Representing a system specification with a temporal dimension in an object-oriented language

Anders Oelmann

SYSLAB[1]
Department of Computer and Systems Sciences
Royal Institute of Technology and Stockholm University
Electrum 230, S-164 40 KISTA, Sweden
and
SISU[2], Box 1250, S-164 28, KISTA, Sweden

Abstract

One way to validate a conceptual specification is to transform it to an executable form and in this way have a prototype of the system, based directly on the specification. Before finding a proper transformation, a suitable representation in the executable form has to be found. This paper describes a way to represent a specification made with the TEMPORA conceptual model in the PROLOG-based object oriented programming language Amore.

1.Introduction

Making a formal specification of the software system to be designed mainly serves the purpose of obtaining a description of the planned system which is complete and non-ambiguous. This description can be validated with respect to the requirements, checked for inconsistencies and for deficiencies in quality. Furthermore, the specification can be translated to an executable form, and in this way become a prototype of the future system. The prototype will serve as a tool for validating the specification. There are great advantages having an executable specification: First of all we will have a prototype early in the development phase, and secondly, validating the prototype is equal to validating the specification. If we find that the prototype fulfils our requirements, we also know that the specification does so.

In the ESPRIT research project TEMPORA, a formal specification language, among other things, is developed. The intention of this paper is to describe a possible way to represent such a system specification in the object-oriented language AMORE [RUBRIC], developed in the ESPRIT project RUBRIC. The specification has been done, as far as possible, according to the formalism which currently is developed by the TEMPORA project. A case study has been made, and it has resulted in an executable prototype of the specification.

---

[1]This work has been partly supported by the National Swedish Board for Technical Development (STU).

[2]Swedish Institute for Systems Development

2.TEMPORA

The TEMPORA project is a 5 year collaborative research and development programme partially funded by the Commission of the European Communities under the ESPRIT programme. The TEMPORA consortium consists of BIM (Belgium), Hitec (Greece), Imperial College (Great Britain), LPA (Great Britain), SINTEF (Norway), SISU (Sweden), University of Liège (Belgium) and University of Manchester Institute of Science and Technology (Great Britain).

The TEMPORA project builds on a rule-oriented system development and extends this paradigm with the explicit modelling of temporal aspects at both specification and application levels. The TEMPORA model is capable of dealing with historical information issues as well as being able of modelling temporal business rules.

2.1. The temporal model

In order to express temporal requirements, TEMPORA has adopted an approach based on temporal logic. Instead of treating time as any other property of concepts, temporal operators are used to support the specification and make it easier to understand. Without temporal operators, temporal requirements must be expressed in terms of complicated expressions on the properties representing time. The language with which we may express temporal rules and queries will in the sequel be called TQL. The basic temporal operators used in Tempora are shown in table 1.

Operator	Pronunciation	Meaning
◇A	eventually	A is true or will be true at some time in the future
◆A	was	A is true or was true at some time in the past
○A	next	A will be true at the next time-point
●A	previous	A was true at the previous time-point
□A	henceforth	A is always true in the future
■A	heretofore	A is always true in the past

Table 1. A is a well formed formulae (wff).

In addition, new temporal operators may be defined in terms of the basic ones. Operators taking in account different units of time have been defined. With these operators it is possible to form expressions using the the temporal units which are normally used in the modelled area, e.g. second, hour, day, month, year, quarter, decade etc.

$\bullet_{\geq}$(A, n.unit) holds if A was true during the previous n units of time.
$\bullet_{=}$(A, n.unit) holds if A was true for one unit, n units ago.
$\bullet_{\leq}$(A, n.unit) holds if A was always true until n units ago.
$\blacklozenge_{\geq}$(A, n.unit) holds if A is true within a period beginning n units ago.
$\blacklozenge_{=}$(A, n.unit) holds if A was true n units ago.
$\blacksquare_{\geq}$(A, n.unit) holds if A has been true for n units.

All times consider the event time, the time at which the system believes that an event took place in the real world. The transaction time, the time at which the system records its belief that an event took place, is not handled in within this model.

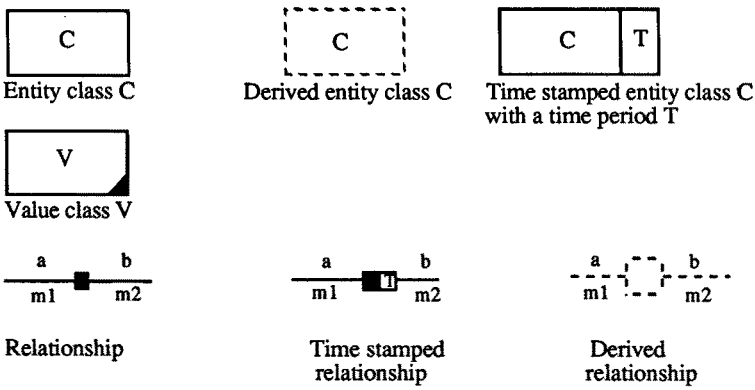
2.2. The conceptual model

A specification developed using the TEMPORA model consists of three components: the *structural component,* the *process component* and the *rule model.* The structural

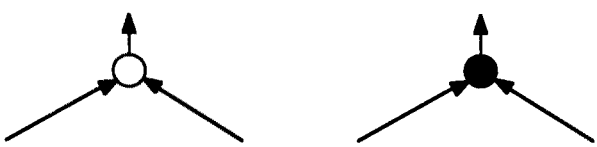
component constitutes a description of the structure of the information, employing a binary-relationship model named *ERT-model*. The process component deals with the definition of operations. A process is the smallest independent unit of business activity of meaning, initiated by a specific trigger and which, when complete, leaves the business in a consistent state. In the rule model rules are divided into static and dynamic rules. Static rules are those which must hold in each valid state of the database, while the dynamic rules are expressions that define valid state transitions in the database.

2.2.1. Structural component

The structural component consists of the ERT-model, which is a binary ER-model, extended with the notion of time. Fig 2.1 shows some of the symbols of the ERT-model:



Relationships connects two entity classes or an entity class and a value class. a and b are relationship names, m1 and m2 indicate mapping constraints in the format (min:max).



Partial ISA-relationship Total ISA-relationship

Two or more arrows going to the same circle indicate disjoint subsets

Figure 2.1 The Amore structural component

When a relationship or an entity class is timestamped, the history of its instances is accessible, while only the current state of the instances of non-timestamped is accessible. Instances of a derived entity class or relation are not stored, instead they are derived from other entities, relationships and values.

2.2.2 The Rule Model

Static rules

The static rules are expressions that must hold in every valid state of the database. It can be said to hold (or not hold) simply on the basis of the extension of the database with

respect to a single state. The static rules are divided into integrity rules and derivation rules.

The purpose of a static integrity rule is to restrict the set of valid states of one or more items of data. The constraints consider cardinality of relationships, domain of attributes, set membership etc.

A derivation rule is a definition of an entity class, relationship class or value classes in terms of other entity classes, relationship classes or value classes.

Dynamic rules

The dynamic rules are expressions that define valid state transitions in the database. It can be said to hold (or not to hold) only by examining at least two states of the database. In effect the dynamic rules specify the interaction of state components and/or event occurrences and they represent either dynamic integrity rules or control of operations.

Dynamic rules are used to specify the conditions under which messages for operations are generated, express the business policy in understandable terms, provide an executable specification that can be used for prototyping purposes, specify the details of the triggers and preconditions that define the different processes, verify that the ERT-model can support the business rules and dynamically derive data.

Transition constraints

Dynamic integrity rules specify restrictions on the behaviour of systems by limiting the conditions under which certain database operations can occur. It can for example be stated that a book may be borrowed only on weekdays.

Action rules

A dynamic action rule defines the conditions necessary for invocation of an operation. Dynamic action rules have a three-part structure, consisting of a *trigger*, a *precondition*, and an *action*.The trigger and preconditions describe the conditions under which a rules becomes fireable, whilst the action part of the rule generates messages to fire operations. A rule has the form: [**WHEN** trigger] [**IF** precondition] **THEN** action.

The *trigger* initiates the execution of the rule. It can e.g. be an external signal, a clock condition or an internal state condition. The *precondition* is a condition which is evaluated whenever the dynamic action rule is fired The *action* is a message sent to an entity class consisting of an operation and parameters.

Example:

WHEN	borrow_request (CLIENT.CL, BOOK.B)
IF	exists(COPY.C that is of BOOK.B)
THEN	client.c <- borrow(COPY.C)

When borrow_request is triggered it is checked if there exists a copy of the requested book. In that case the action which lets the client borrow the copy is invoked.

3. AMORE - A Method Object Rule Environment

3.1 Why AMORE?

The reason why AMORE was choosen as the language for prototype implementation is that it was developed, by UMIST, as the implementation language in the preceding ESPRIT

project RUBRIC. In TEMPORA a similar language, PROBE, is used to represent the ERT-model.

3.1. Amore in general

Amore is an object oriented language based on Prolog. It combines the structuring features of object oriented programming with logic programming. It also contains facets of variables, a feature which can be found in languages for knowledge representation like KRL [Bobrow77].

A program written in Amore program is compiled to a Prolog program. An Amore program consists of a set of class definitions. In the class definition superclasses, slots (= variables), and methods are declared. A method declaration is like any definition of a Prolog predicate. It has a head (= method name) and a body. The body consists of a conjunction of Prolog predicates and/or messages. A message may like any PROLOG predicate succeed or fail, but it can only have one solution. This means that a message can not be backtracked. There is however a *protected mode* in which all changes done to slots, but only to slots, are roll-backed in case the message initiating the change should fail. This may be used as a primitive form of transaction control. In the proceeding we assume that Amore will be used in the protected mode.

Classes in Amore have the same meaning as in most other OO-languages, i.e. that it is a collection of objects which have certain properties in common. Which these properties are, is defined in the *class definition* of the class. A class is represented by an object, a *class object*, in the Amore system. An instance of a class C is created by sending the message `new` to C.

The abstraction principle which normally is best supported by object-oriented languages is specialization/generalization. This is achieved by inheritance. In AMORE inheritance means that a class inherits slots and methods from its superclass. In addition AMORE supports multiple inheritance, which means that a class may have more than one superclass.

3.2. Slots

A slot's primary function is to contain one or more values, or references to other objects. A slot can be either single- or multi-valued. If it is single valued, the slot contains a value. If it is multi-valued the slot contains a list of values. An empty slot contains either the value *undef* or the empty list , depending on if it is single- or multi-valued.

There are two kinds of slots, *instance slots* and *class slots* A class slot is a slot which is common for the class, and belongs to the class object, while an instance slot exists in each instance. In the class definition, class slots are preceded by the keyword *private* and instance slots are preceded by the keyword *common*.

In addition, a slot may have a number of optional facets, for various purposes, associated with itself. There are e.g. facets for constraining the contained value, derive the value or invoke some other action on access of the slot. If the value of the slot should violate any of these constraints or if any of the facet's actions should fail, the access is considered to be illegal and will fail. Any attempt to assign an illegal value to a slot will be inhibited. The following slot facet exist:

• Type facet
The type facet states the valid value set for the slot.

• Constraint facet
The constraint facet provides an all-purpose constraining facility by accepting any Prolog goal as constraint. All values for which this goal is true, is valid with respect to the constraint facet. If we want the value set to be an enumerated set, this can be stated as a constraint facet.

• Cardinality facet
The cardinality facet restricts the number of values that can be stored in a slot by a lower and an upper limit. If the upper limit >1 then the slot will consist of a list of values, otherwise it will consist of the value itself.

• Inverse slot facet
Sometimes when we have a slot S1 in a class C1 representing a relationship to another class C2, i.e. the type of the slot is `instance_of(C2)`, we also want to state that there is an inverse relationship going from C2 to C1. This can be declared in the inverse slot facet. The effect of using this facet is that when updating the slot S1, the inverse slot in C2 is automatically updated.

• Inheritance facet
In case of a class inheriting a slot from its superclass the facets of the slot are inherited as well. It is, however, possible to modify these inherited facets if necessary. There are two ways of doing this: either new facets are declared which replace the ones of the superclass (`override`), or the new facets are appended to the inherited ones (`append`).

Using the override option enables possibilities to redefine inherited slot specifications. This allows over-generalisation of object classes, i.e. that a class is given properties which are not valid for all its subclasses. For those subclasses where this property is not applicable a redefinition is made by overriding the property of the superclass. Over-generalisation can be motivated when there is a property that normally is valid, but not always. The advantage of allowing over-generalisation is discussed in [Borg88].

• Default value facet
Holds the default value of the slot.

• Daemon facets
Daemon facets are procedures which are triggered and executed when the slot is accessed. There are four types of daemons: `if_needed`, `before_changed`, `when_changed` and `after_changed`. The `when_changed`- and `after_changed`-deamon are in fact synonyms, since they work in exactly the same way. In the proceeding we will only discuss `after_changed`, while we actually mean both of them.

An `if_needed`-deamon is invoked when the value of the slot is accessed, and it can be used for deriving the value of the slot, or check access authority. As their names indicate, `before_changed` is invoked before, and `after_changed` after a slot's value is changed. It is suggested [AMORE] that `before`- and `after_changed`-deamons can be used for implementing pre- and post-conditions for updating of a slot, but since all deamons consist of a PROLOG predicate they can be used for a number of other purposes as well.

They can e.g. update other slots or initiate some process by triggering a dynamic action rule.

When there is an append inheritance facet, the execution order of the `before_changed`-deamons goes from the most specific to the most general, i.e. that first the most specific deamon is invoked then its super and so on until the most general class containing this slot or until a class which has inheritance facet override is reached.

3.3. Methods

While the slots can be said to be the structural description of the class, the methods describe the behaviour of the class. When a message is sent to an object, the object reacts by invoking the corresponding method or methods.

In Amore there are three kinds methods: `before`-, `primary`- and `after`-methods. The difference between them are their rules of inheritance and the way in which they are invoked. A `primary` method overrides the `primary` method with the same name of the superclass, while the `before`- and `after`-methods add their behaviour to their counterparts in the superclass.

When an object receives a message, first all `before`-methods are executed, then the `primary` and finally all `after`-methods. The `before`- and `after`-methods are executed in the order given by the precedence list, which is ordered according to the "is subclass of"-relation. The execution of the before methods starts with the most specific class, while the execution of the after methods starts with the most general.

Example:

A class A has a subclass B which has a subclass C. All three classes contain definitions of a method M. The definitions consider `before`- `primary`- and `after`-methods. When sending the message M to class C the methods are invoked in following order:
C.m.before, B.m.before, A.m.before, C.m.primary, A.m.after, B.m.after, C.m.after.

For each message an object can respond to, there must exist one primary method, either in the class definition of the own class, or in one of the inherited classes. The other kinds of methods are optional. If any of the invoked methods should fail, the result of the message is fail, and all changes made to any slot are 'undone'.

3.4. Messages

Communication between objects are carried out through messages. There are messages for activating methods, retrieving a slot's value and for assigning a value to a slot. The syntax of the messages are:

`obj <- msg(<pars>)`	The message `msg` with a list of parameters `<pars>` is sent to the object `obj`.
`obj <- slot=>_var`	The variable `_var` is assigned the contents of slot of the object obj.
`obj <+ slot=>_var`	As above, but no deamons are activated.
`obj <- slot := _var`	The slot slot of obj is assigned the contents of `_var`.
`_var := obj <+ slot`	As above, but no deamons are activated.

A message behaves like a PROLOG predicate in that way it can either succeed or fail, but it has never more than one solution. If a message fails, all changes to any slot caused by the message will be 'undone'.

There are two special variables for message sending: *self* and *super*. Self contains a reference to the object itself, and is used when an object makes a reference to its own slots or calls one of its own methods. Super contains a reference to the object's superclass.

3.5. Example

An example of a simple class definition:

```
defclass employee.                          Definition of class employee
   supers person.                           Superclass = person
   private.                                 Section for class variables begins
   slots.
      number,                               Slot name
         type integer,                      Type facet
         cardinality(1,1),                  Cardinality facet
         after_changed(write(it)).          After-changed deamon

   common.                                  Section for instance variables and methods
   slots.
      works_for,                            Slot name
         type instance_of(company).         Type facet

   method primary.
   take_job(_new_employer) :-               Definition of the primary method take_job
      self<-works_for := _new_employer.
   endclass.
```

The variable *it* is used in the facets contains the value being assigned to the slot.

4. Mapping TEMPORA concepts to AMORE

When having specified a complete system in terms of an ERT-model, static constraints, static derivation rules, dynamic constraints and dynamic action rules, comes the problem of mapping these parts to an Amore program in order to obtain an executable prototype of the system. This translation should naturally be automatic, if achievable. The rest of this paper contains a description of how the different parts of the conceptual model can be represented in Amore. The description is highly informal and does not claim to be complete, but should nevertheless give an idea of the possibilities to make this sort of mapping.

4.1. The TEMPORA temporal model

In addition to the conceptual model there must be mechanisms implemented, supporting the temporal representation and reasoning. There must be means by which it is possible to describe time dependent entities and relationships, and to evaluate rules and queries containing a temporal component. Queries and rules concerning historical information are expressed in the Temporal Query Language (TQL). Instead of suggesting rules for translating TQL-sentences into Amore procedures we have outlined an interpreter in Prolog which accepts a subset of TQL. These and a few other predicates are described below. How the interpreter works will not be discussed here.

`tql(P)`
Holds if the TQL-interpreter finds a valid solution for P.

`sometime_past(P)`
Holds when predicate P is true in this or in some previous state. Corresponds to ◆P.
Example: `sometime_past(works_for(john,ibm))` is true if John works or has worked for IBM.

`within_previous(P,T,N, UNIT)`
Holds when the predicate P was true within a period lasting N units, ending at T. This is a generalization of $\blacklozenge_{\geq}$(A, n.unit).

Example:
`within_previous(works_for(john,ibm), ([89,12,11]), 3, month)` is true if John worked for IBM sometimes during the period 890911 -- 891211.

`beginpredicate(P,T)`
Holds if the predicate P becomes true in a database state beginning at T.
Example:
`beginpredicate(works_for(john,ibm), ([89,01,01])` is true if John started to work for IBM on 890101.

`now(T)`
Holds if T is the current time.

`included(T)`
Is true in a sequence of states if T is included in the interval of time on which the states holds.
Example:
`works_for(john,ibm)` ∧ `included([89,02,01])` is true if John worked for IBM the 890201.

`t(Ts,Te)`
Is true in a sequence of database states if Ts is the start time of the first database state of the sequence, and Te is the end time of the last database state in the sequence.
Example:
`works_for(john,ibm)` ∧ `t(Ts,Te)` is true if Ts is the time when John started to work for IBM, and Te is the time when he stopped working for IBM. If he is still working for IBM, Te will have the value 'now'.

`time_intersect(P0, P1,I)`
Holds when I is the intersection of the time periods P0 and P1.
Example:
```
time_intersect( ([90,03,01],[90,03,20]), ([90,03,15],[90,03,30]),
                ([90,03,15],[90,03,20]) ) is true.
```

`time_greater(T0,T1)`
Holds when time point T0 > T1.

`trunc( T0, T1, UNIT)`
Holds when T1 is the time point T0 truncated with respect to time unit.
Example:
`trunc([90,03,23], [90,03,00], month)`

5. Mapping the ERT-model to AMORE

This chapter discusses how different aspects of a TEMPORA conceptual schema can be represented and implemented with Amore. The examples are taken from the specification of a simple library case with strong requirements for temporally related information.

5.1. Basic components

The components of the ERT-model, entity classes and relationships (timestamped or not, derived or not) must have their counterparts in Amore's object oriented model. What seems to be the most natural representation, is to represent an entity class with an object

class, and a relationship with a slot on each object class representing the involved entity classes. This is possible since all relationships are binary.

An isa-relationship will then become a has-superclass relationship in Amore. In addition to ordinary isa-relations it is possible to restrict the membership of the involved classes by the *disjoint entity class*-rule and the *total entity class*-rule. The representation of these rules are discussed under 'Static Integtity Rules', 6.1.2.

The cardinality constraints of the ERT-model are mapped to the cardinality facet of the slot, and the domain of the property is stated in the type facet.

ERT	**Amore**
Entity class	object class
Entity	object
Relationship	pair of slots
cardinality constraint	cardinality facet
isa	has superclass

Figure 5.1: Mapping of basic concepts

Each object-class representing an ERT entity-class is a subclass of 'entity'. This class contains general methods for instance creation, deletion as well as consistency checks. Which these checks are will be described later.

5.2. Value class

A relationship which has a value class as domain which is one of the pre-defined AMORE types : integer, float, boolean or string, has that type as a type facet. If a relationship has another type of value class this must be stated in the constraint facet. Such a value class could for instance be an enumerated set.

5.3. Derived relationships

A derived relationship is implemented by a slot which has a `when-needed` facet holding the derivation rule. In this way a derivation is performed whenever the slot is read.

Applying these mappings on the ERT schema in figure 5.2 gives the Amore class definitions in figure 5.3.

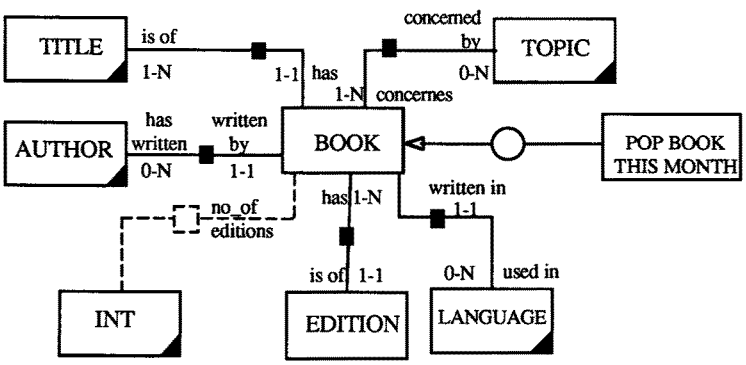


FIGURE 5.2

```
defclass book.                                  /* Entity class */
   supers entity.
   slots.
      written_by,                               /* Relationships */
         type string,
         cardinality(1,1),

      has_title,
         type string,
         cardinality(1,1),

      concerns,
         type string,
         cardinality(1,N),

      has_edition,
         type instance_of(edition),
         cardinality(1,N),
         inverse_of is_of,

      is_written_in,
         constraint(member(it, language<-values)),       /* Value constraint*/

      no_of_editions,
         type integer,
         cardinality(1,1),
         when_needed((self<-has_edition=> _editions, length(_editions,it))).
                        /* Derivation rule for derived relationship */
endclass.

defclass edition.
   supers entity.
   slots.
      is_of,
         type book,
         cardinality(1,1),
         inverse_of has.
endclass.

defclass pop_book_this_month.
   supers book.

endclass.

defclass language.                                  /* Value class */
   slots.
   values,
      default([english,german,french,spanish).
endclass.
```

Figure 5.3

5.4. Timestamped entities and relationships

Timestamped entities an relationships are used in the TEMPORA model when the history of an entity/relationship is important. In an executable prototype it is therefore necessary to be able to preserve the history of those entities/relationships. For this purpose versionable objects are introduced. Each entity class which is either timestamped or involved in a timestamped relation is made versionable. This is achieved by mapping those entity classes on two Amore-classes. One representing the current state of the entity class and the other the history of the entity class. Each instance of the 'history' class represents a version of an entity. The 'current' class is accessed when static rules and queries concerning only the current state are evaluated. When dynamic rules or queries concerning a previous state are evaluated, the 'history' class has to be consulted. The

'history' objects are only accessed by the TQL interpreter, since all references to previous states are expressed in TQL.

For each versionable object representing an existing entity there exists one instance of the 'current' class, and one or more instance of the 'history' class.

When a versionable object is updated the instance of the 'current' class is updated destructively, and a new instance of the 'history' class is created to hold the new state, while previous versions remain unchanged. The creation of new 'history' objects is initiated from the 'current' object. Each slot in the 'current' class has a when-changed deamon which, when triggered, creates a new version.

The class 'history' has three slots, two for the timepoints when the period of validity of the instance starts and ends, (start_time and end_time). The third slot, current, holds a reference to the instance of the 'current' class which the 'history' object is a version of (If the entity exists in the current state).

```
defclass history
   slots.
      current,
      start_time,
         type instance_of timepoint,
      end_time,
         type instance_of timepoint.
endclass.
```

Figure 5.4 Definition of the history class

When a non-timestamped entity E, which is involved in a timestamped relation, is deleted, then all timestamped relations which the entity is involved in, are also removed. In other words the outdated versions of E are also deleted. On the other hand, when a timestamped entity is deleted then all outdated versions remain. This means that both involved entity classes must be timestamped if the information held by a timestamped relationship must survive deletions of involved entities.

Example:

Copy_on_loan in the ERT-schema in fig.5.5 will be mapped to the class definitions in fig 5.6. The class definition of client is not shown but will be mapped in the same way.

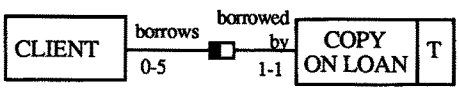


Figure 5.5

```
defclass copy_on_loan
   supers entity,
   slots.
      borrowed_by,
      when_changed(copy_on_loan_history <- new_version(self,borrowed_by,it)).
endclass.

defclass copy_on_loan_history
   supers entity, history
   slots.
      borrowed_by.
```

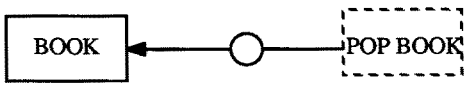
```
   method primary
   new_version(_current, _property, ,_newValue):-
      now(_now),
      /* Set end_time of previous version */
      instance_of(copy_on_loan_history, _previousVersion),
      _previousVersion<-current=>_c, _c=_current,
      _previousVersion<-end_time=>_endTime, endTime=='unbound',
      _previousVersion<-endTime:=_now

      /* create new version and set its start_time */
      self<-new(_newVersion),
      _inst<-_property := _newValue,
      now(_now),
      _inst<- startpoint := _now,
      _inst<- endpoint := 'unbound'.
endclass.
```

Figure 5.6 The versionable object copy_on_loan

5.5. Derived entity classes

A derived entity class is always implemented as an object class, which contains a derivation rule stating under what circumstances an entity exists.

Example:




```
pop_book(author, title) ⇐
      book(author, title)
      ∧ time(t)
      ∧ trunc(t, t0, month)
      ∧ findall(nr,
                (♦≥ (instance_of(copy_on_loan,C),
                              copy_on_loan(nr, author, title, edition,
client)
                    ∧ ●¬ copy_on_loan(nr, author, title, edition,
client)), t0,12.month),
                loans)
      ∧ length(number_of_loans, loans)
      ∧ number_of_loans > 7.
```

FIGURE 5.7: Derived entity class with derivation rule. The derivation rule (is_pop_book) states that a book is a pop_book if it was borrowed more than 7 times during the previous 12 months.

```
defclass pop_book_this_month.          /* DERIVED ENTITY CLASS */
   supers book.
   private.
   method primary.                     /* derivation rule.          */
      pop_book(_author,_title) :-      /* True if book written by _author with title _title  */
      now(_T0),                        /* is a pop_book this month.                  */
      tql((trunc(_T0,_T1,month),
      findall(_nr,within_previous((beginpredicate(instance_of(copy_on_loan,_C)
                                  _C<-edition<-book<-author=>_author,
                                  _C<-edition<-book<-title=>_title, _T),
                  t(_T,_)),
                _T1,12,month),          /* true if _T is within previous 12 months */
                _loans))),
      length(_loans, _no_of_loans),
      _no_of_loans > 7.
endclass
```

FIGURE 5.8: Amore implementation of derived entity class in fig 5.7.

5.6. Complex object classes

A complex object can be represented by an object that has slots for its subparts and for the relationships it is involved in. To be able to distinguish between slots containing subparts from slots representing relationships, there is a class slot `has_component` holding the names of the slots containing references to the subparts. The contents of the slot is defined by the default value slot, and may not be changed.

Example:

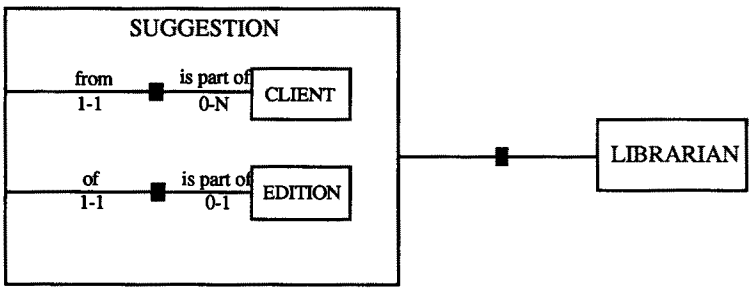


FIGURE 5.9

```
defclass suggestion.
   supers entity.
   private.
   slots.
      has_component,
      default [of,from].
   common.
   slots.
      of,                                  /* subpart */
         type instance_of(edition),
         cardinality(1,1),
      from,                                /* subpart */
         type instance_of(client),
         cardinality(1,1),
      received_by,                         /* relationship */
         type instance_of(librarian),
         cardinality(1,1)

            private.
```

FIGURE 5.10

Though not a part of the TEMPORA's complex objects, it is possible to let the is_part_of relationship be dependent/independent as described in [Kim89]. If an is_part_of-relationship is dependent, it means that the subpart can not exist without the main object, and must therefore be deleted when the main object is deleted. This can be implemented by calling the method delete of all subparts from the method delete of the complex object.

5.7. Complex value class

A complex value class is represented in the same way as a complex entity class. The only difference is that its subparts must be of a value class, and therefore there is no need for methods creating and deleting the subparts.

Example:

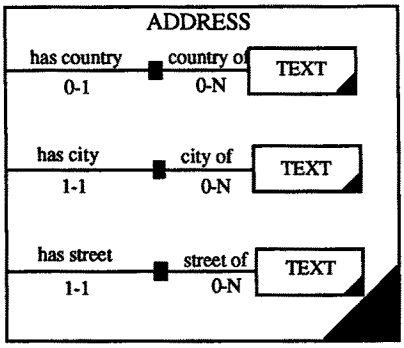


FIGURE 5.11

```
defclass address.
   private.
   slots.
      has_component,
         default [country,city,street].
         common.
   slots.
      country,
         type string,
      city,
         type string,
      street,
         type string,
      :
endclass.
```

FIGURE 5.12

6 Mapping rules to AMORE

In this section we give examples how some of the different kinds of static rules presented in [TEMP] may be mapped to Amore.

6.1 Static rules

6.1.1 Static derivation rules

See *derived entity class* and *derived relationships* above.

6.1.2 Static integrity rules

Relationship cardinality rules are mapped to cardinality facets and type constraints are mapped to type facets of the slots which represent relationships. This gives a declarative representation of these rules, as well as the possibility to use the built-in consistency check. For the other types of rule there is no such support in Amore. Even though there is no special support for the other types of rules, the ambition has been that also these rules shall be expressed declaratively. The reason for this is that the mapping and readability of rule-instances then will be much easier. This has been done by letting certain class slots hold the rules. The consistency control is performed by methods located in the class 'entity'. Since this is the superclass of all entity classes, these methods are also accessible from all entity classes. The invocation of the consistency check must be done by each entity class. Normally a consistency check has to be carried out whenever a new object is created, deleted or changed. This is achieved by adding a

'when-changed' facet to all slots whose values can have an effect on the consistency. The action of the facet is to invoke the inherited constraint check method. There are slots for 'uniqueness rule' (the identifier), 'entity class cardinality' and the 'disjoint entity class rule' (see below)

Disjoint entity class

The disjoint entity class constraint specifies that the sub classes of an entity class do not have any common entities (i.e. their intersection is empty).

When talking about disjoint entity classes we are only interested in disjoint classes which have a super-class in common, since classes which have no common superclass are by definition [TEMP] disjoint. The implementation of the rule must assure that no instance exists in more than one class involved in the disjunction. This means that having disjoint classes C1 and C2 which have elements identified by the attribute A1 and A2 respectively, we want to check that for each element in C1 there is no element in C2 with the same value of A1, and for each element in C2 there is no element in C1 with the same value of A2. Note that A1 and A2 may be the same attribute but it is not necessary. This check is implemented by the method `disjoint_rule_violated` in the class `entity`.

```
defclass entity
private.

method primary.
disjoint_rule_violated(_E1) :-
    instance_of(_E1, _C1),
    _C1<-identifier=> _id,
    _C1<-disjoint_rule=> _DisjointClasses,
    member(_C2, _DisjointClasses), not(_C2 == _C1),
    instance_of(_E2, _C2),
          not(member(_slot, _id),  _E2<-_slot \== _E1<-slot)).
```

FIGURE 6.1 The predicate `disjoint_rule_violated` succeeds if there are two entities _E1 and _E2, which belong to two different entity classes _C1 and _C2 which are disjoint, and _E1 and _E2 have the same value(s) in their identifying slot(s). The other rules are not shown, but they are defined in a similar way.

Example:

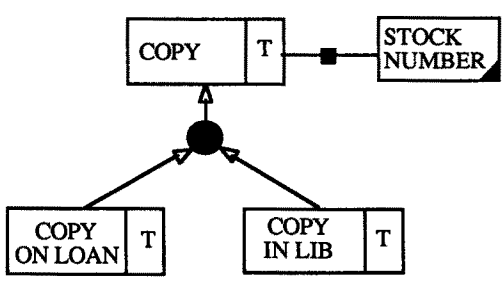


FIGURE 6.2

The schema in fig. 6.2 express the rule: copy in lib and copy on loan are mutually disjoint

If we add the rule a copy is identified by its stock number , the both rules can be represented in the slots *disjoint_rule* and *identifier* of the class *copy*.

```
defclass copy.
   supers entity.
      :
   private.
   slots.
      identifier,
         default [has_stock_number],

      disjoint_rule,
         default [copy_on_loan,copy_in_lib],
   common.

   slots.
      has_stock_number,                                  /* identifying attribute */
         type   integer,
         cardinality(1,1),
         when_changed(/* invoke consistency check */
                      not(self<-uniquness_rule_violated),
                      not(self<-disjoint_subclass_rule_violated)),
```

FIGURE 6.3: Implementation of schema in fig 6.2

Total entity class

The total entity class constraint specifies that the union of two or more entity classes is equal to their superclass. In other words there are no entities belonging to the superclass that do not belong to any of its subclasses. This is achieved by letting the superclass be an abstract class, i.e. that there will be no instances of the superclass, only of its subclasses. As have been described earlier, abstract classes are implemented by letting the class in question have a method `new` which prohibits creation of any instances of the class.

Entity cardinality

The entity cardinality constraint limits the number of entity occurrences within a given entity class.

To implement this rule we provide the class in question with a class variable which is to contain the current number of elements of the class, and by constraining the value of this variable we limit the number of instances. The variable is updated by the methods creating and deleting elements which increases and decreases its value respectively.

Example:

number-of(CLIENT) < 10000

```
defclass client.
   supers suggested_edition.
   private.
   slots.
      number_of_instances,
         type integer,
         cardinality(1,1),
         default 0
         constraint(it<10000).

   private.
   method primary.
   create :-
      self <- number_of_instances => _N, _N1 is _N+1,
      self<-number_of_instances := _N1,
      .
```

```
common.
method primary.
delete :-
    instance_of(_class, self),
    _class <- number_of_instances => _N, _N1 is _N-1,
    _class< number_of_instances := _N1,
    :
```

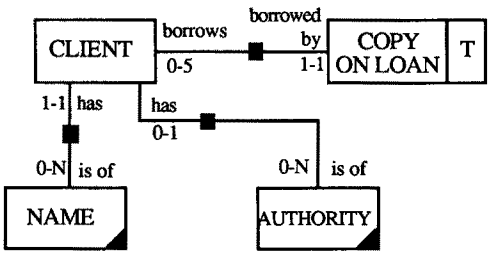
FIGURE 6.4

Involvement subset

Another way to represent an integrity rule is to translate the rule directly to a procedure which checks the consistency. These procedures are installed as `before-changed` deamons in the involved slots .The implementation of the 'involvement subset rule' below shows how that can be done (Fig.6.6). This sollution should be compared to the declarative in fig 6.3.

The involvement subset constraint specifies that the occurrences of one involvement form the subset of another involvement.

Example:



CLIENT that borrows COPY_ON_LOAN is a subset of CLIENT that has AUTHORITY

FIGURE 6.5

This rule is specified in the `before_changed` deamons in both of the involved relationships, stating that if a client makes a new suggestion he must be authorized and if the clients' authorization is removed he may not have any suggestions.

```
defclass client.
   supers entity
       :
   slots.
      has_authority,
         type instance_of(authority),
         cardinality(0,1),
         before_changed((not(it==undef);
                        (self<-suggests_new=>_sugg,
                     _sugg==undef))),

      suggests_new,
         type   instance_of(new_suggestion),
         cardinality(0,1),
         before_changed((it==undef;
                        (self<-has_authority=>_auth,
                     not(_auth==undef)))),
```

FIGURE 6.6

6.2 Dynamic Rules

6.2.1 Dynamic Integrity Rules

A dynamic integrity rule restricts the set of possible state transitions in database by specifying conditions for operations on the database. Since these rules concern the operations on the database, they are located within the methods performing these operations. The rules are expressed in TQL.

6.2.2 Dynamic Action Rules

A dynamic action rule specifies how the database may be transferred from one state to another. The rule has the form: [**When** trigger] [**If** precondition] **Then** actionpart. In TEMPORA a scheduling module based on temporal logic will be employed to control the evaluation of the rules, but here we will try to map the rules directly to Amore.

A dynamic action rule is implemented as a method consisting of two parts: precondition and action. In order to enhance readability and changeability the parts are separated into different method parts. The precondition is located in the before-part while the action is located in the primary-part of the method. The execution order of the methods ensures that the precondition is always evaluated before the execution of the action, and hence the action part is only invoked if the precondition succeeds. The rule is triggered by sending a message to the class holding the method, and therefore the message can be considered to be the trigger of the rule.

Each dynamic action rule is located in the class representing the business function it belongs to. The class object representing the business function also contains slots for transfering references to instances which have been retrieved by the before part, to the primary part. These slots are provided only for reasons of efficiency, in order to avoid search for instances in the action part of instances which have already been retrieved in the precondition.

Example:

WHEN	new_suggestion(AUTHOR.A, TITLE.T, EDITION NUMBER.EN, CLIENT.C)
IF	exists (BOOK.B that is written by AUTHOR.A and that has TITLE.T)
THEN	NEW EDITION <- create(B, EN, C)

This rule expresses that when a new suggestion arrives to the system, a new edition shall be created if the book already exists.

```
defclass suggestion_reception.                          /* Business function */
   supers abstract.
   private.
   slots.
      book,
         type instance_of(book),
      edition,
         type instance_of(edition).

   method before.                                       /* Precondition */
   new_suggestion(_author, _title, _edition_number, _client_name):-
      (instance_of(book,_b),
       _b<-written_by=>_a,  _a==_author,
       _b<-has_title=>_t, _t==_title,
       self<-book := _b).

   method primary.                                      /* Action */
   new_suggestion(_author, _title,_edition_number, _client_name):-
```

```
        self<-book => _b,
        client<-create(_c,_client_name),
        new_edition<-create(_n, _b, _edition_number, _c).
              :
endclass.
```

FIGURE 6.7

The chosen way to represent the dynamic action rules will result in a totally flat rule structure. Another possible structure is a rule hierarchy, where a rule may be described as a specialization of a more general one. If the rules were organized in a hierarchy, the inheritance rules of Amore would support specification of rules by specialization, if the action part was located in the after part instead of the primary part of the method. In this way, only the parts specific for a rule is specified, while the general parts are inherited. With this construction, first all parts of the precondition would be evaluated, starting with the most specific. Then all parts of the action would be executed, starting with the most general.

7 Comments

Here follows some experiences from the work of implementing the case study in Amore.

7.1 Mapping of ERT-schema

Translating the ERT-schema to class definitions in Amore is a fairly straight-forward process. The reason for this is that the ERT-model is basically object-oriented. The problems start when the temporal dimension is to be expressed. In the approach suggested in this paper, versionable objects are used for the representation of the historical information. Since the object model of Amore only considers one single state, versionable objects have to be simulated by mapping a versionable object class on two Amore classes. One class holding the current state and the other previous states. The ideal would of course be that this was handled by the object manager. In our case the lack of built-in versionable objects have not been a important problem, since all access to the outdated versions are by the TQL-interpreter, and therefore the existence of the object classes holding the history is not visible to the user.

7.2 Mapping of rules

Rules which have a built-in support in Amore, like cardinality constraints and type constraints, can be mapped directly to Amore. Those rules are evaluated by the Amore-system whenever it is necessary. But even other types of rules are easy to express declaratively in Amore thanks to the fact that Amore is based on Prolog. The possibility to express a general rule declaratively simplifies the mapping from the rule model to Amore very much. Declarative representaion is however not enough, it must be possible to invoke the rules for evaluation whenever there is a possibility that inconsistency has occurred. This is possible in Amore by attaching deamons (triggers) to slots.

Another strength of Prolog that has been taken advantage of is the meta-programming facility. This has made it possible to build a TQL-interpreter which makes it possible to express temporal queries using the temporal logic of TQL.

7.3 Combination of Object orientation - logic programming

It has been seen that this combination has advantages when we represent a specification containing ER-schema together with rules. But there are some problems with this combination. They are connected with the introduction of messages. Messages differ a

severely from ordinary Prolog predicates. A message can, like a predicate, succeed or fail. If it succeeds it can, however, only have single solution, even though its arguments can be bound in more than one way. This means that rules containing messages (which e.g. can be an access to a slot) can not always be strict declarative.

7.4 Problems with Amore

Some problems when implementing a prototype in Amore is
- Objects in the Amore system are not persistent
- Lack of transaction handling
- Slow execution if the number of object instances grows.

If the intention is to make a prototype which can be validated by using it in a realistic environment, these problems are serious. A possible way to improve the situation could be to build Amore on top of an object-oriented database manager. Preferably one which supports versionable objects, like the ones described in [Björn88, Kim88, Skarra87]. It perhaps also contain some deductive power.

Acknowledgements

I would like to thank Rolf Wohed at SYSLAB and SISU for valuable comments and suggestions during this work.

References

[AMORE] Z. Palaskas, "AMORE: Object Oriented Prolog Extensions Concepts, use, and implementation", E928/UMIST/ UMIST,

[Borg88] A. Borgida, "Modeling Class Hierarchies with Contradictions", in *"Proceedings of ACM SIGMOD"*, pp. 434-443, 1988.

[Bobrow77] D.G. Bobrow and T.Winograd "An Overview of KRL, A Knowledge Representation Language", in Cognitive Science1(1), ,pp 3-46, 1977

[Björn88] A. Björnerstedt and C. Hultén, "Version Control in an Object-Oriented Architecture", in *"Object-Oriented Concepts, Databases and Applications"*, Ed. W. Kim and F. Lochovsky, pp. Addison-Wesley, 1988.

[Kim88] W. Kim and H. Chou, "Versions of Schema for Object-Oriented Databases", *"14th Conference on Very Large Data Bases"*, Ed. pp. 148-159, Los Angeles, 1988.

[Kim89] W. Kim, E. Bertino and G. J. F., "Composite Objects Revisited", *"Proceedings of ACM SIGMOD"*, pp. 337-347, 1989.

[Moon86] D. A. Moon, "Object-Oriented Programming with Flavors", *"OOPSLA '86 Proceedings"*, Ed. pp. 1986.

[RUBRIC] "Rubric Implementation Manual", Esprit project E928, 1989.

[Skarra87] A. H. Skarra and S. B. Zdonik, "Type Evolution in an Object-Oriented Database", in *"Research Directions in Object-Oriented Programming"*, Ed. B. Shriver and P. Wegner, pp. 393-415, The MIT Press, Cambridge, Massachusetts, 1987.

[Stefik86] M. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, vol. no. pp. 40-62, 1986.

[TEMP] "TEMPORA Concepts Manual", Esprit project E2469, UMIST, Manchester, 1989.