

Representing Temporal Information in UML

Jordi Cabot, Antoni Olivé and Ernest Teniente

Universitat Politècnica de Catalunya
Dept. Llenguatges i Sistemes Informàtics
Jordi Girona 1-3, 08034 Barcelona

[jcabot|olive|teniente]@lsi.upc.es

Abstract. The UML is a non-temporal conceptual modeling language. Conceptual schemas in the UML assume that the information base contains the current instances of entity and relationship types. For many information systems, the above assumption is acceptable. However, there are some information systems for which that assumption is a severe limitation. This happens when the functions of the information system require the knowledge of past states of the information base.

In this paper we extend the UML to define a set of temporal features of entity and relationship types, and to provide notational devices to refer to any past state of the information base. Using this extension, a designer may use the UML/OCL as if it were a temporal conceptual modeling language. We also present a method for the transformation of a conceptual schema in this extended language into a conventional one. The method can be automated, and we describe an implementation. The result of our transformation method is a conceptual schema that can be processed by ordinary CASE tools.

1. Introduction

The UML is a non-temporal conceptual modeling language. Conceptual schemas in the UML assume that the information base contains the current instances of the entity types and of the relationship types. Integrity constraints (invariants) defined in the conceptual schema can refer only to the current state of the information base. Operation pre/postconditions can refer only to the state before/after the operation execution. Moreover, the OCL does not provide constructs to refer (navigate) to past states of the information base.

For many information systems, the above assumption is acceptable. However, there are some information systems for which that assumption is a severe limitation. This happens when the functions of the information system require the knowledge of past states of the information base.

For example, assume an information system that must record the complete employment history of employees in some company. Among the requirements of the system there may be:

- The system must answer queries about current and past assignments of employees to departments, including the starting and ending times of each assignment.
- The system must enforce the (classical) integrity constraint: "salaries of employees must never decrease". The strict control of this constraint may not be easy if employees may leave the company and join it again later.

In temporal languages, the formal specification of the above queries and constraint would be straightforward. In non-temporal languages that specification is impossible if the information base contains only the current state. In these latter languages, the only solution is to define a conceptual schema such that a state of the information base includes not only the current state, but also all past ones. Such conceptual schemas and information bases are called historical.

The main contribution of this paper is twofold. First, we extend the UML (using the standard extension mechanisms) to define a set of temporal features of entity and relationship types, and to provide notational devices to refer to any past state of the information base. Using this extension, a designer may use the UML/OCL as if it were a temporal conceptual modeling language. We call temporally-extended a conceptual schema that uses these extensions and notational devices. Second, we present a method for the transformation of a temporally-extended UML conceptual schema into a conventional UML historical one. The method can be automated, and we describe an implementation. The result of our transformation method is a conceptual schema that can be processed by ordinary CASE tools, such as code generators.

Previous work related to the representation of temporal information in the UML is restricted to [4] which describes how to deal with this information during information system design; [16] which proposes some stereotypes which are not appropriate to incorporate temporal semantics in the UML since they do not allow to store the history of the entities in the information base; and [15] which provides a non-standard UML extension to represent temporal information. Then, to our knowledge, ours is the first standard extension of the UML to define temporally-extended conceptual schemas in this language.

Several extensions to the Entity-Relationship (ER) model have also been proposed to allow it to deal with time (see as a survey [6]). The approach of translating an extended high-level temporal model into a non-temporal one was already taken by [2, 9] in this context. However, apart from the logical differences among the UML and the ER, their temporal features are less powerful than the ones we consider in this paper and they do not provide notational means to refer to past states.

The paper is structured as follows. Next section introduces basic concepts. Section 3 describes temporal features of entity and relationship types. Section 4 proposes a UML stereotype to specify these features and describes several operations to define temporal OCL expressions. Section 5 defines the transformation of a temporally-extended UML schema into conventional non-temporal UML. Section 6 presents an implementation of this work. Finally, we give our conclusions and point out future work in Section 7.

2. Basic concepts

We assume that entities and relationships are instances of their types at particular time points, which are expressed in a common base time unit such as second or day. The *life span* Ls of an information system is the temporal interval $Ls = (t_i, t_f)$ during which the information system exists.

We represent by $E(e,t)$ the fact entity e is instance of entity type E at time t . The population of E at t is defined as the set of entities that are instances of E at t . Obviously, given a fact $E(A,T_i)$, T_i must be included in the life span of the system. This condition, also called *temporal integrity constraint*, must be satisfied for each entity type E and may be formally represented as: $\forall e,t E(e,t) \rightarrow \text{BelongsTo}(t, Ls)$.

A relationship type has a name and a set of n participants, with $n \geq 2$. A *participant* is an entity type that plays a certain role in the relationship type. $R(p_1:E_1, \dots, p_n:E_n)$ denotes a relationship type named R with entity type participants E_1, \dots, E_n playing the roles p_1, \dots, p_n , respectively. The order of the participants is irrelevant. Two different participants may be of the same entity type. When the role name is omitted, it is assumed to be the same as the corresponding entity type.

If t is a time point of the life span of the system, we represent by $R(e_1, \dots, e_n, t)$ the fact that entities e_1, \dots, e_n participate in an instance of R at time t . There is an implicit integrity constraint that requires e_i , for $i=1, \dots, n$, be instance of E_i at time t or before.

A *classification interval* of an entity e in an entity type E is the maximum set of consecutive instants of the life period of e in E . An entity may have several classification intervals in an entity type. The set $Ci(e,E)$ denotes all classification intervals of e in E . In a similar way, a classification interval of (e_1, \dots, e_n) in R is the maximum set of consecutive instants during which (e_1, \dots, e_n) is an instance of R . We denote the set of classification intervals of (e_1, \dots, e_n) in R as $Ci((e_1, \dots, e_n), R)$.

3. Temporal features of entity and relationship types

Some temporal features of entity and relationship types are important in conceptual modeling at least for three different reasons: to specify the meaning of an entity or a relationship type, to define integrity constraints that the information base must satisfy and to perform the design of the information system. We deal in this section with *durability* and *frequency*, which were already introduced in [12].

3.1 Durability

Durability of an entity or a relationship type refers to the persistence of the classification of its instances in the type. In this sense, an instance may be classified in a certain type just at particular time points or during certain time intervals. In the latter case, we may establish also two additional properties of the classification intervals of the instance in the type. Then, we distinguish four different values of durability: instantaneous, durable, permanent and constant.

An *entity type (a relationship type) is instantaneous* when its objects (relations) are instances of the type at given instants, without persisting from one instant to the next.

An example of instantaneous entity type could be *SneezingPerson* if we assume, as it seems natural, that sneezing is an instantaneous action. Another example could be a sell. A sell is an object (act, event) that can be classified as an instance of an entity type *Sell* at the instant in which it is produced, and not at any other instant. An example of instantaneous relationship type is *HasRegistered(Student, Subject)* meaning that a student has made a registration in a subject at a given instant.

An *entity type (a relationship type)* is *durable* when its objects (relations) are instances of the type during a certain time interval, i.e. they are not instantaneous. Durable types are the most usual ones.

Typical examples of durable entity types could be *Person*, *Student*, *Subject*, *Employee* or *City*. A relationship type like *RegisteredIn(Student,Subject)*, representing that a student is registered in a subject, is also durable.

Two particular cases of durable types deserve special attention: permanent and constant types. An *entity (relationship) type is permanent* if once an object (relationship) is an instance of the type it continues to be instance of the type until the end of the life span of the system.

For instance, an entity type *Book* is usually considered permanent since once a book exists, it exists forever (during the life span of a system). An example of permanent relationship type is *BornAt(Person, City)*.

In some cases, a type may either be considered permanent or durable. For instance, an entity type *Car* would be permanent if once a car is known to the system it exists forever (i.e. it is never deleted). Otherwise, if a car may be deleted from the system, *Car* would be durable.

The definition of a permanent entity or a relationship type means that its instances must satisfy an integrity constraint. This constraint can be formally defined as:

$$\begin{aligned} E(e,t) &\rightarrow \text{endsAt}(Ci(e,E)) = \text{endsAt}(Ls) \\ R(e_1,\dots,e_n,t) &\rightarrow \text{endsAt}(Ci((e_1,\dots,e_n),R)) = \text{endsAt}(Ls) \end{aligned}$$

Finally, we have that *an entity or a relationship type is constant* if its population is the same during the whole life span of the system. There are several example of constant population types, specially those that are independent of the domain. For instance, *Planet*, *Temperature* or *Color* have constant population.

According to the previous definitions, we have that instances of permanent and constant types can only be instance of the type during a single classification interval. Then, these types must also satisfy the following integrity constraint:

$$\begin{aligned} E(e,t) &\rightarrow |Ci(e,E)| = 1 \\ R(e_1,\dots,e_n,t) &\rightarrow |Ci((e_1,\dots,e_n),R)| = 1 \end{aligned}$$

3.2 Frequency

As far as frequency is concerned, we distinguish among single and intermittent types. An entity or a relationship type is *single* if its entities can only be instance of the type during a single classification interval. Otherwise, it is *intermittent*.

From the previous examples, we have that *Person* or *Sell* are single while *SneezingPerson*, *Student*, *HasRegistered* or *RegisteredIn* are intermittent. Note that the two relationship types would have single frequency if a student can be registered only once in a given subject.

Single types must also satisfy an integrity constraint that can be formally stated as:

$$\begin{aligned} E(e,t) &\rightarrow |Ci(e,E)| = 1 \\ R(e_1,\dots,e_n,t) &\rightarrow |Ci((e_1,\dots,e_n),R)| = 1 \end{aligned}$$

Since these constraints are the same than the ones required for permanent and constant types, these types must necessary have single frequency.

3.3 Valid combinations of temporal features

An entity or a relationship type may have a different value for each of its temporal features. However, as we have seen, not all these combinations are possible because some of them involve contradictory requirements since neither a permanent nor a constant type may be intermittent. Therefore, we have six possible ways to classify an entity or a relationship type as far as its temporal features are concerned:

- **Instantaneous, single.** *Sell* and *HasRegistered*, if we assume that a student can register only once in a subject, are examples of this type.
- **Instantaneous, intermittent.** *Visits(Doctor, Person)* and *CallingCustomer* are examples of this type.
- **Durable, single.** Like *Employee* or *WorksAt(Employee, Company)* if we assume that once an employee is fired it may never be hired again.
- **Durable, intermittent.** Typical examples include *Employee* and *WorksAt* if we do not make the previous assumption.
- **Permanent.** Like entity type *Invoice* if invoices are never deleted from the system once they are issued. Another example is *HasVisited(Person, City)*.
- **Constant.** *Planet* is a classical example of constant entity type.

4. Specifying temporal features in UML

The goal of this section is to extend the UML to specify the temporal features of entity and relationship types directly at the conceptual level and to provide notational devices to refer to past states of the information base. Using these extensions, a designer may use the UML as if it were a temporal modeling language.

A desired quality of conceptual modeling is that designers can perform it by using terms and concepts from the Universe of Discourse. If not, we get models with entities and attributes that define the temporal aspects and that make difficult to understand otherwise intuitive and easy-to-comprehend models [6]. Our extension provides two major advantages in this sense. First, temporal treatment is documented at a high-level as a first class feature and it is dealt with in a standard fashion. Second, the integrity constraints deriving from this temporal treatment must not be explicitly stated since they are already enforced by the temporal features.

4.1 Specifying temporal features of entity and relationship types

We use the standard extension mechanisms of the UML to propose an stereotype to define temporal information in this (non-temporal) language. This stereotype is called `<<temporal>>` and we call *temporally-extended* a conceptual schema that uses it.

The `<<temporal>>` stereotype allows the designer to define the temporal features of entity and relationship types. It contains two tags: durability and frequency that allow to specify the temporal properties of a GeneralizableElement (i.e. a Classifier or an Association). Moreover, we need to attach a constraint to the stereotype that guarantees that the designer selects one of the six valid combinations of temporal features. The definition of `<<temporal>>` is shown in figure 4.1.

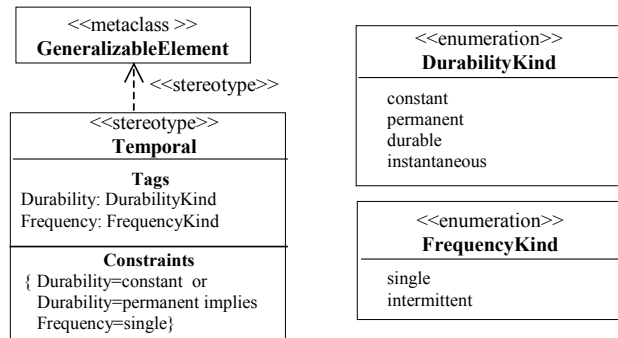


Fig. 4.1 –The `<<temporal>>` stereotype

As an example, assume we want to specify the information regarding employees and the projects they work in. We may have that *Employees* are durable and intermittent, since an entity may be an employee at different time periods; *Projects* are durable and single, since once a project is finished it is never started again; and the relationship type *WorksIn* is durable and intermittent, since an employee may work in a project several times. Figure 4.2. shows the representation of this temporal information in the UML using the `<<temporal>>` stereotype.

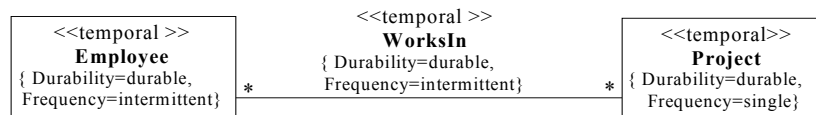


Fig. 4.2 – A temporally-extended conceptual model

It is well-known that attributes can always be represented as binary relationship types. Therefore, temporal attributes should be represented in this way.

4.2 Specifying temporal constraints in OCL

A complete conceptual schema must include the definition of all relevant integrity constraints [10] which may be either *static*, if they only refer to the current state of the information base, or *temporal*, if they refer to past states.

As we have seen, some temporal constraints are inherent to the definition of the temporal features of entity and relationship types. For instance, in the example of Figure 4.2 we have specified that once a project is finished it may never start again by stating that the entity type *Project* has single frequency. Nevertheless, most constraints must be explicitly defined [17, ch. 5].

As an example, consider the temporally-extended conceptual schema of figure 4.3. We may want to define the following constraints related to the salary of an employee:

- It must always be a positive value
- It may never decrease
- It may not raise more than a fifty per cent in single year

The first constraint is static since it involves only the current state of the information base while the other two are temporal because they refer to past states.

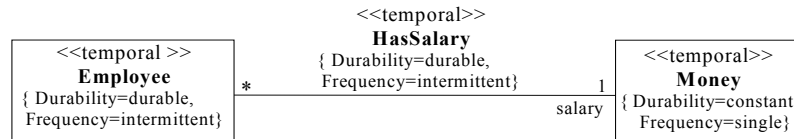


Fig. 4.3 – The salary of an employee as a temporal relationship type

The definition of explicit integrity constraints requires the use of a general-purpose sublanguage [3, ch.2]. In the UML this is usually done by means of the OCL language [13]. However, since the OCL does not provide constructs to navigate to past states of the information base we must find a way to explicitly define temporal expressions. A possibility could be to use some non-standard extension of the OCL with temporal logic [5, 20], but we have discarded it since we prefer not to deviate from the standard.

The key element in the definition of a temporal expression is the need to refer to the value of an attribute or a relationship type at a certain (past) instant of time i . To solve this problem, we assume the implicit existence of some temporal query operations that allow to retrieve information at i . These operations correspond to the temporal interpolation functions of [8].

For each binary relationship type $R(p_1:E_1, p_2:E_2)$ we need two different operations to navigate through the roles of the relationship type: $E_1::p_2At(i:Instant): Set(E_2)$, which returns the set of instances of E_2 related to an instance e_1 of E_1 at i and $E_2::p_1At(i:Instant): Set(E_1)$, with a similar behaviour.

Note that if the multiplicity of some of the roles is one, the result of the operation is not a set but an entity of the corresponding participant. Operations for relationship types with more than two participants would be defined in a similar way.

Moreover, the strict control of a constraint like “the salary of an employee may not decrease” requires, if an employee may leave the company and join it again later, to refer not just to the value of the relationship at i but also to the latest value of the relationship before i if no value exists at i . Then we need two other operations, $E_1::p_2AtOrBefore(i:Instant): Set(E_2)$ and $E_2::p_1AtOrBefore(i:Instant): Set(E_1)$ to obtain this information. As before, if the multiplicity of some of the roles is one, the result of the operation is not a set but an entity.

We require also an operation $allInstances(i:Instant): Set(E)$ for each temporal entity type E to refer to the instances of E at i .

Then, in our example of figure 4.3, we have three temporal query operations of *Employee*: $allInstances(i:Instant): Set(Employee)$, $salaryAt(i:Instant): Money$ and $salaryAtOrBefore(i:Instant): Money$. We can make use of these operations in the OCL expressions to specify the explicit temporal integrity constraints of our example:

- The salary of an employee may not decrease
context Employee inv:
 $self.salaryAt(t_today) \geq self.salaryAtOrBefore(t_today-1)$
- The salary of an employee cannot raise more than a fifty per cent in a year
context Employee inv:
 $self.salaryAt(t_today-365) * 1.5 > self.salaryAt(t_today)$

The behaviour of these operations is independent on whether they are used to specify temporal integrity constraints or for a different purpose. Then, these operations can also be used in the specification of operation pre/postconditions if they need to refer to past states other than the one before the execution of the operation.

5. Translating temporally-extended UML to conventional UML

The `<<temporal>>` stereotype allows the designer to specify the temporal features of the types in the Universe of Discourse. However, this does not necessarily mean that the information system being developed will need to store the history of a certain type. This decision is left to the designer who can choose, for each type, either to store its history or not. An entity or relationship type is called *historical* if the information base includes not only its current state but also all past ones. We show in this section how to translate a temporally-extended conceptual schema into a conventional UML schema both when types are non-historic and historic.

5.1 Non-historical types

Entity and relationship types for which we do not store its history are by default durable, since the information system records its instances during a certain period of time (from their creation until they are explicitly deleted); and single, since once an instance is deleted it is never created again (we may create another instance containing the same information but it will be a different one because it will have a different internal identifier). In this way, the entities of a durable non-historical type may be inserted and deleted when required.

Then, when types are not historical, we only have to explicitly specify when they are permanent or constant to indicate the information system the need to enforce the integrity constraints entailed by those features. In particular, the information system must guarantee that instances of a permanent type may be inserted but never deleted, while those of a constant type may never be inserted (except at the beginning of the life span of the system) nor deleted. We extend the UML with the `<<permanent>>` and `<<constant>>` stereotypes for this purpose.

5.2 Historical types

If we want the information system to physically store the history of entity and relationship types, we need to translate a temporally-extended conceptual schema into a conventional (non-temporal) UML schema. The mapping to perform involves the

translation of temporal entity and relationship types as well as the translation of temporal OCL expressions. To simplify the presentation, we assume only the valid time dimension although transaction time could also be dealt with in a similar way.

5.2.1 Historical entity types

To store the history of an entity type, the information base must register all classification intervals of each instance of the entity type. This is done by including an additional attribute whose goal is to explicitly timestamp the entity type. The type and the multiplicity of this attribute differ depending on the particular temporal features of the entity type.

On the one hand, the type of the timestamp attribute depends on the durability of the entity type. If it is instantaneous, the timestamp type must be *Instant*. If it is durable it must be *Interval*, since we must register the interval of time during which an entity is classified in the entity type. If an entity type is permanent the timestamp type must be *Instant*, since we only need to record the beginning of the classification period of each instance in the type. Finally, if it is constant we do not need any timestamp attribute at all since the life span of its entities corresponds to the one of the information system.

Note that, the current classification interval of an instance of a durable entity type is the one with an open interval. Instant and Interval are data types that represent, respectively, an instant of time and an interval of time with a given granularity (day, hour, second...).

On the other hand, the multiplicity of the timestamp attribute depends on the frequency of the entity type. Thus, if the entity type is single, the timestamp attribute will be single valued. Otherwise, i.e. if it is intermittent, it will be multivalued since we must register all classification intervals of each entity in the entity type.

We mark the timestamp attribute with the `<<timestamp>>` stereotype in order to distinguish it from other possible user-defined time attributes. A `<<timestamp>>` attribute must satisfy some constraints. For instance, if it is a set of intervals they cannot overlap while if it is a set of instants they must be distinct.

We show in figure 5.1 examples of entity types classified according to each of the six valid combinations of temporal features. Note that the translation of each temporal entity type results into a permanent (or constant) non-historical type because now when a given instance is declassified from an entity type we must update its timestamp attribute instead of physically deleting it.

For instance, as a result of this transformation, the class *Student* is permanent since its instances are never deleted. The different lifespan intervals denote the different classification intervals of an entity as a *Student*. Current students correspond to those entities with an open classification interval.

We must note also that permanent and single instantaneous entity types have the same representation. In the first case, the population of the entity type at the present time t is defined as the set of all entities that are instances of E at t . In the second one, the instances that exist at t are not only the instances that have been produced at t but also all entities that have been produced at all instants previous to t .

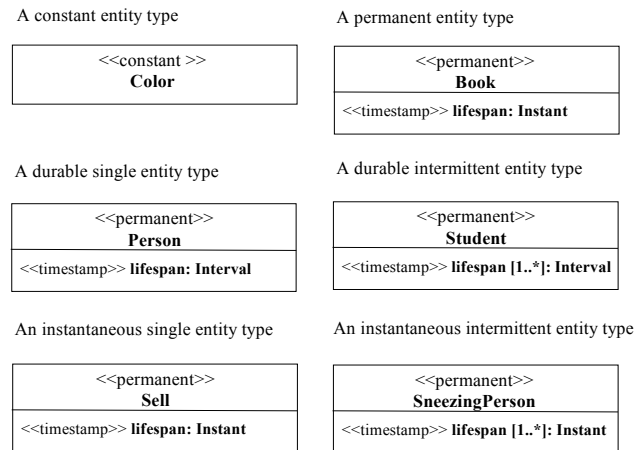


Fig. 5.1 – Examples of historical entity types in the UML

If an entity type is durable, we may also be interested to explicitly distinguish among the entities that belong to the type at the present moment and those that do not. This can be done by means of a derived subtype of the historical entity type, as shown in figure 5.2 for the *Student* entity type.

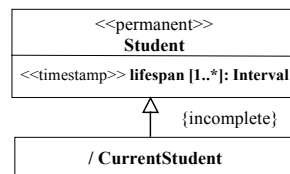


Fig. 5.2 – Specifying current students

All current students are students but there may be some entities that were students in the past but that are not now. *CurrentStudent* is derived from *Student* since it contains all students with an open classification interval.

5.2.2 Historical relationship types

To store the history of a relationship type we proceed in a similar way as we did for entity types. In this case, we associate a timestamp attribute to the reification of the relationship type (and we create this reification if it does not exist). Moreover, the type and multiplicity of the timestamp attribute depend also on the temporal features of the relationship type. Figure 5.3 shows several examples of the representation of historical relationship types.

If a relationship type is durable, we may also distinguish among the entities that belong to the type at the present moment and those that do not by means of a subtype of the historical relationship type. This should be done in a similar way as shown in figure 5.2 when specifying current entity types.

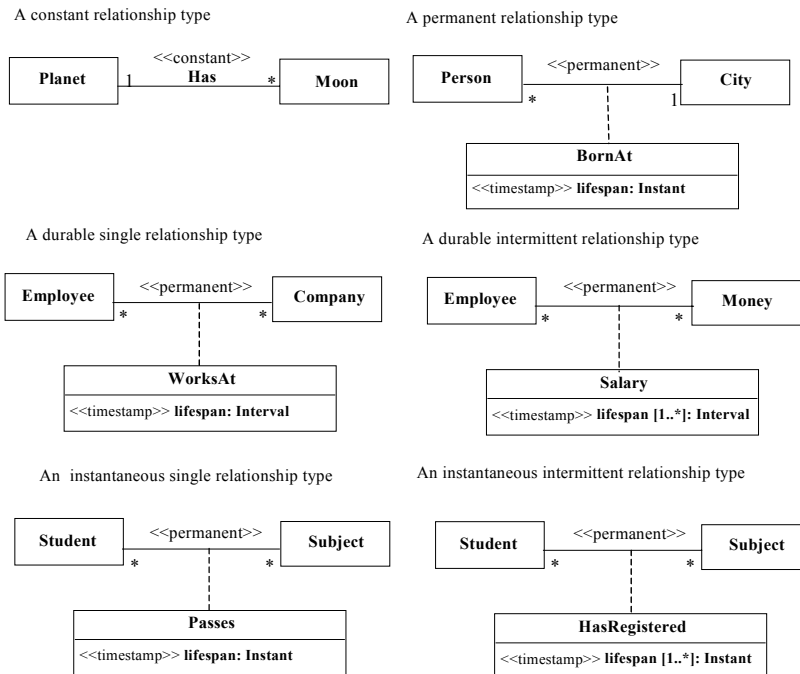


Fig. 5.3 – Examples of historical relationship types in the UML

If one of the roles in the initial schema has multiplicity one, the resulting transformation may have multiplicity * for that role, as it happens in the previous example for the salaries of the employees. At a given instant, an employee has a single salary. However, when we store the whole history, an employee may have had several salaries along his life span. This is why the multiplicity 1 becomes a *. Moreover, in this case, the transformation requires an additional constraint to ensure that the employee does not have more than one salary at a given instant, i.e. that the initial multiplicity constraint is not violated.

Most conceptual schemas will contain both historical entity and relationship types. In this case, we only have to apply to each model element its corresponding translation according to its temporal features. As an example, we show in figure 5.4 the transformation to (non-temporal) UML of the temporally-extended conceptual schema of figure 4.2.

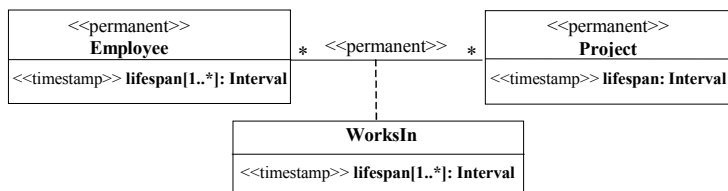


Fig. 5.4 – Employees, Projects and WorksIn as historical types

Clearly, it is better for the designer to specify a temporally-extended conceptual schema than explicitly defining the required transformation. First, because the conceptual model may become very awkward and difficult to understand if it contains plenty of temporal elements. Second, because the designer should learn the transformation to perform for each possible classification of a historical type.

Nevertheless, this transformation must necessarily be performed during design to provide an implementation of the information system. Nowadays, several case tools allow to implement extensions of the UML. We will describe in section 6 an implementation of these stereotypes that makes an automatic mapping between a temporally-extended conceptual schema and the corresponding translations.

5.2.3 Translation of temporal query operations

Once we know how to translate a temporally-extended conceptual schema into a conventional UML one, we may describe the transformation required to translate a temporal OCL expression into standard OCL. This transformation depends on the temporal features of the types that appear in the temporal OCL expression. Without loss of generality, we assume that the data type *Interval* is defined by means of two attributes (*startInstant* and *endInstant*) and that *Instant* is defined by a single attribute (*instant*). Due to space limitations, we only describe the transformation required for the operations of historical durable intermittent binary relationship types.

Given a binary relationship type $R(p_1:E_1, p_2:E_2)$ and assuming that C is the entity type that results of the reification of R , the operations $E_1::p_2At(i:Instant): Set(E_2)$ and $E_1::p_2AtOrBefore(i:Instant): Set(E_2)$ would be specified in standard OCL as:

```

context E1::p2At(i:Instant): Set(E2)
  post:
    let r: Set(C) = self.c->select(lifespan->select(startInstant<=i and endInstant>=i)->
      notEmpty()) in result = r. p2

context E1::p2AtOrBefore(i:Instant): Set(E2)
  post:
    let r: Set(C) = self.c->select(lifespan->select(startInstant<=i and endInstant>=i)->
      notEmpty()) in
      if r->isEmpty() then
        result= self.c -> select (c1 | c1.lifespan->select(endInstant<i)->notEmpty() and
          self.c-> forAll(c2 | c1<>c2 and c2.lifespan->select(endInstant<i)-> notEmpty()
            implies c1.lifespan->exists(i1 | c2.lifespan->forAll(i2 |
              i1.endInstant>i2.endInstant))). p2
      else result = r. p2

```

As an example, the operation *Employee::salaryAt(i:Instant):Money* would result:

```

context Employee::salaryAt(i:Instant): Money
  post:
    let r: Set(Salary) = self.salary->select(lifespan->select(startInstant<=i and
      endInstant>=i)-> notEmpty()) in result = r.money

```

The transformation required for the types with other temporal features would be similar to the one shown here for durable intermittent relationship types.

6. Implementation of the temporal extension

The implementation of the temporal extension allows to show the validity of our approach and, at the same time, it provides the users with the ability to graphically specify temporal information in the UML. We have implemented it in the Objecteering/UML CASE tool [11] which is one of the tools that admits the definition of stereotypes and tagged values. Figure 6.1. shows the example of figure 4.2 specified in this tool according to our implementation.

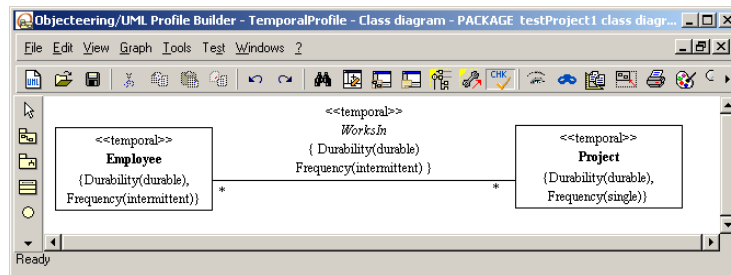


Fig. 6.1 – A temporal model specified in the Objecteering/UML tool

With this extension, the designer may specify not only the temporal features of entity and relationship types but also the standard (non-temporal) schemas resulting from applying the transformation described in section 5.2. However, it does not make much sense to manually specify the result of the transformation since it can be obtained automatically from a temporally-extended conceptual model.

We have implemented this automatic transformation in two different ways. The first implementation uses the scripting facilities offered by some CASE tools while the second one applies an XSL Transformation (XSLT) [18] to the XMI document [14] corresponding to the conceptual schema.

Scripting allows the designer to create scripts that access and modify the models within the tool. This is done by using some kind of metamodel interface offered by the tool. Different implementations will differ according to the chosen tool since each of them offers a specific scripting language and metamodel interface. For this reason, the main limitation of scripting is that the designer may perform automatically the transformation just in a single CASE tool (but not among different ones since they use different languages and metamodels).

Our implementation of scripting has been done in Objecteering/UML. It uses the UML Profile Builder of this tool and requires two different operations: one to transform temporal entity types and the other for temporal relationship types. These operations are implemented with J methods (written in the J language, the proprietary tool scripting language) that are attached to an specific metaclass or stereotype.

Our second implementation is based on the use of XMI. XMI is an interchange format for metadata that is defined in terms of the Meta Object Facility (MOF) standard [14, pp. 1-2]. As shown in [7], some tools allow already to import and to export a schema into (from) an XMI document.

If we use one of these tools, we can export the temporally-extended schema into an XMI document and use an XSLT processor to apply an XSL stylesheet to transform

the XMI document into an equivalent XMI document that contains the transformation of the original schema. Then, this latter document can be imported into another CASE tool to graphically depict the non-temporal model. In this second approach, the translation is performed outside the tool, by means of the XSLT processor. This idea is sketched in figure 6.2.

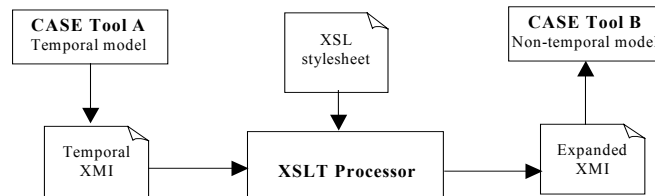


Fig. 6.2 – The process of the XSL Transformation

XMI is a standard specification. Then, we could expect to apply a single XSL stylesheet to transform the XMI documents generated by any CASE tool and to import them into a different CASE tool. This would allow the different members of a project to share the same schemas even though they may be using different tools. Unfortunately, the XMI documents generated by current CASE tools are seldom interchangeable at present. Then, we need to create a different XSL stylesheet for each tool that we want to use.

As an example, we have implemented an XSL stylesheet to transform temporal entity types specified in ArgoUML [1], an open source CASE tool that uses XMI as a native format for storing its models, and used Xalan [19] as the XSLT processor.

7. Conclusions and further work

We have proposed a standard extension of the UML that allows the designer to define a set of temporal features of entity and relationship types appearing in a conceptual schema. Moreover, we have also defined several temporal operations that allow to refer to any past state of the information base. With these extensions, a designer may use the UML/OCL as if it were a temporal conceptual modeling language. We have implemented this extension in the Objecteering/UML CASE tool. To our knowledge, ours is the first standard extension of the UML to define temporally-extended conceptual schemas in this language.

We have also proposed a mapping to translate a temporally-extended conceptual schema into a conventional (non-temporal) one. This mapping depends on the temporal features of entity and relationship types and translates both the static structure of the types as well as the required temporal query operations. This mapping can be automated and, as a result, we obtain a conceptual schema that can be processed by ordinary CASE tools, such as code generators. We have also performed two different implementations of this mapping.

There are other temporal features, like synchronism of the participants in a relationship type or mutability of a relationship type with regards to a participant, that

are of interest when representing temporal information. This is a direction in which our work can be continued.

Acknowledgements

We would like to thank D. Costal, C. Gómez and M. R. Sancho for their many useful comments to previous drafts of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnología and FEDER under project TIC2002-00744.

References

- [1] ArgoUML, <http://argouml.tigris.org/>, visited March 2003.
- [2] S. Bergamaschi, C.Sartori, "Chrono: a conceptual design framework for temporal entities", Int. Conf. on Conceptual Modeling (ER'98), LNCS 1507, Springer, pp. 35-50
- [3] D.W.Embley, B.D.Kurtz, S.N.Woodfield. "Object-Oriented Systems Analysis. A Model-Driven Approach". Yourdon Press, 1992.
- [4] A.Carlson, S.Estepp, M.Fowler, "Temporal Pattern", Pattern Languages of Program Design 4, Addison-Wesley, 1999, pp. 241-262.
- [5] S.Conrad; K.Turowski, "Temporal OCL: Meeting Specification Demands for Business Components", in K.Siau, T.Halpin (Eds.) *Unified Modelling Language: System Analysis, Design and Development Issues*, Idea Pub. Group, 2001, pp. 151-166.
- [6] H.Gregersen, C.S.Jensen, "Temporal Entity-Relationship Models: a Survey", IEEE Transactions on Knowledge and Data Engineering; 11(3), 1999, pp. 464-497.
- [7] Jeckle, M. "UML Tools (Case & Drawing)", <http://www.jeckle.de/umltools.htm>, visited March 2003.
- [8] C.S.Jensen, R.Snodgrass, "Semantics of Time-Varying Information", Information Systems; 21(4), 1996, pp. 311-352.
- [9] P.Kraft, J.O. Sørensen, "Translation of a high-level temporal model into lower level models", Int. Conf. on Conceptual Modeling (ER'01), LNCS 2224, Springer, pp. 383-396.
- [10] ISO/TC97/SC5/WG3. "Concepts and Terminology for the Conceptual Schema and Information Base", J.J. van Griethuysen (ed.), 1982.
- [11] Objecteering/UML, <http://www.objecteering.com>, visited March 2003.
- [12] A.Olivé, "Relationship Reification: A Temporal View", Int. Conf. on Advanced Information Systems Engineering (CAiSE'99), LNCS 1626, Springer, pp. 396-410.
- [13] OMG. "Unified Modeling Language Specification", Version 1.4, September 2001.
- [14] OMG. "OMG XML Metadata Interchange Specification", v. 1.2, January 2002.
- [15] R.Price, N.Tryfona, C.S.Jensen, "Extended SpatioTemporal UML: Motivations, Requirements and Constructs", Journal of Database Management, 11(4), pp. 14-27.
- [16] M. Svinterikou, B. Theodoulidis, "TUML: A Method for Modelling Temporal Information Systems", Int. Conf. on Advanced Information Systems Engineering (CAiSE'99), LNCS 1626, Springer, pp. 456-461.
- [17] B.Thalheim, "Entity-Relationship modeling. Foundations of database technology". Springer, 2000.
- [18] W3C. "XSL Transformations", Version 1.0, November 1999.
- [19] Xalan, <http://xml.apache.org/xalan-j/>, visited March 2003.
- [20] P. Ziemann, M. Gogolla, "OCL Extended with Temporal Logic", in *Perspective of System Informatics*, LNCS 2244, Springer, 2003.