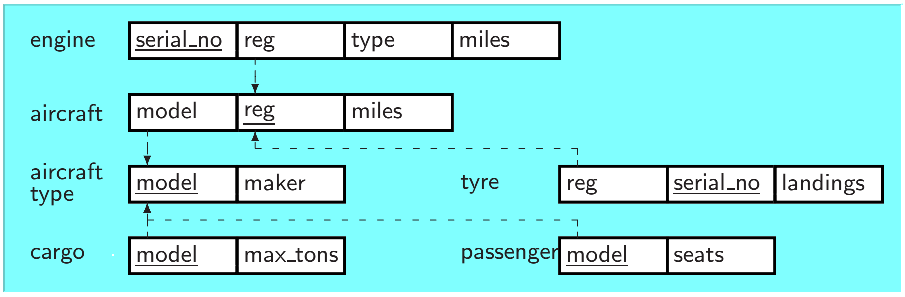


table:⟨⟨aircraft_type⟩⟩
column:⟨⟨aircraft_type, model⟩⟩
column:⟨⟨aircraft_type, maker⟩⟩
table:⟨⟨passenger⟩⟩
column:⟨⟨passenger, model⟩⟩
column:⟨⟨passenger, seats⟩⟩
table:⟨⟨cargo⟩⟩
column:⟨⟨cargo, model⟩⟩
column:⟨⟨cargo, max_tons⟩⟩
table:⟨⟨aircraft⟩⟩
column:⟨⟨aircraft, model⟩⟩
column:⟨⟨aircraft, reg⟩⟩
column:⟨⟨aircraft, miles⟩⟩
table:⟨⟨tyre⟩⟩
column:⟨⟨tyre, reg⟩⟩
column:⟨⟨tyre, serial_no⟩⟩
column:⟨⟨tyre, landings⟩⟩

table:⟨⟨engine⟩⟩
column:⟨⟨engine, serial_no⟩⟩
column:⟨⟨engine, reg⟩⟩
column:⟨⟨engine, type⟩⟩
column:⟨⟨engine, miles⟩⟩
primary_key:⟨⟨engine, serial_no⟩⟩
primary_key:⟨⟨aircraft, reg⟩⟩
primary_key:⟨⟨aircraft_type, model⟩⟩
primary_key:⟨⟨cargo, model⟩⟩
primary_key:⟨⟨passenger, model⟩⟩
primary_key:⟨⟨tyre, serial_no⟩⟩
foreign_key:⟨⟨⟨⟨aircraft, model⟩⟩, ⟨⟨aircraft_type, model⟩⟩⟩⟩
foreign_key:⟨⟨⟨⟨passenger, model⟩⟩, ⟨⟨aircraft_type, model⟩⟩⟩⟩
foreign_key:⟨⟨⟨⟨cargo, model⟩⟩, ⟨⟨aircraft_type, model⟩⟩⟩⟩
foreign_key:⟨⟨⟨⟨tyre, reg⟩⟩, ⟨⟨aircraft, reg⟩⟩⟩⟩
foreign_key:⟨⟨⟨⟨engine, reg⟩⟩, ⟨⟨aircraft, reg⟩⟩⟩⟩

Fig. 2. S_1^{rel}: A relational schema for a database of a aircraft fleet, together with its description as a set of schema objects

2 Models and Schemas

Using the notation of AutoMed [7], a modelling language, or **model**, m contains a set of modelling constructs, where each **construct** c is used to represent some class of data structure that holds set, bag or list of data values, and/or constraints on sets, bags or lists of data values. A **schema** s comprises of a set of schema objects, where each **schema object** o is typed to some construct c. To date, almost without exception, researchers have considered that any given information system uses a single modelling language. Such single modelling language schemas can then be described by $s^m = \{c_1{:}\langle\langle o_1\rangle\rangle, c_1{:}\langle\langle o_2\rangle\rangle, \ldots, c_n{:}\langle\langle o_{m-1}\rangle\rangle, c_n{:}\langle\langle o_m\rangle\rangle\}$.

When a schema object has some collection of data associated with it, we call the data the **extent** of the schema object. The data associated with such **extensional** schema objects o can be returned by the function $Ext(o)$. Using the classification of modelling constructs in [8], there are three classes of construct for which schema objects carry an extent:

- **nodal** constructs may be used to define schema objects that are present in a schema independently of other schema objects. For example, a UML class is a nodal construct, since schema objects such as $\mathsf{class}{:}\langle\langle\mathsf{aircraft\_type}\rangle\rangle$ and $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ in S^{uml} may exist without any other classes in the UML schema. A relational table is also a nodal construct, since schema objects such as $\mathsf{table}{:}\langle\langle\mathsf{aircraft\_type}\rangle\rangle$ and $\mathsf{table}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ may exist without any other tables in the relational schema.
  Typically, the instances of a UML class are identified using object identifies, so we might find that
  $Ext(\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle) = \{\langle 100\rangle, \langle 101\rangle, \ldots\}$
  $Ext(\mathsf{class}{:}\langle\langle\mathsf{aircraft\_type}\rangle\rangle) = \{\langle 200\rangle, \langle 201\rangle, \ldots\}$
  In predicate logic, these extents would cause the term $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle(X)$ to bind X to first 100, then 101, etc.
  Relational tables are typically identified using **natural keys** [10] (*i.e.* keys made up of attributes which have a meaning in the real-world, such a post codes, peoples names, tax numbers, *etc*), so we might find
  $Ext(\mathsf{table}{:}\langle\langle\mathsf{aircraft}\rangle\rangle) = \{\langle\mathsf{G\text{-}CWQS}\rangle, \langle\mathsf{G\text{-}FDWC}\rangle, \ldots\}$,
  i.e. the registration codes of the aircraft.
- **link-nodal** constructs are used to define schema objects that can only exist when connected to other schema objects, but contain data that is not present in the schema objects they are connected to. For example, in UML $\mathsf{attribute}{:}\langle\langle\mathsf{aircraft}, \mathsf{miles}\rangle\rangle$ is a link-nodal construct, since it can only exist when connected to $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$. A relational column such as $\mathsf{column}{:}\langle\langle\mathsf{aircraft}, \mathsf{miles}\rangle\rangle$ is also link nodal since it can only exist when connected to $\mathsf{table}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$. The definition of link-nodal constructs implies that the following rule about the extent of link-nodal schema objects is always true:
  $\textit{link-nodal}{:}\langle\langle E,A\rangle\rangle(X,Y) \rightarrow \textit{nodal}{:}\langle\langle E\rangle\rangle(X)$
  Hence, given the extent of $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ and the above rule, we might find that $Ext(\mathsf{attribute}{:}\langle\langle\mathsf{aircraft}, \mathsf{miles}\rangle\rangle) = \{\langle\mathsf{100{,}2945321}\rangle, \langle\mathsf{101{,}506834}\rangle, \ldots\}$, and give the extent of $\mathsf{table}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ and the above rule we might find that
  $Ext(\mathsf{column}{:}\langle\langle\mathsf{aircraft}, \mathsf{miles}\rangle\rangle) = \{\langle\mathsf{G\text{-}CWQS{,}2945321}\rangle, \langle\mathsf{G\text{-}FDWC{,}506834}\rangle, \ldots\}$
  Note that a peculiarity of the natural key based modelling languages is that the link-nodal schema object used to define the key will contain duplicates, so we might find $Ext(\mathsf{column}{:}\langle\langle\mathsf{aircraft}, \mathsf{reg}\rangle\rangle) = \{\langle\mathsf{G\text{-}CWQS{,}G\text{-}CWQS}\rangle, \langle\mathsf{G\text{-}FDWC{,}G\text{-}FDWC}\rangle, \ldots\}$.
- **link** constructs are used to define schema objects that can only exist when connected to two or more other schema objects, and contain data that is also present

in the schema objects they are connected to. For example, the UML schema object $\mathsf{association}{:}\langle\langle\mathsf{fixed\_on},\mathsf{aircraft},\mathsf{engine}\rangle\rangle$ is a link construct, since it has to be connected to $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ and $\mathsf{class}{:}\langle\langle\mathsf{engine}\rangle\rangle$. The definition of link-nodal constructs implies that the following rule about the extent of link-nodal schema objects is always true:

$link{:}\langle\langle R, E_1, E_2\rangle\rangle(X,Y) \rightarrow nodal{:}\langle\langle E_1\rangle\rangle(X) \wedge nodal{:}\langle\langle E_2\rangle\rangle(Y))$

Hence, given the extent of $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ and $\mathsf{class}{:}\langle\langle\mathsf{aircraft\_type}\rangle\rangle$ above, we might find

$Ext(\mathsf{composition}{:}\langle\langle\_,\mathsf{aircraft\_type},\mathsf{aircraft}\rangle\rangle) = \{\langle 200{,}100\rangle, \langle 200{,}101\rangle, \ldots\}$.

Note that there are no examples of link constructs found in the relational model.

A fourth type of construct is used to define **constraint** schema objects that have no associated extent, but place restrictions on the extents of the schema objects that appear within the constraint schema object. Figs. 1 and 2 illustrate the representation of a UML schema and a relational schema as a set of schemes. The UML schema includes a generalisation constraint schema object, and the relational schema includes primary_key and foreign_key schema objects, but null/notnull constraints have been omitted from the schema for brevity, since they are not used in this paper.

3 Temporal Constraints

We will identify in the following subsections three temporal constraints that have to some extent already implicitly been used in data modelling languages, but to date have not been explicitly identified as general modelling concepts in their own right that may be applied to any non-temporal data modelling language, though as we will see, sometimes have been made available for specific modelling constructs in specific modelling languages.

To accurately characterise the concepts, we will use discrete linear **temporal logic** [11] to define when certain properties hold. In the discrete linear model of time, we view the state of the information system passing through a (possibly infinite) series of states, where each state has one successor (next time) state, and one predecessor (previous time) state. In the terminology of temporal databases, we are modelling the **transaction time** [1] of the information system (but note that in this paper, we do *not* assume that we keep a transaction time history of the states of the information system).

The temporal logic we use in this paper is first order predicate logic with the addition of two binary operators, Until and Since, and hence is often referred to as **US-Logic**. The statement $A\ \mathsf{Until}\ B$ means that A holds at every time up to and including the time when B holds. From this operator, a number of derived unary and binary operators can be defined (where $\top$ represents truth, and holds in every state):

$\bigcirc A \equiv A\ \mathsf{Until}\ \top$
$A\ \mathsf{While}\ B \equiv A\ \mathsf{Until}\ \neg\bigcirc B$
$\Diamond A \equiv \top\ \mathsf{Until}\ A$
$\Box A \equiv A\ \mathsf{While}\ \top$

which we illustrate with the following examples:

1. $\bigcirc\langle\langle \mathsf{A}\rangle\rangle(X)$ means that in the next time there is an instance X of schema object $\langle\langle \mathsf{A}\rangle\rangle$. Hence the formula $\langle\langle \mathsf{A}\rangle\rangle(X) \rightarrow \bigcirc\langle\langle \mathsf{A}\rangle\rangle(X)$ means that if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at any time, then it will be an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at the next time, and $\langle\langle \mathsf{A}\rangle\rangle(X) \rightarrow \neg\bigcirc\langle\langle \mathsf{A}\rangle\rangle(X)$ means that if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at any time, then it will not be an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at the next time.
2. $\langle\langle \mathsf{A}\rangle\rangle(X)\,\mathsf{While}\,\langle\langle \mathsf{B}\rangle\rangle(X)$ holds if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ for the entire period that X continues to be an instance of $\langle\langle \mathsf{B}\rangle\rangle$.
3. $\Diamond\langle\langle \mathsf{A}\rangle\rangle(X)$ means that in some future time there is an instance X of schema object $\langle\langle \mathsf{A}\rangle\rangle$. Hence the formula $\langle\langle \mathsf{A}\rangle\rangle(X) \rightarrow \Diamond\langle\langle \mathsf{A}\rangle\rangle(X)$ means that if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at any time, then X will be an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at some future time, and $\langle\langle \mathsf{A}\rangle\rangle(X) \rightarrow \neg\Diamond\langle\langle \mathsf{A}\rangle\rangle(X)$ means that if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at any time, then it will never be an instance of $\langle\langle \mathsf{A}\rangle\rangle$ again.
4. $\Box\langle\langle \mathsf{A}\rangle\rangle(X)$ means that in all future times there is an instance X of schema object $\langle\langle \mathsf{A}\rangle\rangle$. Hence the formula $\langle\langle \mathsf{A}\rangle\rangle(X) \rightarrow \Box\langle\langle \mathsf{A}\rangle\rangle(X)$ means that if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at any time, then it will so for ever more, and $\langle\langle \mathsf{A}\rangle\rangle(X) \rightarrow \neg\Box\langle\langle \mathsf{A}\rangle\rangle(X)$ means that if X is an instance of $\langle\langle \mathsf{A}\rangle\rangle$ at any time, there will be some time in the future when it is not an instance.

3.1 Monogamy and Lifetime Monogamy

In general, the concept of **monogamy** involves something being related to just one other thing at any one time. In data modelling, this concept is captured using optional or mandatory cardinality constraints — *i.e.* cardinality constraints with an upper bound of one. For example, association:$\langle\langle \mathsf{fixedon}, \mathsf{aircraft}, \mathsf{engine}\rangle\rangle$ in Fig. 1 makes class:$\langle\langle \mathsf{engine}\rangle\rangle$ have a monogamous relationship with class:$\langle\langle \mathsf{aircraft}\rangle\rangle$, meaning each engine can only be fixed on one aircraft at a time. In our representation of data modelling, we can say that an instance of a nodal schema object appearing in some link-nodal or link schema object is monogamous for that schema object if one of the following rules hold, which in essence state that there cannot be two instances of the link-nodal or link schema object for the same monogamous schema object instance.

$$\text{monogamous}(nodal{:}\langle\langle E_1\rangle\rangle, link{:}\langle\langle R,E_1,E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$link{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \rightarrow \neg\exists Z.link{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Z) \wedge Y \neq Z$$
$$\text{monogamous}(nodal{:}\langle\langle E_2\rangle\rangle, link{:}\langle\langle R,E_1,E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$link{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \rightarrow \neg\exists Z.link{:}\langle\langle R,E_1,E_2\rangle\rangle(Z,Y) \wedge Y \neq Z$$
$$\text{monogamous}(nodal{:}\langle\langle E\rangle\rangle, link\text{-}nodal{:}\langle\langle E,A\rangle\rangle) \stackrel{\text{def}}{=}$$
$$link\text{-}nodal{:}\langle\langle E,A\rangle\rangle(X,Y) \rightarrow \neg\exists Z.link\text{-}nodal{:}\langle\langle E,A\rangle\rangle(X,Z) \wedge Y \neq Z$$

Hence for Fig. 1, we can state:

monogamous(class:$\langle\langle \mathsf{engine}\rangle\rangle$,association:$\langle\langle \mathsf{fixedon}, \mathsf{aircraft}, \mathsf{engine}\rangle\rangle$)
monogamous(class:$\langle\langle \mathsf{tyre}\rangle\rangle$,aggregation:$\langle\langle \_, \mathsf{aircraft}, \mathsf{tyre}\rangle\rangle$)
monogamous(class:$\langle\langle \mathsf{aircraft}\rangle\rangle$,composition:$\langle\langle \_, \mathsf{aircraft\_type}, \mathsf{aircraft}\rangle\rangle$)

Note that this definition of monogamy does not prevent **serial monogamy**, *i.e.* an instance of the nodal class being monogamous at any one time, but changing its relationships over time. For UML **associations**, this definition is intuitively correct. For example, it would allow a class:$\langle\langle \mathsf{engine}\rangle\rangle$ instance to be moved from one class:$\langle\langle \mathsf{aircraft}\rangle\rangle$ to another. However, the definition of UML **aggregation** and **composition** are defined to usually imply that members of the aggregation are not allowed to change from one group to another [2]. Here we suggest that this 'usually' be strengthened to a **lifetime monogamous** temporal constraint, that prevents serial monogamy. Once a certain value Y has been associated with a schema object in its connection with a particular instance X of some other schema object, then during one period of existence of X there may not be some different value Z used instead of Y. Specifically:

$$\text{lifetime\_monogamous}(\mathit{nodal}{:}\langle\langle E_1\rangle\rangle, \mathit{link}{:}\langle\langle R, E_1, E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link}{:}\langle\langle R, E_1, E_2\rangle\rangle(X,Y) \rightarrow$$
$$(\neg\exists Z.\mathit{link}{:}\langle\langle R, E_1, E_2\rangle\rangle(X,Z) \wedge Y \neq Z)\ \mathsf{While}\ \mathit{nodal}{:}\langle\langle E_1\rangle\rangle(X)$$

$$\text{lifetime\_monogamous}(\mathit{nodal}{:}\langle\langle E_2\rangle\rangle, \mathit{link}{:}\langle\langle R, E_1, E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link}{:}\langle\langle R, E_1, E_2\rangle\rangle(X,Y) \rightarrow$$
$$(\neg\exists Z.\mathit{link}{:}\langle\langle R, E_1, E_2\rangle\rangle(Z,Y) \wedge X \neq Z)\ \mathsf{While}\ \mathit{nodal}{:}\langle\langle E_2\rangle\rangle(Y)$$

$$\text{lifetime\_monogamous}(\mathit{link\text{-}nodal}{:}\langle\langle E, A\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link}{:}\langle\langle E, A\rangle\rangle(X,Y) \rightarrow$$
$$(\neg\exists Z.\mathit{link}{:}\langle\langle E, A\rangle\rangle(X,Z) \wedge Y \neq Z)\ \mathsf{While}\ \mathit{nodal}{:}\langle\langle E\rangle\rangle(X)$$

i.e. lifetime monogamy for a nodal schema object in a link or link-nodal schema object implies monogamy for the duration of a single lifespan of the nodal schema object. We will interpret the semantics of UML modelling to imply for Fig. 1:

$$\text{lifetime\_monogamous}(\mathsf{class}{:}\langle\langle\mathsf{tyre}\rangle\rangle, \mathsf{aggregation}{:}\langle\langle\_, \mathsf{tyre}, \mathsf{aircraft}\rangle\rangle)$$
$$\text{lifetime\_monogamous}(\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle, \mathsf{composition}{:}\langle\langle\_, \mathsf{aircraft\_type}, \mathsf{aircraft}\rangle\rangle)$$

The first line above means that an instance of $\mathsf{class}{:}\langle\langle\mathsf{tyre}\rangle\rangle$ can only ever be associated with one $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ during one period of existence of a tyre, *i.e.* a tyre can only be used on one aircraft, but there is no constraint of which of the tyre or the aircraft existed first, and the tyre can be taken off the aircraft without destroying either the tyre or the aircraft. The second line means that an instance of $\mathsf{class}{:}\langle\langle\mathsf{aircraft}\rangle\rangle$ can only ever be associated with one $\mathsf{class}{:}\langle\langle\mathsf{aircraft\_type}\rangle\rangle$.

Note that the UML concept of **readOnly** implies that a value must be set during object initialisation, and hence implies a mandatory cardinality constraint in combination with a lifetime monogamous temporal constraint. Note that the relational model has no constructs that imply the lifetime monogamous constraint on link schema objects or link-nodal schema objects, and UML does not provide the constraint in conjunction with optional cardinality constraints. However, this does not mean the it would not be useful to introduce a specific lifetime monogamous temporal constraint to these models. For example, if $\mathsf{attribute}{:}\langle\langle\mathsf{aircraft}, \mathsf{reg}\rangle\rangle$ were lifetime monogamous, one could build a plane without a registration code, register it, and later cancel the registration code before scraping the aircraft, but ensure that one never assigns a different registration code to the aircraft. This would also be readily implemented in SQL using triggers to control the updating of a column, such that a state column was set to true when a data column was set to null, prohibiting any further setting of data column value.

3.2 One-off

The **oneoff** temporal constraint means that once a schema object instance is deleted, the same instance cannot exist again. For nodal schema objects, this constraint is easily characterised as

$$\text{oneoff}(nodal{:}\langle\langle E\rangle\rangle) \stackrel{\text{def}}{=}$$
$$nodal{:}\langle\langle E\rangle\rangle(X) \wedge \neg\bigcirc nodal{:}\langle\langle E\rangle\rangle(X) \rightarrow \neg\Diamond nodal{:}\langle\langle E\rangle\rangle(X)$$

stating that if X is an instance of $nodal{:}\langle\langle \mathrm{E}\rangle\rangle$ at any time, and at the next time it is not an instance of $nodal{:}\langle\langle \mathrm{E}\rangle\rangle$, then there will be no future time when X is an instance of $nodal{:}\langle\langle \mathrm{E}\rangle\rangle$. The oneoff temporal constraint is often associated with nodal schema objects in object-oriented models, where once a class instance has been deleted, the same class instance cannot be restored. For example, once instance $\langle 100\rangle$ has been deleted of class:⟨⟨aircraft⟩⟩, there would not in the future be an instance $\langle 100\rangle$ of class:⟨⟨aircraft⟩⟩. By contrast, models such as the relational models when based **natural keys** do not support the oneoff temporal constraint. This is because a relational database has no mechanism to stop a natural key being reinserted into the database after it has previously been deleted.

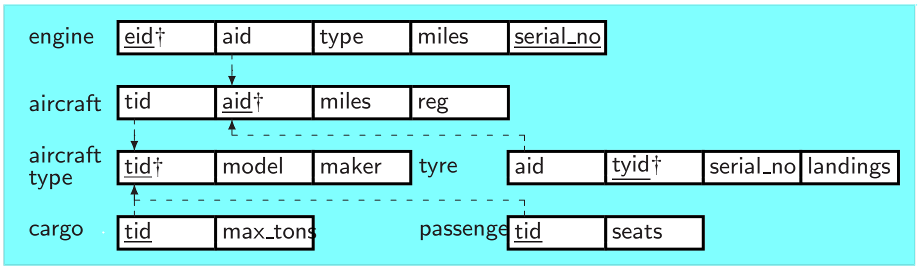


Fig. 3. S_2^{rel}: A variant of S_1^{rel} using auto-increment keys, marked with a † in the diagram. Note that cargo and passenger do not have auto-increment keys, since they inherit the value of the key from aircraft_type)

If the relational system uses **auto-increment** keys, then the behaviour would be more similar to that of the object oriented system. Fig. 3 presents a version of Fig. 2 where auto-increment keys have been used (and Fig. 4 the SQL definitions of some of the tables), and then we can state

oneoff(table:⟨⟨aircraft⟩⟩)
oneoff(table:⟨⟨aircraft_type⟩⟩)
oneoff(table:⟨⟨engine⟩⟩)
oneoff(table:⟨⟨tyre⟩⟩)

since once a key value has been generated for an auto-increment key, the value will not be generated again in the future. A problem remains with the tables implementing the subclasses passenger and cargo, which are unable to use auto-increment keys. We can solve this problem with our implementation of the final temporal constraint presented in the next subsection.

Definition of oneoff for link-nodal and link schema object takes a similar form to that for nodal schema objects:

```
CREATE TABLE aircraft_type
(      mid INT PRIMARY KEY,
       maker VARCHAR(20)
)

CREATE TABLE passenger
(      mid INT PRIMARY KEY REFERENCES aircraft_type ON DELETE CASCADE,
       seats INT
)

CREATE TABLE cargo
(      mid INT PRIMARY KEY REFERENCES aircraft_type ON DELETE CASCADE,
       max_tons INT
)

CREATE FUNCTION delete_aircraft_type() RETURNS TRIGGER
AS 'BEGIN
       DELETE FROM aircraft_type WHERE aircraft_type.tid=OLD.tid;
       RETURN NULL;
       END' LANGUAGE plpgsql;

CREATE TRIGGER passenger_subclass_aircraft_type AFTER DELETE ON passenger
FOR EACH ROW EXECUTE PROCEDURE delete_aircraft_type();

CREATE TRIGGER cargo_subclass_aircraft_type AFTER DELETE ON cargo
FOR EACH ROW EXECUTE PROCEDURE delete_aircraft_type();
```

Fig. 4. Definition using the Postgres RDBMS SQL language of triggers being used to implement the final temporal constraints

$$\text{oneoff}(\textit{link-nodal}{:}\langle\langle E,A\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\textit{link-nodal}{:}\langle\langle E,A\rangle\rangle(X,Y) \wedge \neg\bigcirc \textit{link-nodal}{:}\langle\langle E,A\rangle\rangle(X,Y) \rightarrow$$
$$\neg\textit{link-nodal}{:}\langle\langle E,A\rangle\rangle(X,Y)\,\mathsf{While}\,\textit{nodal}{:}\langle\langle E\rangle\rangle(X)$$

$$\text{oneoff}(\textit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\textit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \wedge \neg\bigcirc \textit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \rightarrow$$
$$\neg\textit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y)\,\mathsf{While}\,\textit{nodal}{:}\langle\langle E_1\rangle\rangle(X) \wedge \textit{nodal}{:}\langle\langle E_2\rangle\rangle(Y)$$

Neither UML nor the relational models have constructs that imply oneoff to link-nodal or link schema objects, and the semantics of such a constraint would only be of use in relatively few circumstances. For example, if we added to the UML schema in Fig. 1 oneoff($\mathsf{attribute}{:}\langle\langle\mathsf{aircraft},\mathsf{reg}\rangle\rangle$), then an aircraft could change its registration number, but not revert to a previously used registration number. It should be noted that the general implementation of oneoff is costly in storage terms, since it requires a transaction time history be kept a schema object declared as one-off so that a check can be made each time a new instance is created that the instance had not been present at some time in the past. The specific case of oneoff being applied to object identifiers and auto-increment keys is not costly in storage terms since only a single variable incrementing new values need be kept in order to ensure unique values over time.

3.3 Final

The **final** temporal constraint means that once a instance of a schema object has been created, then that instance will remain until one of the instances of the schema objects it is dependent upon is deleted. Specifically, for nodal schema objects, we state:

$$\text{final}(nodal{:}\langle\langle E_1\rangle\rangle, nodal{:}\langle\langle E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$nodal{:}\langle\langle E_2\rangle\rangle(X) \rightarrow nodal{:}\langle\langle E_2\rangle\rangle(X)\,\mathsf{While}\,nodal{:}\langle\langle E_1\rangle\rangle(X)$$

meaning that once a instance exists in $nodal{:}\langle\langle E_2\rangle\rangle$, it must continue to exist whilst the same values exists in $nodal{:}\langle\langle E_1\rangle\rangle$. UML **generalisations** imply the final temporal constraint between the child and parent nodes. For the schema in Fig. 1 we can state:

final(class:⟨⟨aircraft_type⟩⟩,class:⟨⟨passenger⟩⟩)
final(class:⟨⟨passenger⟩⟩,class:⟨⟨aircraft_type⟩⟩)
final(class:⟨⟨aircraft_type⟩⟩,class:⟨⟨cargo⟩⟩)
final(class:⟨⟨cargo⟩⟩,class:⟨⟨aircraft_type⟩⟩)

Hence, when an instance of class:⟨⟨passenger⟩⟩ is deleted, then so must the instance of class:⟨⟨aircraft_type⟩⟩, and *vice versa*.

There is no modelling construct in the relational model that directly implies final on its schema objects, but there is some limited support for implementing the final constraint. Firstly, if we added the SQL constraint ON DELETE CASCADE to primary keys of table:⟨⟨passenger⟩⟩ and table:⟨⟨cargo⟩⟩, as illustrated by the table definitions in Fig. 4, then we would be able to state:

final(table:⟨⟨aircraft_type⟩⟩,table:⟨⟨passenger⟩⟩)
final(table:⟨⟨aircraft_type⟩⟩,table:⟨⟨cargo⟩⟩)

since deleting a row from table:⟨⟨aircraft_type⟩⟩ will cause the cascading of a delete on table:⟨⟨passenger⟩⟩ or table:⟨⟨cargo⟩⟩. Secondly, if we added the SQL trigger for each of the passenger and cargo table, as illustrated by the trigger definitions in Fig. 4, which executes a function that deletes the same identifier from the parent aircraft_type table, then a deletion of either table:⟨⟨passenger⟩⟩ or table:⟨⟨cargo⟩⟩ would trigger a deletion of table:⟨⟨aircraft_type⟩⟩. The presence of such a trigger then allows us to state:

final(table:⟨⟨passenger⟩⟩,table:⟨⟨aircraft_type⟩⟩)
final(table:⟨⟨cargo⟩⟩,table:⟨⟨aircraft_type⟩⟩)

In defining the final constraint for link-nodal constructs, there are two cases to consider. Applying the first rule below to a UML attribute or a relational column would mean that once a value was assigned to the attribute/column it could not be changed. For example final(attribute:⟨⟨aircraft, reg⟩⟩) would mean that a registration number of a aircraft could not be changed. Interestingly, this modelling concept is absent from the UML[1] and relational languages, but is present in some object oriented programming

[1] It is interesting to note that the semantics of final would appear to match the semantics of the addOnly property of link nodal and link constructs available in some versions of UML prior to UML 2.0. Also removed in UML v2.0 was the concept of **createOnly**, which stated that values could be added once but no more to a property. Both addOnly and createOnly forbid changes.

languages which UML targets (for example the final keyword in Java and readonly keyword in C#).

$$\text{final}(\mathit{link\text{-}nodal}{:}\langle\langle E,A\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link\text{-}nodal}{:}\langle\langle E,A\rangle\rangle(X,Y) \rightarrow \mathit{link\text{-}nodal}{:}\langle\langle E,A\rangle\rangle(X,Y)\ \mathsf{While}\ \mathit{nodal}{:}\langle\langle E\rangle\rangle(X)$$

$$\text{final}(\mathit{link\text{-}nodal}{:}\langle\langle E_1,A_1\rangle\rangle, \mathit{link\text{-}nodal}{:}\langle\langle E_2,A_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link\text{-}nodal}{:}\langle\langle E_1,A_1\rangle\rangle(X,Y) \wedge \mathit{link\text{-}nodal}{:}\langle\langle E_2,A_2\rangle\rangle(Z,Y) \rightarrow$$
$$\mathit{link\text{-}nodal}{:}\langle\langle E_2,A_2\rangle\rangle(Z,Y)\ \mathsf{While}\ \mathit{link\text{-}nodal}{:}\langle\langle E_1,A_1\rangle\rangle(X,Y)$$

The second rule causes a value appearing in one link-nodal that is also appearing in a second link-nodal to cause the same second value to continue to exist whilst the first continues to exist. For example, if we stated on S_2^{rel}

final(column:⟨⟨aircraft, tid⟩⟩,column:⟨⟨aircraft_type, tid⟩⟩)

then we would have the same semantics present in the relational model as we gave to the UML composition construct above, and aircrafts would not be able to change aircraft types.

For link constructs, there again two types of final constraint. Applied to a UML association, the first rule below says that once an instance of the association has been created it remains in existence whilst both of the classes it associates exist. The second two rules strengthen the rule to say that the instance of the association will continue in existence until just one of the classes it associates is deleted.

$$\text{final}(\mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \rightarrow$$
$$\mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y)\ \mathsf{While}\ (\mathit{nodal}{:}\langle\langle E_1\rangle\rangle(X) \wedge \mathit{nodal}{:}\langle\langle E_2\rangle\rangle(Y))$$

$$\text{final}(\mathit{nodal}{:}\langle\langle E_1\rangle\rangle, \mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \rightarrow \mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y)\ \mathsf{While}\ \mathit{nodal}{:}\langle\langle E_1\rangle\rangle(X)$$

$$\text{final}(\mathit{nodal}{:}\langle\langle E_2\rangle\rangle, \mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$\mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y) \rightarrow \mathit{link}{:}\langle\langle R,E_1,E_2\rangle\rangle(X,Y)\ \mathsf{While}\ \mathit{nodal}{:}\langle\langle E_2\rangle\rangle(Y)$$

In UML, the **composition** construct often implies a coincidence in lifetimes of the classes in the composition. We propose to restrict this definition to stating that the member class of a composition is final in the composition. In Fig. 1, this means we can state:

final(class:⟨⟨aircraft⟩⟩,composition:⟨⟨_, aircraft_type, aircraft⟩⟩)

meaning that once an aircraft has been assigned to an aircraft type, it cannot be changed to another aircraft type. We would not want to associate the final temporal constraint with UML **aggregations**. For example, if final(aggregation:⟨⟨_, aircraft, tyre⟩⟩) was declared for the UML schema, then once ⟨100,107⟩ has been added as an instance, it would remain until the tyre ⟨107⟩ was deleted from class:⟨⟨tyre⟩⟩, preventing us from removing instances from aggregations.

4 Related Work

There has been a considerable amount of work conducted into the modelling of temporal constraints in temporal data models (for example [12,13,14,15,16]). By contrast, this paper considers temporal constraints that may be used, or are already implied, in existing non-temporal data models.

In the field of data modelling, the most comprehensive previous treatment can be found in [17,18], which deal with the temporal behaviours of nodal and link-nodal constructs, but not of link constructs. Also, [17,18] do not explicitly relate their definitions to specific modelling languages, though the relationship with ER and UML modelling is clear. In [17], for nodal constructs, the concept of **permanent** constraint is defined where an object, once it exists, must stay in existence whilst the information system remains in operation.

$$\text{permanent}(nodal{:}\langle\langle E\rangle\rangle) \stackrel{\text{def}}{=}$$
$$link{:}\langle\langle E\rangle\rangle(X) \rightarrow \Box nodal{:}\langle\langle E\rangle\rangle(X)$$

This is similar to the final constraint, with the difference that the final constraint is always defined relative to some other object. We argue that this is more intuitive, since is corresponds to the real world concept of something entering its final state, yet not necessarily continuing to exist forever.

In [17], there is the concept of **frequency** being **single** or **multiple**. Applied to entities the concept of single corresponds to exactly to the definition of one-off in this paper. However, applied to attributes, it differs from one-off in allowing an attribute to change value and then return to value during its single existence (where the definition of one-off states that a particular value may be used only once). The authors also introduce concept of **durability**, which may be **durable** or **instantaneous**, where instantaneous means that an object only exists for one chronon in the temporal model. This is a common distinction in temporal data models, since it allows instantaneous schema objects to be stored with one time value per instance (the time of the instance occurred at), whilst durable schema objects require a pair of time values representing the interval the instance exists for (or set of such pairs if there is set of intervals).

In [18], there was a discussion of how generalisations could be classified into **static** if sub-class memberships could not evolve over time, or **dynamic** if they could. In [19], the concept of **temporal behaviour** of UML associations is defined, and characterised as **static** or **dynamic** depending on whether the values on the association for a particular class may be changed. There is also consideration to the definition of delete propagations across associations.

Work on the temporal constraints on nodal objects has also been conducted in the field of ontologies. In [20,21] a classification of unary predicates (equivalent to our nodal constructs) into **rigid**, **anti-rigid**, and **non-rigid** was introduced. The rigid constraint takes the same definition as the permanent constraint in [17]:

$$\text{rigid}(nodal{:}\langle\langle E_1\rangle\rangle) \stackrel{\text{def}}{=}$$
$$nodal{:}\langle\langle E_1\rangle\rangle(X) \rightarrow \Box(nodal{:}\langle\langle E_1\rangle\rangle(X))$$

Recently, it has been proposed that ORM be extended to include this distinction [22]. The definition of rigid shares the flaw we discussed in relationship to the permanent constraint from [17]. This flaw was recognised in [23], which proposed **existential rigidity**, where a value appearing in $nodal{:}\langle\langle E_1\rangle\rangle$ forces the value to also appear in some related $nodal{:}\langle\langle E_2\rangle\rangle$:

$$\text{existential\_rigid}(nodal{:}\langle\langle E_1\rangle\rangle, nodal{:}\langle\langle E_2\rangle\rangle) \stackrel{\text{def}}{=}$$
$$nodal{:}\langle\langle E_1\rangle\rangle(X) \rightarrow \Box(nodal{:}\langle\langle E_1\rangle\rangle(X) \rightarrow nodal{:}\langle\langle E_2\rangle\rangle(X))$$

If we applied this to our relational model as:

existential_rigid(table:⟨⟨aircraft_type⟩⟩,table:⟨⟨passenger⟩⟩

existential_rigid(table:⟨⟨aircraft_type⟩⟩,table:⟨⟨cargo⟩⟩)

then we would have to implement a temporal history of data instance from tables table:⟨⟨passenger⟩⟩ and table:⟨⟨cargo⟩⟩ (which are the relational tables implementing the subclasses of table:⟨⟨aircraft_type⟩⟩), since if a value where to be deleted and reinserted into table:⟨⟨aircraft_type⟩⟩ then we would need to ensure that if also appeared in the correct subclass table.

From the above discussion, it can be seen that the definitions in this paper are the first to be made across all types of modelling construct, and also the first to be defined in a manner that is implementable without the necessity of maintaining a full transaction time history of data.

5 Summary and Conclusions

In this paper, we defined three temporal constraints called lifetime monogamy, oneoff, and final, that may be used to model the changes that are permitted to the extents associated with schema objects in static non-temporal data modelling languages. Loosely speaking, (i) lifetime monogamy models the concept that mandatory or optional relationships are restricted further to disallow serial monogamy, (ii) oneoff models the concept that things cannot be reincarnated, and (iii) final models the concept that once a value has been assigned, it cannot be changed.

We have given precise definitions of these three constraints in linear temporal logic, and have discussed how some of these constraints are already fully or partially implied by constructs found in the UML and relational languages. Our definitions are made in terms of very general modelling concepts of nodal, link-nodal and link modelling constructs, and this approach has previously been shown to be capable of representing a wide variety of modelling languages [8,4].

The precise definitions of temporal constraints serve to give two advantages. First, the modelling constructs of UML and the relational model are better understood, leading to a more accurate modelling of the real world when using these languages. Secondly, the definitions serve to expose the differences that exist between modelling languages, and allow action to be taken to overcome these differences. We illustrated this second advantage by describing how SQL CASCADE and TRIGGER constructs can be used to implement the temporal constraints, and hence make a relational based system be capable of holding a schema that corresponds more exactly with a UML schema than is the case in current approaches to UML to relational mapping.

To shorten the presentation, we have restricted the class of modelling languages discussed to those with binary link schema objects and with link and link-nodal schema objects that only connect with nodal schema objects. However the extension of the work to remove those restrictions is straightforward.

References

1. Jensen *et al*, C.: A consensus glossary of temporal database concepts. SIGMOD Record **23**(1) (1994) 52–64

2. Group, O.M.: Unified Modeling Language: Superstructure 2.1.1. Technical report, OMG (2007)
3. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: More Dynamic Object Re-classification: FickleII. ACM Transactions On Programming Languages and Systems **24**(2) (2002) 153–191
4. Boyd, M., McBrien, P.: Comparing and transforming between data models via an intermediate hypergraph data model. Journal on Data Semantics **IV** (2005) 69–109
5. McBrien, P., Poulovassilis, A.: A semantic approach to integrating XML and structured data sources. In: Proc. CAiSE'01. Volume 2068 of LNCS., Springer (2001) 330–345
6. Cabibbo, L., Carosi, A.: Managing inheritance hierarchies in object/relational mapping tools. In: Proc. 17th CAiSE. Volume 3520 of LNCS., Springer (2005) 135–150
7. Boyd, M., Kittivoravitkul, S., Lanzanitis, C., McBrien, P., Rizopoulos, N.: AutoMed: A BAV data integration system for heterogeneous data sources. In: Proc. CAiSE'04. Volume 3084 of LNCS., Springer (2004) 82–97
8. McBrien, P., Poulovassilis, A.: A uniform approach to inter-model transformations. In: Proc. CAiSE'99. Volume 1626 of LNCS., Springer (1999) 333–348
9. Date, C., Darwen, H., McGoveran, D.: Relational Database: Selected Writings 1994–1997. Addison-Wesley (1998)
10. Date, C.: Object identifiers vs. relational keys. [9] chapter 12 457–476
11. Fisher, M., Gabbay, D., Vila, L., eds.: Handbook of Temporal Reasoning in Artificial Intelligence. Elsevier (2005)
12. Artale, A., Parent, C., Spaccapietra, S.: Modeling the evolution of objects in temporal information systems. In: Proc. FoIKS. (2006) 22–42
13. Finger, M., McBrien, P.: Temporal conceptual-level databases. In: Temporal Logics: Mathematical Foundations and Computational Aspects (Vol 2). OUP (2000) 409–435
14. Gregersen, H., Jensen, C.: Temporal entity-relationship models: a survey. IEEE Trans. KDE **11**(3) (1999) 464–497
15. Spaccapietra, S., Parent, C., Zimanyi, E.: Modeling time from a conceptual perspective. In: Proc. CIKM. (1998) 432–440
16. McBrien, P., Seltveit, A., Wangler, B.: An entity-relationship model extended to describe historical information. In: Proceedings of CISMOD '92, Bangalore, India (1992) 244–260
17. Costal, D., Olivé, A., Sancho, M.R.: Temporal features of class populations and attributes in conceptual models. In: Proc. ER. (1997) 57–70
18. Olivé, A., Costal, D., Sancho, M.R.: Entity evolition in IsA hierarchies. In: Proc. ER. (1999) 62–80
19. Albert, M., Pelechano, V., Fons, J., Ruiz, M., Pastor, O.: Implementing UML association, aggregation, and composition. A particular interpretation based on a multidimensional. In: Proc. CAiSE. (2003) 143–158
20. Guarino, N., Carrara, M., Giaretta, P.: An ontology of meta-level categories. In: Proc. 4th KR. (1994) 270–280
21. Guarino, N., Welty, C.: Ontological analysis of taxonomic relationships. In: Proc. ER. (2000) 210–224
22. Halpin, T.: Subtyping revisited. In Proper, H., Halpin, T., Krogstie, J., eds.: Proc. EMMSAD 07. (2007) 128–138
23. Anderson, W., Menzel, C.: Modal rigidity in the ontoclean methodology. In: Proc. FOIS. (2004) 119–127