TOOBIS

DELIVERABLE T23D1.1

PARIS 1
SORBONNE

University of Paris 1-Sorbonne

C. Souveyet, R. Deneckere, C. Rolland

Contents

1. Introduction

Time is everywhere in the information systems. Even if time is required for every application, the use and the need related to time are different. We identified five classes of time dependent applications :

Temporal database correspond to applications including data and business rules strongly dependent on time. Time is linear and discrete.
Temporal reasoning applications require reasoning on events (often fuzzy events) and their possible temporal ordering. Temporal reasoning is based on intervals and not on periods (where time is not anchored to time line). A particular use of temporal reasoning is Automatic Planning.
Branching time applications: classically time is considered in a linear manner, but some applications deal with activities done in parallel with different time and/or several calendars (versioning in design database, events in parallel systems etc.).
Real-time systems applications are focusing on the duration, the priority and the synchronization of tasks.
Time series management: time series are defined by a set of vectors for different time periods. A vector is a set of values for a time period . In general, time series are based on periodicity requiring a user-defined granularity or a user-defined calendar. This management requires special functionalities on the data management and time management.
Conventional databases are designed to store and retrieve current data. When update operations, data relative to evolving real situations, are erased and replaced by their new values. It means that past data are no longer in the database, only current data are kept. Except if the designer of the application has designed their historical storage.
The importance of time in databases has been revealed from 1975. From this date, applications required more and more to manage past, present and future data. This justifies to add to DBMS, mechanisms dedicated to model, retrieve, store and modify data dependent on time. This trend is confirmed by the introduction of SQL/T (temporal extension of SQL) in SQL3. This new generation of temporal DBMS integrates time into the data model in order to facilitate the management of temporal aspects in applications. Such DBMS have simplified the storage and the request of temporal data. But, they don’t solve design problems such as : definition of the data schema, specification of the activities of the application, verification of the coherence of the temporal database and also how to administrate such a database. The temporal extension of the OO method is focusing on these design problems relative to temporal databases.

The TOOBIS method is described in two fragments of methods. The first fragment deals with the basic OO method whereas the second fragment describes the temporal extension of the OO method.

The Unified Method Language should become the ODMG method for designing OO applications. An important issue for the TOOBIS method is its compatibility to UML.

The document is a long version of the Methodology deliverable. It is composed of 4 chapters :

- chapter 1 : a state of the art : it gives a synthesis of the comparison of six existing OO methods in terms of ways of modeling and ways of working.
- chapter 2 : the basic OO method (OOM) introduces the basic concepts of the TOOBIS method.
- chapter 3 : temporal extension (TOOM) introduces the customized concepts of the TOOBIS method to deal with the temporal aspects.
- chapter 4 : the Unified Method Language (UML) compliance. It shows the compatibility of the TOOBIS method with UML.

2. State of art : A Comparison of Six Object-Oriented Analysis Methods

Contents

2.1 Introduction

Object-oriented analysis and design (OOA&D) has raised exceptional interest both in practice and academics. However the state of art in OOA&D is extremely diffuse and it is becoming increasingly difficult to follow the developments in the field. Books introducing new methodologies [Rumbaugh91], [Jacobson93], [Martin92], [Shlaer88, 92] [Coad 90, 91] are being published at a tremendous speed and CASE tools and environments are being marketed with an accelerating pace. The methods are somewhat immature both theoretically and empirically. Their theoretical grounds are poorly documented, many of them just push upwards concepts of programming languages and combine object-oriented concepts with more traditional information systems and software engineering concepts such as E/R modeling and data flow modeling. Their practical experiments are very limited and most of them are subject to considerable evolution stimulated by both marketing considerations and practical experience. All this makes a reasonable comparison difficult.

There exists some comparative analyses of OOA&D methods [De Champeaux91], [Iivari94], [Goor92] which mainly use comparison criteria derived from the OO tradition. Even though this kind of internal evaluation highlights the differences and similarities between methods, it does not provide much information on the applicability of OOA&D methods as means of Information Systems development. In addition none of these comparative reviews do consider applications with a strong temporal dimension.

The purpose of this chapter is to analyze six OO methods with two main objectives in mind :

- identify the concepts and methodological rules that can be of very help for the development of temporal information systems
- study how fragments of various methods can be assembled to form a new method.

The analysis task has been driven by the requirements of the GlaxoWellcome and Delta applications. It leads to the proposal of a method engineering approach in which the TOOBIS method will be a *modular method* constructed by *assembling fragments of various methods*.

The analysis and comparison of the six methods is based on an analytical framework which is introduced in section 2. Section 3 presents the six methods using the framework. Section 4 discusses the results of the analysis.

2.2 The framework

There exist many different definitions of a method[Brikkemper90], [Lyytinen89], [Wynekoop93]. However most of them converge to the fact that every IS Development Method (ISDM) combines two aspects, namely the *product part* and the *process part*. Following [Olle91] we define an ISDM as composed of two parts : a *product model* and a *process model.* The product model is composed of a set of concepts and rules to manipulate them. The process model usually prescribes the steps to be followed and the activities to be performed in order to develop the information system. Therefore, a method provides the concepts to describe the product and the methodological guidelines to produce a quality product with a reasonable efficiency. As said in [Olle91], the product is the desired target of an IS development whereas the process is the route to be followed to reach the target.

Based on this assumption the framework employs two *reference models* : one to evaluate the product model of an ISDM and a second one to analyze its process model. These reference models are summarized in Table 1, Table 2 and Table 3. In the following, we introduce these two reference models in turn.

2.2.1 The product reference model

The product reference model is sum up in Table 1: Product reference model. It is based upon two assumptions :

- first, an information system is inherently a dynamic system including three aspects or perspectives in [Olle91] : structure (data), functions (process) and behavior (control).

- second, as a model of the Universe of Discourse (UoD) [Griethuysen82], an information system includes three levels :

- the model of the UoD, which defines the parts of the reality which have to be abstracted in the IS (we will refer to this level as the organizational level),

- the model for the information about the UoD i.e. the conceptual level,

- the model of the technical information system (we will refer to this level as the technical level).

Based on assumption 1, Table 1: Product reference model comprises **four columns** : *structure abstraction*, *function abstraction*, *behavior abstraction* and *intra level integration*.

The *structure abstraction* column captures the concepts proposed by the ISDM to model the data structure. *Object class* is an example of concept relevant as structure abstraction.

A *function* abstraction serves to model processes or operations. *Processes*, *operation types* and *methods* are examples of concepts that can belong to this column.

The *behavior* column contains concepts for modeling the system behavior i.e. how to control process execution. *Events*, *triggers* are examples of concepts used in the behavior column.

The *intra level integration* column serves to identifying (if existing) the concepts or constraints of the method which allow the integration of the three types of abstractions. *Encapsulation* in a class description of the data structure and the operations on that data structure is a means to ensure part of the integration between the structure abstractions and the function abstractions of a specific method. We will distinguish between formal integration and informal integration. Formal integration is assured through the definition of concepts; for example, defining the Class concept as including necessarily the definition of attributes and the one of operations acting on these attributes is a formal integration rule. It is the class definition which implies that operations cannot exist elsewhere than within a class. Vice-versa, acknowledging that a data store of the functional model may be equivalent to a class in the object model of the same OMT analysis schema is said to be informal.

Based on the second assumption , table 1 comprises **three raw** : the *analysis level raw* (sub-divided into *organizational level* and *conceptual level*), *design level* and the *inter levels* raw.

The organizational level raw captures the concepts provided by the ISDM product model to model organizational units such *actors*, *use cases*, *processes* and *flows*

The conceptual level raw identifies concepts to conceptually model the information about the UoD that has to be maintained by the IS. *Object class* and *event* are examples of concepts in this raw.

Concepts in the logical level raw are those which describe the technical information processing carried out by the IS. *Class, method and inheritance* are concepts which could belong to this raw.

The inter levels raw is justified by the importance of pre and post traceability [Gotel&Flinkestein93] [Ramesh92, 93]. Traceability is recognized nowadays as a key factor for handling change in the requirements and design of a system in an efficient manner. The raw aims to picking up the concepts (if any) that an ISDM method provides for pre and post product traceability. The link between an *actor class* at the analysis level and the corresponding C++ *class* at the design level is an example of inter levels concept.

Combining the four columns and the three raw leads to the product reference model presented in Table 1. The table comprises 12 cells that serve as the basis of presenting and comparing the product models of the six selected methods.

2.2.2 The process reference model

The process reference model is summarized in Table 2: Process modeling and Table 3: Process reference model. Table 2 aims at capturing the key features of the process modeling approach of the ISD method whereas Table 3 dives more deeply into the methodological guidelines of the method.

Obviously the process part of ISDMs is less formalized and less complete than the product part. As developed in [Rolland95] guidelines provided by methods are unsatisfactory. First, they are restricted to large-grained life-cycle descriptions most often treated as linear sequences, where crucial aspects of the process such as feedback loops and iteration are not represented [Boehm88], [Curtis88,92]. This emphasis incurs some risks of neglecting the influence of product structure on the development approach. For instance, the way the product is structured may have a strong influence on the steps needed for its construction. Furthermore, this linear view of process seems inadequate to model processes such as the requirements engineering process which are highly non deterministic in nature [Jarke94]. It is probably difficult, if not impossible, to write down a realistic state-transition diagram that adequately describes what has to happen during requirements engineering.

In addition they are based on informal means of process definition, and this has made them difficult to follow systematically.

Today, the process knowledge acquired during the development of a project cannot be easily capitalized to improve the current process model. Process models should help to materialize previous experiences in terms of knowledge

which may be reused to guide processes in new projects. Presently, reuse of process knowledge is only performed manually.

The production of systems is an evolutionary activity. It is clear that during this activity, not only do requirements and products change but also the process itself changes. In addition, it is possible for the process to change even while it is being executed. Accommodating this dynamic evolution of processes changes is a key but unresolved issue in process centered development environments. Process models should be designed to support this dynamic evolution of processes.

Both the ISE [Olle91] and the SE [Dowson94] communities acknowledge the importance of process guidance. However despite thirty years of information systems development methodologies [Olle91], [Loucopoulos92] the level of guidance provided to application engineers by methods and CASE tools is still low, especially in the early phases of information systems development. Methods are often incompletely and not clearly defined whereas a formal and detailed description based on notations having a well defined semantics is required to provide automated guidance during process performance. Consequently, current CASE tools are essentially passive; they help in capturing, storing manipulating and documenting

Table 1: Product reference model

		Structure abstraction	Function abstraction	Behavior abstraction	Intra Level Integration Support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	Actors Use cases	Input flows and flows States/stores Organizational functions	Events Triggering conditions	Operation (class) <-> activity (event) <-> process (flow)
	CONCEPTUAL LEVEL	Object class Attributes Association	Action/operation types IS functions	Events Triggering conditions	Event -> Operation -> Object
DESIGN / TECHNICAL LEVEL		Abstract class	Functions and procedures	Transactions Triggers	Class - trigger Procedure - input/output class
INTER LEVELS		Actor -> Object	Organizational functions -> Actions Actions -> Procedures	Events (organizational) -> Events (analysis) Events (analysis) -> ECA rules (design)	

products but offer very little to support the largely intellectual process of product construction. Most process centered software environments [Jacherri92],[Belkhatir94],[Bandinelli93], [Kaiser88] provide active guidance [Dowson93] which consists of strict enforcement of a process model. In fact, as indicated by [Tominaga94], process models are used to describe tools activity and to allow automatic invocation of tools. Process models are used to guide tools usage and not to support the application engineer in the performance of his/her intellectual activities. Sugiyama [Sugiyama94] nicely phrases the difference in referring to the latter as process scripts (the process models to support humans) whereas the former are called program scripts (the process models to guide tools).

Furthermore, recent in depth studies [Lubars93] demonstrate that developers know very little about the development processes they performed. All these explain the recent emphasis on development process modeling and support. Process engineering is considered today as a key issue by both the Software Engineering (SE) community and the Information Systems Engineering (ISE) community. Recent interest in process engineering is part of the shift of focus from the product to the process view of systems development. Process engineering is a rather new research area. However, there is already considerable evidence for believing that there shall be both, improved productivity of the software systems industry and improved systems quality, as a result of improved development processes [Dowson93], [Armenise93], [Jarke94].

This leads to new process models with more and more semantic power and modeling capabilities. Table 2a aims at capturing the process modeling approach selected by the ISDM method. We propose to rely upon the Dowson's classification of process models [Dowson88]. Dowson identifies three classes, namely *activity-oriented models*, product-oriented models and *decision-oriented models*.

Activity-oriented models come from an analogy with problem solving, i.e. finding and executing a plan of actions leading to the solution. They are sequential in nature. Examples of models of this class are the Waterfall model[Royce70], the Spiral model[Boehm88] and the Fountain model[Henderson90] Limitations of these activity-oriented models come from their representation of development processes like programs which do not integrate at all the interactive aspects of the IS development.

Product-oriented process models [Finkelstein90], [Franckson91], [Humphrey89] represent the development process through the evolution of the product. They promote a view of development processes which is still centered around the notion of development activity but present the advantage to link development activities to their output : the product.

Decision-oriented models are the more recent models [Potts89], [Jarke92], [Jarke93]. They put the emphasis on the decisional aspect of the process. The successive transformations of the product are consequences of decisions. Such models are semantically more powerful than the previous ones because they explain not only how the process proceeds but also why transformations happen. They provide a more complete knowledge about the process which is particularly useful for capturing design heuristics, reusing developed products and design decisions, and for dealing with backtracking issues

Most of ISD methods are activity based but it is clear that to overcome the limitations of current methods and provide an active process support during the development of applications in CASE environments, methods have to make a step forward to more formalized and more powerful process models. In Table 2 we aim at capturing

- the process modeling approach of the ISDM (column 1). This is based upon the Downson's classification;
- the process modeling formalism (formal language or set of precisely defined concepts or natural language only) (column 2);
- and, if the process model explicitly integrates the links between product and process or not (column 3)

Table 3 details the ISDM process model looking to the three following aspects:

- - the *steps* recommended by the method and their ordering. As most of the methods are activity based, steps are expressed in terms of activities such as *Define actors* or *Build scenarios.*

- the *heuristical knowledge* provided by the method. The OMT following guideline "A phenomenon of the real world is considered as a class if it has an OID, otherwise it is viewed as an attribute." is an example of such an empirical knowledge.

- the *intra level integration support* provided by the method. It can be rules enforcing the intra level integration concept definition (see Table 1) or checking rules which help managing the overlap of concepts associated to the different perspectives. Enforcing the encapsulation principle is an example of the first type of rule. Verifying in an OMT analysis schema that the union of operations of the Dynamic model and activities of the Functional model is equal to the operations of classes in the Object schema is an example of the second type of rule.

The raw of Table 3 are the same raw as defined in Table 1: the *analysis level* raw, *the design level* raw and the *inter levels* raw. *Cells* of the table are fill in up with examples of steps, heuristical guidelines and integration rules for each of the two modeling levels, namely analysis and design and for dealing with the rules supporting the inter levels mapping. This last set of rules explain how the method views and support the relationships between the two abstraction levels in the development process. There are two possible views : by transformation or by expansion. According to the transformational view the analysis specification is mapped onto a logical one taking technical considerations into account. With the expansion view, the analysis specification is completed to handle the technical aspects left aside at the analysis level. OMT proceeds by expansion whereas O* use a transformational view.

Table 2: Process modeling

	Process Modeling Approach	Process Modeling Formalism	Process - Product
Method	Activity oriented model	Natural language	Implicitly defined links

Table 3: Process reference model

		Step	Heuristics	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	Problem statements analysis Define actors Build scenarios and use cases	“An actor is a phenomenon of the real world who interacts with the IS by sending or receiving events”	“identify a use case implies to identify information modified or/and read by it”
	CONCEPTUAL LEVEL	Identify objects, classes, associations, events, operations, trigger conditions, states, object life cycle Build state transition graphs Build data flow diagrams	“A phenomenon of the real world is considered as a class if it has an OID, otherwise it is viewed as an attribute.”	“Each operation should be triggered by at least one event”
DESIGN / TECHNICAL LEVEL		Define methods, data types, constraints Build algorithms Identify abstract machines	"Mapping each subtype to a subclass produces technically correct coding but the result may not be efficient. Using flags instead of actual subclasses facilitates changes in classification"	“A class should have at least one method”
INTER LEVELS		Map an actor to an object Transform association-types	“An association-type is mapped into a valued attribute if one of its maximal cardinality is 1”.	

2.3 Presentation of the selected methods

2.3.1 The selected Object-Oriented Methods

The six OO methods selected for analysis and comparison are the following :
- OMT (Object Modeling Technique) [Rumbaugh91],
- OOSE (Object Oriented Software Engineering) [Jacobson93],
- OOA&D (Object Oriented Analysis and Design) [Martin92],
- OOAD (Object Oriented Analysis and Design) [Shlaer88], [Shlaer92],
- OOA/OOD (Object Oriented Analysis/ Object Oriented Design) [Coad90, 91],
- O* [Lee94], [Brunet93], [Rolland96]

The selection is based on the following three criteria :
- the method should cover the analysis level,
- the method should include a reasonably specific approach to OOA,
- the method should be accessible by books or/and research papers and documents,

2.3.2 Presentation of the selected methods

The analysis of the OOA methods introduced below takes place for each method successively in three parts. First, their basic concepts and notations are mapped on the structure of Table 1. Secondly the process modeling approach is evaluated in Table 2, and finally they are analyzed from the detailed process viewpoint by mapping their steps and rules to the framework of Table 3.

2.3.2.1 OOA&D [Martin92]

Table 1 is filled up in Figure 1 with some significant examples. In order to differentiate the schemes with the concepts included in them, the different diagrams of the method are shown underlined.

OOA&D is the only method that systematically addresses the organizational level in order to have a good understanding of the IS environment (some methods study this level but only with the aim of starting the modeling of the application). In OOA&D, the *Object Flow Diagram* gathers objects and activities. *Activities* produce or exchange objects. By dividing complicated processes into manageable chunks, OOA&D enables an understanding that is critical in complex situations. In top down analysis, Object Flow Diagrams are commonly decomposed to Object Flow Diagrams until enough is known to represent a process as an event schema. However, this is not necessarily the case. Object Flow Diagrams are one way of describing processes, event schemes are another. Therefore, an Object Flow Diagram can be decomposed into either or both descriptions.

Martin and Odell assumed that the product orientation leads to object orientation but it is not immediately obvious that the requirements that 'each activity should produce exactly one product and that each product should be produced by one activity' makes it object-oriented. Furthermore, there is the question of whether this requirement is natural. The relationship between products and objects seems quite intuitive.

The concept of object is defined in very general terms and covers a large range of concepts. That gives us a very fuzzy definition of object even if it seems to be mainly entity objects including values. Martin and Odell not model attributes as a separate construct in their Object Schema, but as relationships among objects. OOA&D expresses the life cycle of one particular object with the help of one or several *State Transition Diagrams*. In addition to showing

<table>
<tr><td colspan="2"></td><td>Structure abstraction</td><td>Function abstraction</td><td>Behavior abstraction</td><td>Intra Level Integration support</td></tr>
<tr><td rowspan="4">ANALYSIS LEVEL</td><td rowspan="2">ORGANIZATIONAL LEVEL</td><td></td><td><u>Object flow diagram</u>
Activity
Product</td><td rowspan="2"><u>Event schema</u>
Event
State
Triggering rule
Control condition</td><td rowspan="2">An event is a noteworthy state change of an object.
Each activity produces exactly one product, and each product is produced by exactly one activity.</td></tr>
<tr><td colspan="2"><u>Object flow diagram</u>
External object type
External activity / operation
Internal object type
Internal activity / operation</td></tr>
<tr><td rowspan="2">CONCEPTUAL LEVEL</td><td rowspan="2"><u>Object schema</u>
Object type
Relationship
Generalization link
Composition link</td><td></td><td><u>State transition diagram</u>
State
Transition</td><td rowspan="2">Event schemes represent the state changes that occur within the structure of object schemes.

Event schemes represent the behavioral aspects of an object. They specify processes by expressing procedural causes and effects in terms of the structural specification included in the object schema.</td></tr>
<tr><td colspan="2"><u>Event schema</u>
Internal Operation
External Operation
Event
Triggering rule
Control condition</td></tr>
<tr><td colspan="2">DESIGN / TECHNICAL LEVEL</td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="2">INTER LEVELS</td><td></td><td></td><td></td></tr>
</table>

Figure 1: Application of the Product reference model to OOA&D

the possible states of the objects, these diagrams also show the permissible state changes. Martin and Odell describe object life cycles as subtypes of the object type. Then, an Object Schema describes the states of an object life cycle as a specialization hierarchy of the object type. This nicely links Object Schemes and State Transition Diagrams, but at the same time makes object schemes complex.

Martin and Odell propose the representation of external object and external operation. This makes the diagrams to represent what goes on in a system or collection of systems and also which external objects or operations affect the system.

They propose a special notation, *Event Schema*, to model the functionality and behavior of the system. It is a formal approach based on the foundations of the Finate State Machine (hypothetical machine that can exist in only one of a finite number of states at any given moment). Event schemes incorporate structure expressed by Object Schemes. They have a relatively rich variety of concepts for modeling behavior (event, trigger, triggering rule and control condition) but their relationship to the Object Schema is not totally clear since the events of the Event Schema are not explicitly linked with the objects of the Object Schema. An event is defined as a noteworthy change in object state.

All classic object concepts (class, instance, association, multiple inheritance, composition, generalization...) are managed and the Event Model used is original according to other OO methods. It permits to emphasize the causal view of the dynamic, like in O*.

We can see in Figure 2 (where is presented Table 2) that the links between the process and the product are implicit. There is no formal language for expressing the process model and rules are stated in natural language. Nor exists a meta-schema of the process.

In order to be more explicit on the process, Table 3 is complete in Figure 3: Application of the Process reference model to OOA&D

Martin and Odell suggest that a useful OO analysis method should rest on a behavioral foundation. The structural approach is a visual metaphor that provides a static vision of how things are laid out in space, in contrast with the behavioral approach that describes what a thing does. Accordingly, they propose a specific approach for object behavior analysis and assume the object structure analysis to be embedded in it. Consequently, with only a few exceptions the steps emphasize behavior abstractions.

The first step of the process is to *'Identify the realm'*. This can be interpreted as taking place using object flow diagrams defining the activities and products of the enterprise. The products identified can also be interpreted as object types in the realm of interest. We need also to *'Identify the goal event type'* to provide an understood purpose to the selected realm. *'Clarify the goal event type'* will enable the analyst to define the underlying meaning of the chosen event type and provide its label.

The second step is to *'Generalize the goal event type'* in order to help the analyst to determine whether the analysis is being performed at the appropriate level of abstraction or scope.

The third step *'Define operations conditions'* permits to consider the operations and their triggering conditions.

The fourth step *'Identify the operations causes'* is used to determine when a control condition should be evaluated, so that the associated change in state can occur.

Step 1 through 4 guide the analyst in working with a given event, formalizing it, and determining what leads up to it. Before proceeding through the next cycle, a few refinements are useful and they are made in the fifth step *'Refine cycle results'*.

	Process Modeling Approach	Process Modeling Formalism	Process - Product
Martin and Odell (OOA&D)	Activity oriented process model	Natural language	Implicitly defined links

Figure 2: Process modeling of OOA&D

		Steps	Heuristics	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	0. Define analysis focus 0.a. Identify the realm	In order to model organizations in top down fashion at the strategic level, you will have to use an Object Flow Diagram .	An object flow diagram is useful to represent basic structure and when the dynamic is not yet comprehensible, otherwise, to express a process that can generate code, an event schema is appropriate.
	CONCEPTUAL LEVEL	0.b. Identify the goal event type 1. Clarify event type 1.a. Generalize event type and choose level 1.b. Integrate the event type 2. Generalize event type 2.a. Generalize event type and choose level 2.b. Integrate the event type 3. Define operation conditions 3.a. Identify the operation 3.b. Determine whether operation is internal or external 3.c. Identify control conditions 3.d. Normalize control conditions 4. Identify operation causes 4.a. Identify triggering event types 4.b. Indicate complex control conditions 4.c. Normalize triggering event types 4.d. Specify trigger rules 5. Refine cycle results 5.a Generalize triggering event type 5.b. Specialize goal event type 5.c. Check for duplicate events	By explicitly stating the kind of state change and the actual object types involved, the meaning of the event is rigorously expressed. The event type is expressed along with its pre and post states. By generalizing these, the event type is also generalized.	An operation causes event to happen. If you wish to express the possible states relative to an object class you will have to use a state transition diagram.
DESIGN / TECHNICAL LEVEL				
INTER LEVELS				

Figure 3: Application of the Process reference model to OOA&D

In the next step, the analyst takes the newly identified triggering event type, and continues again with step 1. As long as internal events exist, this cycle continue. It stops when only external operations remain and the starting event type has been reached.

The process is a step by step goal-driven analysis technique for constructing event schemes. It is a cyclic process that continue until reaching the starting event type. The process stays in the behavior abstraction for all steps and explores the structure abstraction only when it is necessary to continue the construction of the event schema. Martin and Odell intimately integrates object structure analysis techniques with the object behavior analysis techniques, but they do not say clearly when or how object types are identified and structured. They only give some hints that all the major steps include object identification and generalization. There are only few conceptual heuristics directly attached to the process, since it concerns only the event schema, so the process is highly intuitive and does not indicate at all when or how structures such as relationships and compositions are identified.

Finally, Martin and Odell seem to employ a transformational view. They do not discuss OOD in detail, but indicate a mapping from OOA terms to OOPL constructs (structures and behavior) or to relational DBMS.

2.3.2.2 OOAD [Shlaer88,92]

Table 1 is filled up in Figure 4 with some significant examples. In order to differentiate the schemes with the concepts included in them, the different diagrams of the method are shown underlined.

Shlaer and Mellor discuss OOA in terms of different models. A *Domain Chart* is produced for the entire system. A *Subsystem Communication Model* is produced for each domain in order to depict the event communication between the various subsystems within the domain. The majority of the work products of OOA are placed at the subsystem level: an *Information Model*, an *Object Communication Model*, an *Object Access Model* and supporting tables, descriptions, and lists are produced for each subsystem. Below the subsystem

lie the objects that make it up, a *State Model* is produced for each object and relationship having a relevant behavior. The actions of each State Model provide the next level; an Action Data Flow Diagram is produced for each state of each State Model. Finally, a *Process Description* is produced for each complex process of the action.

Three work products are developed for the Information Model of a subsystem: an *Information Structure Diagram* (graphic representation of the Information Model); an *Object and Attribute Descriptions Document* (lists each object and attribute in the model, provides an organized collection of the object, attribute, and domain descriptions); and a *Relationship Descriptions Document* (lists each relationship of the model with its relationship description). The Information Model is expressed using the relational model that does not support object identity. They do not adhere to encapsulation, but allows other objects to access the attributes of an object directly. Differing from the other methods, they do not have a concept for complex object. They don't have the concept of aggregation, nor that the one of abstract class. Each object that has a State Model is documented by providing: a State Transition Diagram; a *State Transition Table*; and a description of each action on the State Transition Diagram. For all the State Models, an *Event List* represents a listing of all events that have been defined. An event is defined as an abstraction of an incident or signal in the real world that tells us that something is moving to a new state which means that an event is triggered by the completion of an external or an internal operation. The dynamic abstraction of OOAD is similar to OMT but the concepts are less developed.

		Structure abstraction	Function abstraction	Behavior abstraction	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	Domain chart Object communication model Action data flow diagram -external entity			
	CONCEPTUAL LEVEL	Information model Object Subtype/Super type (inheritance) Association Subsystem Attribute	Process model Action data flow diagram Process Event Data flow Data store External entity Process description	State model State transition diagram Event State Transition Action Object communication model Object Event Object access model	A Data store in the process model is considered as an Object in the information model. An Event Data Flow in the Process Model is considered as an Event in the State Model.
DESIGN / TECHNICAL LEVEL		OODLE notation (object oriented design language) Class diagram Class structure chart Dependency diagram Inheritance diagram			
INTER LEVELS					

Figure 4: Application of the Product Modeling to OOAD

The work products of the process modeling phase are organized into documents: the *Object Access Model*, an *Action Data Flow Diagrams*, a *State Process Table*, and a *Process Descriptions Document*. The functional abstraction of OOAD also is similar to OMT but its Data Flow Diagram is used to specify the actions defined on the objects and not to specify a treatment using several objects.

The product of OOA&D is full of different models and documents (Domain Chart, Object Communication Model, Action Data Flow Diagram, Information Model, Process Model, State Model, Object Access Model...) which lead to increase the complexity of the analysis.

The concept of 'domain' in OOA includes an implicit categorization of objects. A domain "is a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain". Shlaer and Mellor furthermore suggest a more specific classification for domains: *application domains* (subject matter of the system from the perspective of the end user of the system), *service domains* (provides generic mechanisms and utility functions as required to support the application domain), *architectural domains* (provides generic mechanisms and structures for managing data and control for the system as a whole) and *implementation domains* (include programming languages, networks, operating systems, and common class libraries; provide the conceptual entities in which the entire system will be implemented), proposing that OOA typically covers application domains and the top level service that have application domains as their clients. Each domain, possibly decomposed in subsystems, is analyzed following three points of view: an Information Model (extended entity-relationship type) is realized for each subsystem; a State Model is realized for each object of the information model having a life cycle (even the passive objects, such account, can have a life cycle); and an Action Data Flow Diagram is realized for each activity linked to an object state.

Shlaer and Mellor present a transformation of OOA into an object oriented design. The design is expressed in terms of a single program. Each program is made up of a main program, four architectural classes and some number of application classes. It is represented in the OODLE notation. The obtained design is not the only possible one that can be derived but represents a basic synchronous design that can be extended, elaborated, and adjusted to accommodate size and performance constraints.

Figure 5 presents Table 3. The links between the process and the product are implicit. There is no formal language for expressing the process model and rules are stated in natural language. It doesn't exist a meta-schema of the process and product models.

In order to be more explicit on the process, Table 3 is complete in Figure 6: Application of the Process reference model to OOAD.

Shlaer and Mellor don't explicitly define the process of their method. The analyst has to discover it step by step. Consequently, the heuristics are strongly linked to the product. The method has a linear process that justly covers the global system behavior with the Domain Chart to the Information Model, and then covers the behavior abstraction with the State Model and finally the functional abstraction with the Process Model. Seeing that there is no encapsulation, the process doesn't need to come back to the structure abstraction to put the operations inside object like some other methods, OMT for example.

The first step in the OOA of Shlaer and Mellor is *'Domain analysis'*. The implicit categorization of objects into domains refers to structure, the mission statements of domains to function and the 'bridges' between domains to behavior. After domain analysis, they recommend constructing the OOA models for the application domain first.

The second step refers to the *'Initial definition of subsystems'* within domains. The initial, tentative assignment of objects to the subsystem corresponds to structure, and the descriptive of the responsibility of the subsystem to function.

	Process Modeling Approach	Process Modeling Formalism	Process - Product
Shlaer and Mellor (OOAD)	Activity oriented model	Natural language	Implicitly defined links

Figure 5: Process modeling of OOAD

		Steps	Heuristics	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	1. Domain analysis 1.a. identification of domains 1.b. identification of bridges 2. Definition of subsystems	If the domain is unmanageable (too large) the specification should be partitioned into subsystems.	
	CONCEPTUAL LEVEL	3. Construction of information models 3.a objects 3.b. attributes 3.c. relationships 3.d. subtypes and supertypes 4. Construction of state models 4.a. life cycles for objects 4.b. life cycles for relationships 4.c. object communication model 5. Construction of process model 5.a. action data flow diagram 5.b. object access model	Objects are identified by considering what the conceptual entities, or 'things' are in the problem being analyzed. An attribute must contain no internal structure. A single incident in the real world can be abstracted into more than one event.	If instances of an object are created or destroyed in run time, the life cycle should be formalized. If an object represents a specification, quality standard, or recipe, its life cycle probably does not need to be formalized. In order to represent the behavior of an object, we can draw a state transition diagram. A not atomic action can be detailed by an action data flow diagram.
DESIGN / TECHNICAL LEVEL		6. Construction of the OOD model 6.a. Produce Class Diagrams and Structure Charts		
INTER LEVELS		6.b. Transform application into architecture 6.c. Transform architecture into implementation	Each transition, even if it exists in different state models, is represented in one transition class. An object that does not have a state model is represented by a passive class in the OODLE.	

Figure 6: Application of the Process reference model to OOAD

The third step corresponds to the *'Development of Information Models'* to abstract the conceptual entities in the problem in terms of objects and attributes and to formalize the associations that pertain between the entities into relationships. The fourth step corresponds to the *'Development of State Models'* to formalize the life cycle of both objects and relationships. The actions identified and outlined in State Transition Diagrams are function abstractions.
The fifth step refers to the *'Development of Process Models'*, the major focus of which lies in functionality (the actions are dissected into fundamental and reusable processes). The Object Access and Subsystem Access Models specified in step 5 describe behavior.
Process modeling may also include the identification of users and external entities. Steps 6-9 repeat steps 2-5, applying them to the domain of the user interface which is one of the typical service domains. The other domains between the application domain and user interface may be analyzed depending on the hierarchy of domains in question.
The last step is to *''Construct the OOD model'* which permits to transform the object oriented analysis into an object oriented design. Shlaer and Mellor use a language independent graphical notation (OODLE: Object Oriented Design LanguagE) for depicting the design of an object oriented program.
The OOA covers the structure, function and behavior abstraction of the desired system. Although there is overlapping between the concepts of the different perspectives, the relationships between these abstractions are relatively clearly defined in the process, which is different from other OO methods, like OMT. The use of 'domains' and 'subsystems' can be expected to be useful in large projects but the modeling of the organizational context is weak. Moreover, the use of relational model information modeling is doubtful, even though the latter is largely consistent with the earlier conceptual information modeling tradition. The instructions are sometimes insufficient, concerning the domain integration or the derivation of object life cycles for example.

2.3.2.3 O* [Brunet93]

Table 1 is filled up in Figure 7 with some significant examples.
O* includes a conceptual model, a conceptual specification language and some abstraction mechanisms for supporting conceptual modeling. The method also includes a set of methodological guidelines for aiding in the modeling of real world systems by following the object oriented approach.
Others methods, like OMT of Rumbaugh or OOAD of Shlaer and Mellor have a product divided in three parts corresponding to three different models, one for the structure (Object Model), one for the dynamic (State Transition Graph), and one for the functional (Data flow Diagrams). On the contrary, O* uses only one model covering the three perspectives. The structural aspects of the IS are described in the *Global Structure Graph*, that provides the global structural perspective of the schema. The behavioral aspects of the IS are described in some *Global Behavior Graphs*, that provide the global behavioral perspective of the schema, through the hypothesis that the behavioral aspects are responsible for causing state changes to the structural aspects of the IS. The local structural aspects and local behavioral aspects of the objects in the individual object classes are described in *Local Class Views*. These views correspond to the local structural and behavioral perspectives of the objects defined.
In O*, there are three kinds of classes, namely Object Class, Message Class and Calendar Class. The declarations of each class define other related concepts such as properties/attributes in object class and messages class, and events in all kinds of classes. The Object Class describes and groups a set of objects of a same nature in terms of the structural and behavioral characteristics of the objects. The Messages Class define intentional description of the common characteristics of the objects. The Calendar class can be simulated at the system's internal clock (it describes time occurrence).

		Structure abstraction	Function abstraction	Behavior abstraction	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL				
	CONCEPTUAL LEVEL	Properties Domain Static constraint Message Class Calendar Class *Global structure graph* Composition link Reference link Inheritance link Object class	*Local class view* State transition diagram Operation Triggering factor Condition Dynamic constraint	*Global behavior view* Internal Event External Event Temporal Event Dynamic link	• Basic operations are encapsulated in object classes. • An event must invoke at least one operation but can invoke more than one operation. • An operation acts on one object only. • An operation is invoked by one or many events. • A state transition diagram is defined on a class. • An object can trigger one or more than one internal event.
DESIGN / TECHNICAL LEVEL					
INTER LEVELS					

Figure 7: Application of the Product reference Model to O*

The concepts of O* are very formally defined which leads to avoid any ambiguity about their signification.

The two basic concepts of O* model, namely Object and Object Class, are adopted from the object oriented paradigm. The main aim of the model is to adopt the concept of object from the analysis phase right through the implementation phase. Moreover, the model also takes into account the life cycles of objects for the determination of structural links between object classes, besides the inheritance links of the object oriented paradigm. Other than the inheritance link (based on the multi instanciation concept), two types of structural links can be distinguished: reference links and composition links, where each link can be defined between two object classes after identifying the dependency between the life cycles of objects in these classes. The concept of operation provides a means for locally describe the set of possible actions to be performed on objects. There are two kinds of operation that may be defined in an object class. Basic operations correspond to actions that change object states (they are triggered by events) and Service operations that merely provide access service to other objects.

An event occurrence is defined as a projection of a real world phenomenon whose occurrence at a given instant generates an impact on some objects of the IS. Simpler, the occurrence of an external event corresponds to the arrival of a valid message, the occurrence of an internal event corresponds to the state change of an object, while the occurrence of a temporal event corresponds to the verification of a temporal assertions.

O* has powerful facilities to model objects and constraints, as well the facility to check the completeness and correctness of the model. It provides an additional feature by integrating the enhancement facility such as the abstraction mechanisms for reducing the complexity of the IS in both its static and dynamic aspects. This method allows the formalization of static constraints (cardinality, value and inheritance constraints) and of dynamic constraints (State Transition Diagrams).

O* groups the advantages of object oriented and semantic models.

Table 2 is presented in Figure 8: Process modeling of O*.

The NATURE process modeling approach strongly couples the context of a decision to the decision itself. This approach aims at capturing not only activities performed during the RE process but also why these activities are performed (the decisions) and when (the decision contexts).

The central aspect of the process way of modeling is that it makes the notion of situation (in which to decide) explicit and relates it to the broader question of context handling. Figure 9 gives an overview of the process model, introducing its key concepts and their relationships (with a binary E/R based notation).

The central concept of this meta-model is the one of context. A *context* allows to describe a step of the process. In a situation the designer has to make a decision in order to progress in the RE process, therefore, a context is the coupling of a situation and a decision.

A *situation* is most often a part of the specification it makes sense to take a decision on. Situations are built from parts of the specification undergoing the RE process. A *decision* reflects a choice that a requirement engineer makes at a given point in time of the RE process. A decision encapsulates two aspects, namely *intention* and *approach*. An *intention* expresses what the engineer wants to achieve, it is a goal. An *approach* characterizes ways to fulfill an intention. Considering IS development, we currently recognize two different approaches, namely bottom up and top down. Following a bottom up approach means to define first the component and then the composite concepts, whereas following a top down approach means to first define composite elements and then their components. A *decision* is defined as an *intention* - to achieve - possibly following an *approach*.

	Process Modeling Approach	Process Modeling Formalism	Process - Product
J. Brunet (O*)	Decisional approach	NATURE process meta-model	Situation-Decision Action-Product

Figure 8: Process modeling of O*

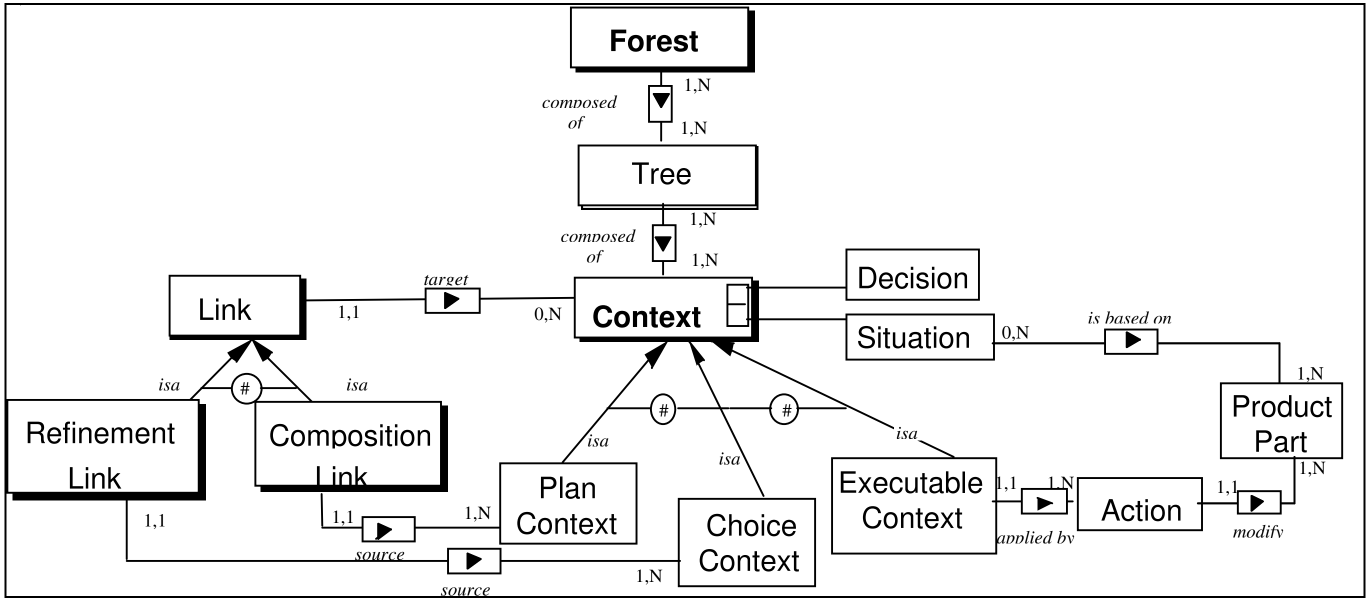


Figure 9: NATURE process meta model

There's three different types of contexts:

- the executive based context describe how to implement a decision, it allows to represent the change of the specification under development. Therefore, an executive based context is associated to an action.
- the choice based context allows to represent the various alternatives, it refines decisions. To help the selection process, *arguments* and *choice criteria* are introduced.
- the plan based context provides a means to decompose a context into several other contexts and to express the chaining among the composite context.

This meta model aims at offering a framework to precisely describe the RE process at various levels of details.

Table 3 is completed in Figure 11 which shows the O* process using the NATURE process meta model.

O* has a flexible process that permits you to begin with the wished perspective (structure, behavior, functional) and to develop the other ones only when it became necessary to the modeling of the application. The methodological process of O* is adopted from the fountain model. As a whole, the process steps serve as an aiding tool to the modeling process.

O* has a flexible process that permits you to begin with the wished perspective (structure, behavior, functional) and to develop the other ones only when it became necessary to the modeling of the application. The methodological process of O* is adopted from the fountain model. As a whole, the process steps serve as an aiding tool to the modeling process.

The process is decomposed in several chunks organized in a tree that permits you to have a flexible process.

The chunk *'Find element'* can be decomposed in two alternatives: '*Initiate discovery'* and *'Expand'*. The first chunk allows the analyst to find an element from a problem statement. It aim to find and differentiate real world phenomena that share the same structural and behavioral characteristics by classifying them into events, objects or actors. The second chunk allows to expand an element in order to discover other elements related to it. These elements can be events, objects, operations or actors. This chunk provides a huge flexibility to the method. In order to be more explicit, Figure 10 represents the tree concerning this chunk.

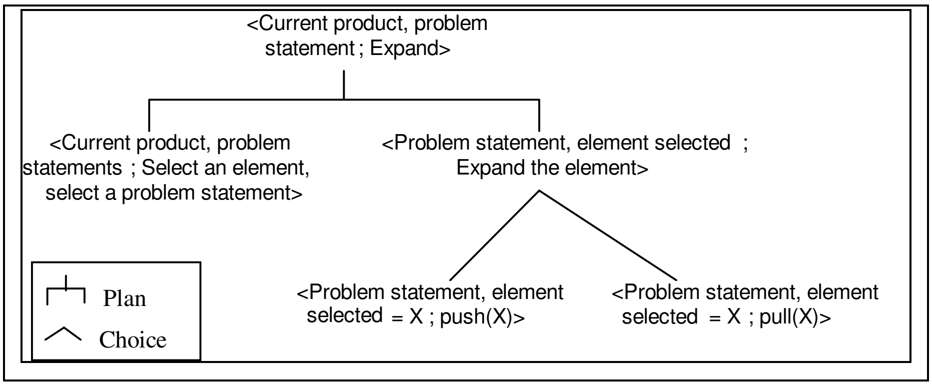
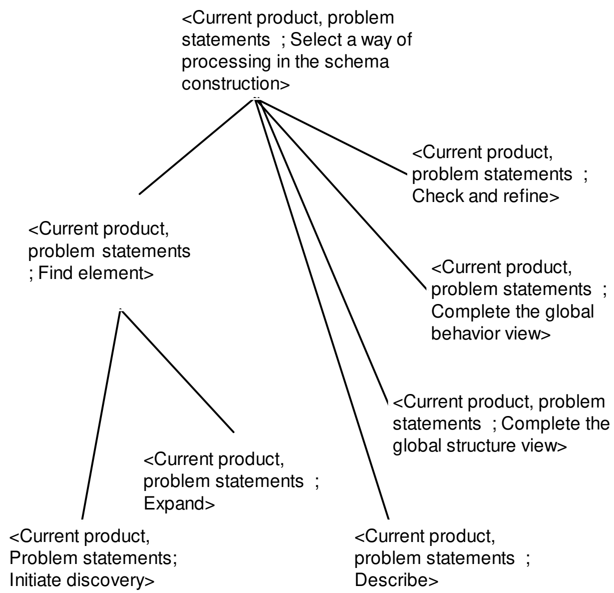


Figure 10: Part of the O* method decisional tree

This diagram shows you that if you have an element, you can expand it to find an other one. This new element will be related to the first one with a dynamic link. For example, if you have discover an event, you can apply this decisional tree to find the object/actor that triggered it (pull branch) or an operation invoked by it (push branch).

The chunk *'Describe element'* aims to obtain the local description of an element which can be an object, an actor, an event, an operation or a trigger. It makes a local study of the characteristics of the inventoried objects and the choices of the relevant characteristics. The acquisition of knowledge of the domain application is done in an informal way.

<table>
<tr><th colspan="2"></th><th>Steps</th><th>Heuristics</th><th>Intra Level Integration support</th></tr>
<tr><td rowspan="2">ANALYSIS LEVEL</td><td>ORGANIZATIONAL LEVEL</td><td></td><td></td><td></td></tr>
<tr><td>CONCEPTUAL LEVEL</td><td>General levels of the O* method decisional tree:

</td><td>If you have defined an event and that you want to work on a top down approach, you will have to define an operation. On the contrary, if you want to work on a bottom up approach, you will have to define an object.

If you have defined an operation and that you want to work on a top down approach, you will have to define an object. On the contrary, if you want to work on a bottom up approach, you will have to define an event.

If you have defined an object and that you want to work on a top down approach, you will have to define an event. On the contrary, if you want to work on a bottom up approach, you will have to define an operation.</td><td>If you are at the beginning of the application, you will have to start the process with the first branch of the tree: <…, find element> and <…, initiate discovery>.

If you have already an element in your current product, you can choose the branch <…, expend> in order to find other elements related to the one you have.

If you have more than two objects, you can choose to <…, complete the global structure view>.

If the problem statement is an adverbial phrase starting with a temporal adverb like "when" for example, that means that you can define an event.</td></tr>
<tr><td colspan="2">DESIGN / TECHNICAL LEVEL</td><td></td><td></td><td></td></tr>
<tr><td colspan="2">INTER LEVELS</td><td></td><td></td><td></td></tr>
</table>

Figure 11: Application of the Process reference model to O*

The chunk *'Complete the global structure view"* deals with the study of semantic links between the object classes by determining the possible composition links, reference links and inheritance links between the classes, with the aim of obtaining a global structure graph. By considering the classes globally, their semantic links are derived from the static interactions between the corresponding real world entities.
The chunk *'Complete the global behavior view'* aims to determine the behavioral interactions between objects by considering the classes globally in the behavioral point of view (they are analyzed via internal events and operations). This step emphasize events.
The chunk *'Checking and refine"* ensures that the conceptual schema is valid by checking its correctness with respect to its initial requirements formulated by the user.
These chunks allows us to view the model with three different points of view: one on the structure perspective (the global static graph), one on the structure-functional perspective (the local class view) and one on the structure-functional-behavior perspective (the global behavior graph). The object paradigm is used at each step of the process. The O* point of view is either local than global. The local view consider separately each class with its static and dynamic structures, the identification of the component classes and life cycles, the control of the consistency and the completeness between the static and the dynamic aspects. The global view gives us a general view on the relations and the communication between the different classes.

2.3.2.4 OMT [Rumbaugh91]

Table 1 is filled up in Figure 12 with some significant examples. In order to differentiate the schemes with the concepts included in them, the different diagrams of the method are shown underlined and the names of the models are bold.
The OMT methodology supports the entire software life cycle. It covers life cycle phases from problem formulation through requirement analysis, design, and implementation. The OMT methodology is based on the development of a three-part model of the system, which is then refined and optimized to constitute a design. The *object model* captures the objects in the system and their relationships. The *dynamic model* describes the reaction of objects in the system to events, and the interactions between objects. The *functional model* specifies the transformations of object values and constraints on these transformations.
The object model is an extension of the ER model [Chen76]. The dynamic model is an assembly of state diagrams of objects those behavior is important for the information system. The functional model is a slight modification of the data flow diagrams in SA.
The three models of OMT correspond directly to the structure, function and behavior perspectives. The set of concepts of one perspective is overlapping with concepts of another perspective. For instance, the object class concept of the structure perspective may be represented by a data store concept in the function perspective and the activity or the action concept of the behavior perspective may be represented by a process concept of the functional perspective. But the links between the three models are not explicit.Even though the role of the dynamic model is more obvious, the relationship between object types and state transition diagrams is not clearly documented. Obviously each object type may have 0 to N state transition diagrams, which may be clarified by identifying the possible states of each diagram as (special) attributes of the corresponding object class. The interaction between state transition diagrams is not clearly described and documented, but one can imagine two interaction mechanisms. State diagrams of the same object may communicate through a common condition that is a state of some attribute of the object in question (p .107). More generally, state diagrams of objects of the same type or different types may communicate through action/event mechanisms (an object can send an event to another object or set of objects, p.

		Structure abstraction	Function abstraction	Behavior abstraction	Intra Level Integration Support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	Actor		Scenario Event	For each event in the scenario it must exist an actor that caused that event.
	CONCEPTUAL LEVEL	Object Model Object class Attributes Generalization link Aggregation link Qualification Association link Module	Functional Model Data flow diagram Atomic Process Non atomic Process Data flow Data store Actor Constraints	Dynamic Model State transition diagram Event State Conditions Transition Operation: -Activity -Action Event trace Event flow diagram	Each event in the dynamic model corresponds to an operation in the object model. An atomic process in the functional model should be represented as an operation in the object model. A data store in the functional model should be represented as class or an attribute in the object model. A state transition diagram is defined for one class.
DESIGN / TECHNICAL LEVEL					
INTER LEVELS					

Figure 12: Application of the Product Reference Model to OMT

103). The functional model is not well-integrated with the previous two modeling perspectives. It can be constructed quite independently of the previous two (p.179). The role of data flows and data stores as objects is not entirely clear. According to Rumbaugh, data stores are ‘objects in the object model, or at least fragments of objects, such as attributes’ and data flows are ‘values in the object model’ and ‘other data flows represent normal objects’ (p. 138). The identification of operations of objects in OMT is fragmented in all three models. Operations are identified in object modeling, actions and activities in dynamic modeling and functions in functional modeling. Even though Rumbaugh, outline how to extract operations from the object models, state diagrams and data flow diagrams. For instance, ‘operations from object structure include reading and writing attribute values and association links’, ‘each event sent to an object corresponds to an operation on the object’, ‘actions and activities in the state diagram may be functions, these functions having interesting computational structure should be defined as operations on the object model’ and ‘each function in the data flow diagram corresponds to an operation an object (or possibly several objects)’ (p. 183 - 185). This identification process seems quite intuitive and pays only minimal attention to possible inconsistencies between the models and to their reconciliation.

The OMT method is useful for interactive systems and less interesting for databases based information systems because its dynamic model is insignificant for a purely static data repository, such as a database. The dynamic model of OMT is useful only for interactive systems.

Table 2 is presented in Figure 13 and, in order to be more explicit on the process, Table 3 is completed in Figure 14: Application of the Process Reference Model to OMT.

The analysis process in the OMT methodology is sequential in nature. The principal process steps correspond to the construction of three models : structural, dynamic and functional. Each step is an assembly of activities. It is possible to combine several activities or perform certain activities in parallel. Iteration of the activities is necessary at successively lower levels of abstraction, adding more details to the model. The links between these activities and their output (the product) are implicit.

The first analysis step in OMT is to generate a *‘problem statement’* (step 0). Rumbaugh, do not explain its content clearly, but based on their example, the problem statement is assumed to include some ideas related to the application concept. *‘Object modeling’* (step 1) is assumed to focus on the ‘Universe of Discourse’, ‘the identification of relevant object classes from the application domain’ (p. 153). The identification process of aggregation links is not clear. According to Rumbaugh et al, ‘aggregation is just an association with extra connotations’ (p. 156). The interaction scenarios, event traces and event flow diagrams identified in the *dynamic modeling* (step 2) model applications in terms of users and their interactions with the system. The emphasis on the user interface in the dynamic modeling is not totally clear. According to Rumbaugh, the purpose of scenarios is ‘to model typical dialogue between user and system to get a feel for expected system behavior'. These scenarios show the major interactions, external display formats, and information exchanges’ (p. 170). On other hand, they point out that ‘most interactions can be separated into two parts: application logic and the user interface. The analysis should concentrate first on information flow and control, rather than the presentation format. This may be interpreted as suggesting that the user interface is a secondary concern in analysis, a means of identifying the external events. The authors also take up the possibility of mocking up at least one user interface format (p. 172). Dynamic modeling is not limited to the user interface, of course, but every object that exhibits non-trivial behavior is modeled. Preparing typical interaction scenarios is also supposed to lead to the identification of information objects in addition to user interface objects, the primary emphasis lying on the behavior modeling. *‘Identifying input and output values’* (step 3.a) result in a description of the system known as a context schema in Structured Analysis. In addition to IS process/functions, one may identify additional data stores and data flows which are candidate objects when *‘building*

	Process Modeling Approach	Process Modeling Formalism	Process - Product
Rumbaught (OMT)	Activity oriented model	Natural language	Implicitly defined links

Figure 13: Process modeling of OMT

		Steps	Heuristics	Intra Level Integration Support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	O. Problem statement 2.a. Prepare scenarios of typical interaction sequences		
	CONCEPTUAL LEVEL	1. Object Modeling 1.a. Identify objects and classes 1.b. Prepare a data dictionary 1.c. Identify associations between objects (including aggregations) 1.d. Identify attributes of objects and links 1.e. Organize and simplify object classes using inheritance 1.f. Verify that access paths exist for likely queries 1.g. Iterate and refine the model 1.h. Group classes into modules 2. Dynamic Modeling 2.a. (see organizational level) 2.b. Identify events between objects 2.c. Prepare an event trace for scenarios 2.d. Build a state diagram 2.e. Match events between objects to verify consistency 3. Functional Modeling 3.a. Identify input and output values 3.b. Build data flow diagrams showing functional dependencies 3.c. Describe functions 3.d. Identify constraints 3.e. Specify optimization criteria 4. Adding operations 5. Iterating the Analysis	1. Classes often correspond to nouns. Associations often correspond to verbs or verb phrases. Aggregation is just an association with extra connotations. Attributes usually correspond to nouns followed by possessive phrases, such as "the color of car". Generalizing common aspects of existing classes into a super classes. Refining existing classes into specialized subclasses. 2. Examine the scenario to identify all external events. Every scenario or event trace corresponds to a path through the state diagram. Every event should have a sender and a receiver, occasionally the same object. 3. A process can be expanded into another data flow diagram. Each input and output of the process is an input and output of the new diagram.	Events can be represented as operations on the object model. A process in the functional model corresponds to an operation in the object model. A top-level process corresponds to an operation on a complex object, and lower-level processes correspond on more basic objects that are part of the complex object or that implement it. Data flows to or from actors represent operations on or by the objects. Data stores are also objects in the object model, or at least fragments of objects, such as attributes. Both actors and data stores are objects in object model.

DESIGN / TECHNICAL LEVEL	6. System design 6.a. Organize the system into subsystems 6.b. Identify concurrency inherent in the problem 6.c. Allocate subsystems to processors and tasks 6.d. Choose the basic strategy for implementing data stores in terms of data structures, files, and databases 6.e. Identify global resources and determine mechanisms for controlling access to them 6.f. Chose an approach to implementing software control 6.g. Consider boundary conditions 6.h. Establish trade-off priorities 7. Object design 7.a. Obtain operations for the object model from the other models 7.b. Design algorithms that implement operations 7.c. Optimize access paths to data 7.d. Implement software control by fleshing out the approach chosen during system design 7.e. Adjust class structure to increase inheritance 7.f. Design implementation of associations 7.g.Determine the exact representation of object attributes 7.h. Package classes and associations	7.a. Find an operation for each process in the functional model. Define an operation for each event in the dynamic model, depending on the implementation of control. The network of process within the data flow diagram represent the body of an operation. The flows in the data flow diagram are intermediate values in the operation. 7.b. Choose algorithms that minimize the cost of implementing operations. Select data structures appropriate as necessary. Assign responsibility for operations that are not clearly associated with a single class. 7.c. Add redundant associations to minimize access cost and maximize convenience. Rearrange the computation for greater efficiency. Save derived values to avoid re-computation of complicated expressions. 7.e.Abstract common behavior out of groups of classes. Use delegation to share behavior where inheritance is semantically invalid. 7.f Implement each association as a distinct object or by adding object-valued attributes to one or both classes in the association.	A process in the functional model corresponds to an operation in the object model. An event from the dynamic model may also become an operation on an object. The process in the data flow diagram constitute sub operations.
INTER LEVELS		Each operation from the analysis model must be assigned an algorithm. An association traversed in a single direction can be implemented as an attribute pointing to another object as a set of objects, depending on the multiplicity of the association. A bi-directional association can be implemented as a pair of pointers, but operations that update the association must always modify both directions of access. Associations can also be implemented as association objects. A qualified association with multiplicity ‘one’ can be implemented as a dictionary object. Qualified associations with multiplicity ‘many’ are rare, but they can be implemented ad a dictionary of sets of objects.	

Figure 14: Application of the Process Reference Model to OMT

data flow diagrams' (step 3.b). *'Describing functions'* (step 3.c.) naturally concerns functions, while the *constraints identified* in the functional modeling (step 3.d) are assumed to concern all object categories, covering both static and dynamic constraints alike.

The analysis process begin by the construction of the object model. When the object model is finished, the construction of the dynamic model can begin. The construction of functional model may follow the construction of the dynamic model or may be constructed in parallel with the dynamic model. Each step in the analysis process is composed of several activities, that require a number of iterations. When the tree models are finished, the object model is detailed by adding the operations to object classes.

Design follows analysis. The design process has two major steps: system design and object design. System design is the first design stage in which the basic approach to solve the problem selected. The output of system design is the high-level structure of the system called the system architecture. Building of the system architecture requires to accomplish the following activities. The first step in system design is to *divide the system into a small number of components* (step 6.a.). Each subsystem encompasses aspects of the system that shares the same common property - similar functionality, the same physical location, or execution on the same kind of hardware. One important goal of system design is to *identify which objects must be active concurrently and which objects have activity that is mutually exclusive* (step 6.b). Each concurrent *subsystem must be allocated to a hardware unit*, either a general purpose processor or a specialized functional unit (step 6.c). The decisions of *management of data stores* is another step in the system analysis (step 6.d) Rumbaugh characterize different kinds of data and describe the advantages and disadvantages of using a database for management of data. The system design must *identify global resources and determine mechanisms for controlling access to them* (step 6.e). Hardware control closely matches the analysis model where all interactions are shown as events between objects. System designer must *choose among several ways to implement control in software*(step 6.f). Most of system design is concerned with steady-state behavior but *boundary conditions must be considered* as well: initialization, termination, and failure (step 6.g). An essential aspect of system architecture is *making trade-offs* between time and space, hardware and software, simplicity and generality, and efficiency, and failure (step 6.h).

Object design follows analysis and system design. The object design phase adds implementation details, such as restructuring classes for the efficiency, internal data structures and algorithms to implementation operations, implementation of control, implementation of associations, and packing into physical modules. Object design extends the analysis model with specific implementation decisions and additional internal classes, attributes, associations, and operations.

The operations identified during analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations. But, according to Rumbaugh, 'the object model from analysis may not show operations' (p.229). Therefore, an analysis activity *'adding operations'*(step 4), may be considered as a first step in the object design (step 7.a). The problems concerning this activity are shown in the description of the analysis process. According to OMT authors, in making the conversion of the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model, 'we begin the process of mapping the logical structure of the analysis model into a physical organization of a program'. It is an initial design derived from analysis that must be extended and restructured for purposes and optimization. *Operations obtained in the first step are transformed in the algorithms that implement these operations* (step 7.b). Choosing algorithms involve choosing the data structures they work on. Therefore, the classes, attributes, and associations from analysis must be implemented as specific data structures. During the expansion of algorithms, new object classes must be introduced to store intermediate results during program execution and to avoid the need for recomputation. The next step concerns design *optimization* (step 7.c). Redundant associations must be added 'to minimize access cost and maximize convenience'. Algorithms are optimized for greater efficiency. The *control implementation* step (step 7.d) refines the strategy for implementing the state-event models presented in the dynamic model. The OMT methodology propose three approaches to implementing the dynamic model: 'procedure-driven system', 'event-driven system' and 'using concurrent tasks'. The *generalization/specialization structures are optimized* in the next step : creation of the abstract classes, choice between inheritance and delegation (step 7.e). The OMT methodology propose tree approaches for *implementation of associations*: 'implement as an attribute in one direction only', 'implement as attributes in both directions', 'implement as a distinct association object, independent of either class' (step 7.f).

During system and object design the three analysis models (object, dynamic and functional) are used to obtain the final design model, but the object model is the main framework around which the design is constructed. According to Rumbaugh, 'the information in the analysis object model must be present in the design. Usually the simplest and best approach is to carry the classes from analysis directly into design. Object design then becomes a process of adding details and making implementation decisions.' Rumbaught. affirm that the final design document is also composed of three models: detailed object model, detailed dynamic model and detailed functional model and the design model is an expansion of the analysis model. But in reality only the object model of the analysis stage is expanded in the design stage object model. The dynamic model is used to obtain operations for the object model and the system control present in the dynamic model is transformed into pseudo codes, or state machines, or inter-task calls in languages that support concurrency. In the functional model Rumbaugh use also a transformation process mode. For instance, data stores of the analysis functional model in the analysis step is transformed into data structures, files, and databases during system design. There are certain concepts in the design process that do not exist in the product model. Subsystems, data structures, files, and databases are examples of such concepts.

2.3.2.5 OOA [Coad90, Coad91]

Table 1 is filled up in Figure 15 with some significant examples.

Coad and Yourdon [Coad90] propose the Object-Oriented Analysis (OOA) method which is extended by the Object-Oriented Design (OOD) method. The method has a minimal set of concepts, and only one specific concept, namely the concept of subject. A subject is a kind of system view. Views are nevertheless assumed to be disjoint and they are not presumed to have any functionality.

The OOA method clearly expresses the structure and the function of the information system but it doesn't show clearly the behavioral perspective of IS. The object life history and the state-event-response are only strategies to identify the services of objects. They are not a product but only methodological means. The event and state concepts do not exist in the final model. Using message connection, one represents the behavior of the IS. According to Coad and Yourdon, 'each message connection represents a message being sent, as well as a response being received'. Message connections are used to accommodate service needs.' Human interaction with a system could be shown on the model with message arrows going from the user to each object and classification structure'. The message connections in the object diagram show only the existence of dynamic relationships between the message sender and the receiver, but it doesn't show what concrete receiver's service the sender needs. These details are specified in the sender's and receiver's service specifications. 'The needed processing is named in the sender's service specification, and is defined in the receiver's services specification'.

Coad and Yourdon make a distinction between 'problem domain' objects and 'implementation' objects, OOA being concerned with the former and OOD with the latter. They introduce the Human Computer Interaction (HCI) issues in

		Structure abstraction	Function abstraction	Behavior abstraction	Intra Level Integration support
ANALYSIS LEVEL	**ORGANIZATIONAL LEVEL**	User	Task scenarios		
	CONCEPTUAL LEVEL	Problem domain object Object class Generalization-Specification structure Whole-part structure Attribute Instance connection Subject	Service	Message connection	Object life history and State-Event-Response are the strategies to identify the services. For each instance connection exist a message connection.
DESIGN / TECHNICAL LEVEL		HCI object			
INTER LEVELS		Design model is an expansion of analysis model			

Figure 15: Application of the Product Reference Model to OOA

OOD and consequently regard HCI objects as implementation objects. These are included in Table 1 to indicate their inclusion in the total OO analysis and design method.
The distinction between OOA and OOD made by Coad and Yourdon is clearly based on object expansion, OOA being concerned with ‘problem domain’ objects, to which OOD adds various implementation objects, modifying the class structure accordingly.
Table 2 is presented in Figure 16 and, in order to be more explicit on the process, Table 3 is completed in Figure 17: Application of the Process Reference Model to OOA.

Coad and Yourdon point out that steps 1 - 5 in OOA (*‘Identifying objects’, ‘Identifying structures’, ‘ Identifying subjects’, ‘Defining attributes’ and ‘Defining services’*) are activities and not necessarily sequential steps, but emphasize that their ordering represents the most common overall approach (p. 25). *Object identification* is based on the study of the problem domain (step 1). *Structures identification* concerns the grouping of objects in structures. Two kinds of structures are proposed: classification structures or generalization/ specialization structures which represent respectively the whole-part structure or aggregation relationship (step 2) and the inheritance structure. *Subject identification* allows to group objects of the same application domain in one graphical representation (step 3). This decomposition makes the diagrams more readable. The *attributes definition* step (step 4) identifies the data and states of objects and the instance connections between objects. The services definition step (step 5) is composed of 4 activities: identifying the fundamental services (occur, calculate, monitor) for each object or classification structure, construct objects life histories and state-event-responses to identify additional services, and *identification of messages* that trigger these services (step 5). Therefore, steps 1 - 4 are highly structure-oriented, focusing on the ‘Universe of Discourse’ only, and step 5 is the only one to address functionality an behavior.
The OOA method proposed by Coad and Yourdon is useful especially for the modeling of the structural (object) perspective of the system. A behavioral perspective is studied only in the design model (OOD).
Steps 6 - 9 correspond to design : *‘Designing the problem domain component’* (step 6), *‘Designing the human interaction component’* (step 7), *‘ Designing the task management component’* (8), and *‘Designing the data management component’* (step 9). Coad and Yourdon emphasize that these activities are similarly not sequential steps (p. 26). Consequently, it is assumed that they proceed largely in parallel or without any predefined order. *‘Designing the human interaction component’* comprises a number of more detailed steps. The steps *‘Classify the humans’* (step 7a) and *‘Design the human and their task scenarios’* (step 7b) model the application concept. *‘Design the command hierarchy’* (step 7c) is assumed to imply some idea of the user interface objects, while *‘Design the detailed interaction’* (step 7d) primarily models the behavior of the user interface, and step *‘Design the HCI classes’* (step 7e) defines and structures user interface objects.
Coad and Yourdon propose very few methodological rules or heuristics for the analysis process. The concepts of the method are mainly defined through examples.

	Process Modeling Approach	Process Modeling Formalism	Process - Product
Coad and Yourdon (OOA)	Activity oriented model	Natural language	Implicitly defined links

Figure 16: Process modeling of OOA

		Steps	Heuristics	Intra Level Integration
ANALYSIS LEVEL	**ORGANIZATIONAL LEVEL**			
	CONCEPTUAL LEVEL	1. Identifying objects 2. Identifying structures 2.a. generalization-specialization 2.b. whole-part 3. Identifying subjects 4. Defining attributes 4.a. attributes 4.b. instance connections 5. Defining services 5.a. identify object states 5.b. identify the required service 5.c. identify message connections	Add a subject corresponding to each structure. Add a subject corresponding to each object. If a number of subjects exceeds 7 or so, then refine the subjects further. If an attribute applies to a majority of specialization, put it in the generalization; then, override it for the specialization that do not need it. Attributes with "not applicable" values imply another classification structure. Object with a single attribute imply that a higher level of abstraction is available. Add message connections for instances already connected by instance connections.	Each object has at least one message connection. Each classification structure has at least one message connection. Each classification structure level must have at least one attribute or service.
DESIGN / TECHNICAL LEVEL		6.Designing the problem domain component 7.Designing the human interaction component 7.a Classify the humans 7.b Design the human and their task scenarios 7.c Design the command hierarchy 7.d Design the detailed interaction 7.e Design the HCI classes 8.Designing the task management component 9.Designing the data management component		
INTER LEVELS				

Figure 17: Application of the Process Reference Model to OOA

2.3.2.6 OOSE [Jacobson93]

Table 1 is filled up in Figure 18 with some significant examples.

Jacobson [Jacobson93] , propose an object-oriented method called "OOSE" which cover the whole life cycle of the information system construction: requirements, analysis, design , implementation and test. The basic concepts in OOSE are Actors and Use Cases. An actor is specific role played by a system user, and represents a category of users that demonstrate similar behavior when using the system. ‘Primary actors’ define the categories of users directly using the system. Both , humans and other external systems or devices may be the users. An actor is regarded as a class, and users as instances of this class. A use case is a system usage scenario characteristic of a specific actor. Each execution of a use case may be considered as an instance of this use case. Jacobson group similar use cases into the use case classes. Each use case execution create an instance of use case class. A use case class is a description that describes the transactions of use case and is expressed in natural language with terms from the problem domain. An assembly of use case descriptions specify the whole functionality of the system. An actor has one or more use cases, which are specific ways of using the system. A use case defines the functionality inside the system, and is a specific way of using some parts of the functionality. Based on examples, it seems that use cases are structured according to the organizational functions (tasks/activities) that the actor performs. They also describe the information system acts, i.e. a special sequence of related transactions in the interaction between the actor and the system.

There are several problems with the interpretation of the actor and use case concepts, as defined in OOSE. No clear definition of the semantics of use cases, and no consistent guidelines on how the user should be described are provided. It is not clear what kind of events we should concentrate on while describing use cases; external stimuli-responses only, or internal system activities as well. OOSE treats use cases as classes with inheritance-like relations, but, at the same time, they are seen as sequences of events.

Each use case is associated with a specific actor, but at the same time, OOSE allows use case description in which several actors are involved. Moreover, an actor is defined as a specific role played by a user. The same person may be an instance of several different actors. This means that, in extreme, one physical user can appear as different actors in a single use case. These uncertainties leave too much room for tree interpretations of the actor and use case concepts, and may cause a lot of confusions. The number of use cases may be very large in cases of complex systems. Since produced independently, they might be inconsistent. Use cases may be contradictory, as they express goals of different actors. OOSE does not offer support for resolving such problems.

A distinctive feature of OOSE is the explicit distinction between entity objects, control objects and interface objects. The actor communicates with the system through the interface objects. The interface objects describe the two-directional communications between the system and its users. The entity objects are used for long time system information treatment. Control objects in OOSE may easily turn out to be function and behavior abstractions. Recognizing this danger, Jacobson, recommend that it is appropriate to allocate the functionality of use case first without using control object at all. One possibility for avoiding the above danger is to require that a control objects should take care of a particular information object type (e.g. an input transaction, a report, etc.). The semantics of Acquaintance association are also exceptional in the sense that it is assumed to be unidirectional. An acquaintance association describes a static relationship between two objects. The Consists of association is a specific type of acquaintance association that describes the situation when one object is composed of several other objects. A Communication association describes the communication (e.g. the dynamic relationship) between two objects.

		Structure abstraction	Function abstraction	Behavior abstraction	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	Actor Primary actor	Use	case	An actor has one or more use cases. Each use case is composed of an assembly of events invoked by an actor. For each suit of events, which are initiated by an actor, exists a use case.
	CONCEPTUAL LEVEL	Problem domain object Entity object Control object Interface object Attribute Acquaintance association Consists of Inheritance association Sub-system	Operation	Communication association Use case	An object can participate in several use cases. The communication between two objects is realized by a communication association.
DESIGN / TECHNICAL LEVEL		Bloc Attribute Acquaintance association Consists of Inheritance association Sub-system	Operation	Stimuli Interaction diagram State transition graph	An interaction diagram describe the communications between the blocs. An interaction between two blocs exist then one bloc sends a stimuli to another. The stimuli trigger the operations.
INTER LEVELS		Entity object, Control object, Interface object -> Bloc		Use case -> Interaction diagram	

Figure 18: Application of the product reference model to OOSE

The design model in OOSE have his own concepts. It is composed of blocs that are the abstractions of the veritable implementations. The communications between blocs are modeled as blocs sending stimuli to each other. These communications are described in the interaction diagrams. There is one interaction diagram for each use case defined in the requirement analysis phase. It describes the use cases in terms of communicating objects. Interaction diagrams support use cases with extensions. In this case, probe positions are added to an interaction diagram. A probe position indicates a position in the use case that is to be extended and often a condition is required for when the extension is allowed to take place. Finally, the state transition diagrams are used to describe behavior in terms of which stimuli can be received and what stimuli may cause. OOSE uses an extension of SDL notation (Specification and Description Language).

There is no clear definition of the event concept in OOSE. According to Jacobson, " new event gives rise a new operation. The events are the stimuli that are sent by one object to another and trigger the operations. "

Table 2 is presented in Figure 19 and Table 3 in Figure 20.

Jacobson do not propose an equally clear-cut process model, but distinguish five separate models : requirements model, analysis model, design model, implementation model and test model. The following review focuses on the first three of these. The latter two steps may proceed concurrently. The aspects of problem domain objects in parentheses may be identified in the requirements analysis, but Jacobson, recommend one to do so.

The analysis process of OOSE have two major phases: requirement model construction and analysis model construction. Requirements model delimits the system and defines its functionality. It consist of three parts: use case model, problem domain object model and interface description. The requirement process start *with the use case model construction* (step 1). The i*dentification of the actors* (step 1.1) and *use cases* (step 1.2) in the requirement analysis is assumed to precede the *identification of user interface descriptions objects* (step 2) and *problem domain objects* (step 3), even though use cases are expressed in terms of user interface descriptions and problem domain objects. These three steps can also take place in parallel without any definite order. Examples of use cases suggest that they include quite detailed sketches of use interface objects, interface operations and user interface acts. Jacobson, recommend that the *analysis of problem domain objects* (step 3) should include only the object name and possibly the attributes and static instances (i.e. acquaintance and consists of associations). Inheritance links, dynamic instance connections and objects operations (steps 3.d-f) may be identified in the analysis model. The collection of use cases is called the use case model. The use case model serves as a central model of which all other models are derived. It is the basis in the analysis, construction and testing.

The second model in the analysis process is the analysis model. The construction of analysis model begin by the identification of *interface objects* (step 4.a), *entity objects* (step 4.b) and *control objects* (step 4.c). These three kinds of objects are normally identified from use cases in an iterative manner when developing the analysis model. Referring to Jacobson, recommendation that it is appropriate to allocate the functionality of a use case first without using control objects, analysis of the control objects is assumed to follow the analysis of the former two.

Jacobson, emphasize that it is normal not to identify operations in the analysis model. Consequently, the objects operations are identified from use cases in the design model. Step 5, *Working with the analysis objects*, is assumed to include the allocations of functionality between the user interface objects, entity objects and control objects, even though Jacobson, discuss these allocation options in the context of corresponding object categories. They rely heavily on 'use case views' of the objects related to a specific use case to illustrate the distribution of functionality between objects. These 'use case views' describe the objects and their communication associations. Step 6 corresponds to *the identification of subsystems*, which are formed by grouping objects or lower-level subsystems. One means of forming subsystems is to describe their dependencies in terms of the associations (including

	Process Modeling Approach	Process Modeling Formalism	Process - Product
Jacobson (OOSE)	Activity oriented model	Natural language	Implicitly defined links

Figure 19: Process modeling of OOSE

		Step	Heuristics	Intra Level Integration support
ANALYSIS LEVEL	ORGANIZATIONAL LEVEL	I. Requirements analysis - requirements model 1.Use case model 1.1 actors 1.2 use cases 2. User interface description	The actors define the roles of the users of the system. If one (or one part of) use case must be affected by several actors, it is necessary to specify the use cases only for one actor.	The interactions between an actor and the system are specified by one or several use cases.
	CONCEP-TUAL LEVEL	3. Problem domain object model 3.a object name 3.b logical attributes 3.c static instance connections 3.d (inheritance 3.e dynamic instance connections 3.f operations) II. Analysis - analysis model 4. Identification of analysis objects 4.a. interface objects 4.b entity objects 4.c control objects 5. Working with the analysis objects 6. Identification of subsystems	A functionality of a use case that is directly dependent of system environment is transformed into an interface object. A functionality of a use case that concerns the treatment and storage of information is transformed into an entity object. A use case functionality that is specific to one or more use cases is transformed into several control objects. The operations that have to be out by an entity object may include: _ creating and deleting the entity object, _ storing and fetching information, _ behavior that must be changed if the entity object is changed.	The communication between two objects is realized by a communication association.
DESIGN / TECHNICAL LEVEL		III. Design - design model 7. Implementation environment identification 8. First approach to a design model 9. Description of object interactions - design model	An interaction diagram is drawn for each concrete use case.	The communications between blocs are modeled as blocs sending stimuli to each other and they are described in the interaction diagrams.
INTER LEVELS			Each use case is decomposed in objects of tree kinds. Each object of the analysis model is transformed into a bloc of the design model. Each use case of the analysis model is transformed in an interaction diagram of the design model.	

Figure 20: Application of the process reference model to OOSE

communication associations) between the objects in different subsystems. Jacobson, point out that the division into subsystem is normally made at the end of the analysis, but in larger projects it may be outlined before the analysis model is developed. The analysis model describes the structure of the system and is a step towards design.

The design model refine the analysis model and adapt it to the implementation. The first step in design phase is the *implementation environment identification*. This step includes the identification of the implementation environment consequences on the design, and the strategic decisions of the implementation: incorporation of the DBMS , choice of programming languages, choice of libraries of components etc. This step must be realized in parallel with the analysis phase and it must be ready before the system design phase. The veritable system design begins by *the development of first approach to a design model* (step7) when the analysis model is translated into the design model. The OOSE of Jacobson, adopts transformational view, in which initially every analysis object is transformed into a 'bloc'. The resultant bloc structure may be refined in different ways by adding new blocs, by possibly deleting existing ones or more (implementation) classes. The final step of system design affects the objects interactions and identify the objects interfaces (step 8). The result of this step is the design model that consists of interaction diagrams and state transition graphs. An interaction diagram is drawn for each concrete use case.

Use cases defined in the requirement model are reused in the other models. They are used to define the system objects, the objects operations, interaction diagrams in the design model.

2.4 Evaluation

This section is an evaluation of the six selected methods which is made by reference to the proposed framework. Based on the analysis of the strengths and weaknesses of methods we also make suggestions for the TOOBIS method. The section is organized in three parts : the two first are related, respectively, to the product and process models of methods; the third one summarizes the choices made for the TOOBIS method.

2.4.1 Analysis of the six product models

The comparison of the various concepts and notations shows that all methods include the three types of *abstractions* mentioned in the *product reference model* :namely, structure, function and behavior abstractions. However, the levels are unequally treated. OOSE, OMT and OOA&D integrate the organizational level whereas the design level is explicitly handled by OMT, OOA/OOD and OOSE. All methods integrate a conceptual level.
There is a consensus among the six methods on the *structure abstraction* but significant differences in the way of modeling the system functions and behavior. We comment these aspects in turn. Then, we review the topics of inter and intra levels integration.

2.4.1.1 The structure abstractions

Except for the class concept which originates from OO approaches, almost all the other notions are adaptations of the E/R model concepts. The notions of *class*, *aggregation*, *association* and *inheritance* are the preeminent concepts. They provide means to structure complex objects, classify them into hierarchies and relate one to another. However, aggregation to define complex objects is not supported by OOAD.
The terminology is not uniform, neither for the class concept nor for the relationship names. The class is named object (OOAD), object-type (OOA&D). Names respective to methods are reported in Figure 21: Synthesis of the structure abstraction.
A class is characterized by properties (also named attributes or functions); see Figure 21. The same occurs for the relationship names.
Although the concepts are defined in similar ways, there are differences. For instance, attributes cannot be associated to references in O* whereas they can be associated to OMT associations. A reference link between two object classes in O* is constrained : (a) instances of the referred object class are shared by instances of other object classes - (b) the life cycle of the referred class contains the life cycle of the referring object class. Similar constraints exist in O* for the composition link. The constraints provide a strict semantics to the reference and composition links. Such formal definitions do not exist in OMT for example and therefore, there is complete indetermination in the use of association and/or aggregation. The same situation can be modeled as an aggregation between objects or as an association. Similarly the use of different types of relationship links in OOA&D and OOAD is intuitive and subject to modeling errors which could be costly at the implementation and maintenance stages. Vice-versa in O*, the static links between object classes (reference, composition, inheritance) are strictly defined and guidelines provide advice to the analyst on how to proceed in the situation at hand.

methods	class	properties	inheritance	association	composition	strict definition
OOA&D	object-type	attribute function	generalization	relationship function	composition	no

OOA	object	attribute	subtype/super-type	association		no
O*	object class	attribute	inheritance	reference	composition	yes
OMT	object class	attribute	generalization	association	aggregation	no
OOA/OOD	object class	attribute	generalization specification structure	instance connection	Whole-part structure	no
OOSE	Entity object Control object Interface object	attribute	inheritance association	acquaintance association	consists-of	no

Figure 21: Synthesis of the structure abstraction

For the TOOBIS method, we choose the concepts of *object class, properties, association*, *composition* and *inheritance* with formal definitions which provide a strict semantics of these concepts.

2.4.1.2 The behavioural abstractions

Behavior modeling is tackled in three ways :

- the event driven approach (OOA&D and O*),
- the state transition approach (OMT, OOSE and OOAD),
- the message passing approach (OOA/OOD and OOSE).

In the first approach, the behavior of the application is described in terms of events and triggering rules. An event is a particular state change of objects which plays the role of a stimulus implying the system reaction. The reaction of the system to an event is expressed through triggering rules in terms of operations to perform on objects. This is a causal approach of dynamics in the sense of cybernetics and causal theory of organizations. It has numerous advantages among which are (a) the possibility of both a global and a local study of the system behavior and (b) a tight coupling of perspectives, the function and behavior perspectives in OOA&D and of the three perspectives in O*. The important principle of *minimality* (minimum set of concepts for a maximum representability) which is a well established principle in science is then, fully followed.

The second approach uses state-transition diagrams. A state transition diagram is only local to a class; it exhibits the possible states of an object in the class and the allowed transitions between these states. This approach lacks the representation of the global system behavior; this makes for example, the validation of the specification more difficult. In addition, the behavioral modeling is disconnected from structure and function modeling, which is bad.

The last approach is the weakest way of behavioral modeling. The expressing power of the corresponding techniques is very low. Message passing is a programming technique, not a conceptual modeling technique and the use of such mechanisms at the conceptual level is only justified by the fact that OO programming techniques have been pushed (sometimes erroneously) forward to design and analysis. They should not been accepted as conceptual. A message passing between an object A to object B does not explain *when* (in which situation) and *why* objects of classes A and B have to cooperate which is a necessary condition for conceptual modeling.

The classification of the six methods according to these three approaches is shown in Figure 22: Synthesis of the behavior abstraction. Note that, surprisingly, OOSE uses the message passing approach at the analysis level and state-transition modeling at the design level.

Another point of differentiation relates to the *intra-level integration aspect* (integration of the three abstractions). Two methods even belonging to the same behavioral modeling approach can offer very different solutions in term of integration. OMT and OOA are the worst in the sense that there is no formal rules of integration. Some guidelines are provided in the process but the formal correspondence among concepts is not provided. For example, an activity in the FM of OMT could be or not be an operation of a class in the OM. But this is not formally defined. In OOA&D

there is a tight coupling between the behavior and the function abstractions; the structure modeling is a part from the behavioral modeling while the three are fully integrated in O*. Therefore with a very limited set of concepts, O* subsumes all the other models which require more concepts with less modeling power.

methods	event driven	state-transition	message connection	intra level integration
OOA&D	X			function
OOA		X		no
O*	X			function/structure
OMT		X		no
OOA/OOD			X	function/structure
OOSE		X (design)	X (analysis)	function/structure (analysis) no (design)

Figure 22: Synthesis of the behavior abstraction

Behavior in temporal applications is an important aspect to study. This is why we propose to choose and event oriented approach and a strong integration of behavioral modeling with the two other perspectives.

2.4.1.3 The function abstractions

There are two trends :

- data-flow modeling (OMT and OOAD),
- encapsulation (OOA&D, O*, OOA/OOD and OOSE).

As shown in Figure 23: Synthesis of the function abstraction at the analysis level, the data-flow technique is used in OMT and OOAD for describing the system functions through channel agency diagrams. DF based modeling approaches explicitly recognize the encapsulation principle, which is a pity for so-called OO methods!!. However, OOAD recommends that operations found in the functional model should be integrated later in the class definitions while OMT only mentions that it could be done.

Following the encapsulation principle, functions called operations (or services) are defined in the class scheme. OOA&D is specific as function modeling is part of behavioral modeling. Therefore, operations are introduced in the event schema. Inputs and the outputs of an operation are defined in its signature.

Figure 23 shows the impact of the two modeling trends on the intra-level integration. Data flow modeling implies a separate and specific model for the function abstraction. Therefore, the integration with the other abstractions is only made *a posteriori*, at the end of the analysis process (see previous comments on OMT and OOAD). By essence the encapsulation principle forces to an integration of the function abstraction to the structure abstraction. This is made in OOA/OOD, O* and OOSE.

Methods	Data flow approach	integrated approach	intra-level integration
OOA&D		X	behavior
OOAD	X		no
O*		X	structure/behavior/function
OMT	X	(X)	-no -structure/function
OOA/OOD		X	structure/behavior/function

OOSE		X	structure/behavior/function

Figure 23: Synthesis of the function abstraction at the analysis level

In the method proposed for the TOOBIS project, we choose to follow the encapsulation principle which is part of the OO paradigm and also followed by O2. This will ease the mapping from the conceptual specification onto an O2 schema.

2.4.1.3.1 Intra-level integration

Integration between abstractions is an important issue and a key factor of success or failure of the practical modeling exercise. The way integration is taken into account in methods is therefore an important element of choice. The analysis of methods on this point is shown in Figure 24: Synthesis of the integration of the three abstractions at the analysis level. It is a pairwise comparison. We found out three different ways of integration :

- in one single model,
- in two different models with overlapping concepts,
- in two different models without overlapping concepts,

OOA/OOD, OOSE, O* use only one model at the analysis level. Obviously this has a number of advantages: it minimizes the number of concepts, makes explicit their relationships and then, facilitates their understanding , eases the validation of the specification because links between the three abstractions are done explicitly in the definition of concepts. At the other extreme OMT and OOAD require the use of three models which are redundant. A concept in one given model can overlap one or several concepts of another model. Overlapping concepts can have different names with no clear defined overlap. OOA&D has an atypical position with two models without overlapping concepts. For instance, the operation and the event concepts in the event schema are defined relatively to the object type concept but it doesn't appear in this model (it is an implicit link only).

methods	**number of models**	**structure- behavior**	**behavior - function**	**function - structure**
OOA&D	2	two different models, no overlapping concepts	one model	two different models, no overlapping concepts
OOAD	3	two different models, overlapping concepts,	two different models, overlapping concepts	two different models, overlapping concepts
O*	1	integrated in one model	integrated in one model	integrated in one model
OMT	3	two different models, overlapping concepts	two different models, overlapping concepts	-two different models, overlapping concepts - encapsulation of operation in object class
OOA/OOD	1	one model	one model	one model
OOSE	1	one model	one model	one model

Figure 24: Synthesis of the integration of the three abstractions at the analysis level

Integration of the three perspectives -structure, behavior and function- seems to show a number of significant advantages: minimality of the set of concepts, orthogonality of concepts, ease of understanding and use, better and quicker validation of specifications. We propose to follow this line in TOOBIS.

2.4.1.3.2 Inter-levels integration

We consider the three following questions :

- how analysis and design do relate?
- how is the organizational level taken into account ?
- is traceability of specifications ensured ?

(a) There are two possible ways of integrating analysis and design products : by *transformation* or by *expansion*.

The *transformation* mode assumes a different model at each level. The analysis product needs, therefore, to be transformed into a valid design representation. This mode is required when the two levels are based on different paradigms; for instance a conceptual OO schema and a relational database schema.

Conversely, the *expansion* mode is applicable when the two levels are based on the same modeling paradigm; for example the Object Oriented paradigm. Design decisions based on technical requirements are made to augment the conceptual product with design features which were not relevant at the conceptual level.

Considering the fully Object Orientation TOOBIS project, we propose the expansion mode.

(b) Three methods deal with organizational modeling : OMT, OOSE and OOA&D. Use Cases, scenarios and Object Flow diagrams are respectively used by OOSE, OMT and OOA&D to identify the relevant events and activities of the application.

The method of the TOOBIS project should propose the use of these kinds of methodological tools to initiate the analysis process.

(c) Traceability is recognized nowadays as a key factor of success for handling change in the requirements of a system in an efficient manner. Product traceability can be handled in an ISDM method in a rather simple manner; by keeping track of links between product elements belonging to different levels; for example between an *actor class* at the analysis level and the corresponding C++ *class* at the design level. Another approach consists in tracing the analysis and design process itself to record the design decisions and their rationale.

None of the studied OO methods proposes product traceability mechanisms. OOSE mentions the importance of traceability but no concrete concept is introduced. O* that promotes a decision-oriented process model for describing its way of working is supported by mechanisms for keeping track of all taken decisions and their rationale as well as their causes and their consequences in terms of product evolution. The tracing mechanism is a process tracing mechanism.

The TOOBIS method should, if possible, be accompanied by traceability mechanisms.

2.4.2 Analysis of the proposed process models

The evaluation of the process dimension of methods presented in this section uses the criteria of the framework summarized in Table 2 and Table 3. It is summarized in Figure 25 which is progressively detailed in the following. We comment in turn (1) the process modeling approaches, (2) the steps and heuristical knowledge and (3) the meta-process i.e. the process of constructing process models.

methods	paradigm	global organization for the analysis level	organization of heuristics	level of instructions
OOA&D	activity	behavior -> (structure, function)	no	poorly incomplete
OOAD	activity	structure -> behavior -> function	included in the model description	partly insufficient
O*	decision	"situational" approach	modeling approach	efficient guidance
OMT	activity	structure -> behavior -> function -> structure	specific chapter	partly insufficient
OOA/OOD	activity	structure -> (function -> behavior)	no	insufficient (function, behavior)
OOSE	activity	"use cases" approach	no	insufficient (function, behavior)

Figure 25: summary on the process part

2.4.2.1 Process modeling approaches

The analysis of Table 2 shows that almost all methods (all except O*) follow an *activity-oriented process modeling approach.*

Limitations of this modeling paradigm have been acknowledged and discussed in many papers. Despite its limitations, the use of this paradigm, is rather universal in classical methods. As said in the introduction of this document is it *de facto* the approach used, not by choice but more because of the lack of focus that methods put until recently on the process dimension. Things are changing and the focus is shifting from product modeling to process modeling. O* benefits from this shift and is associated to a *decision-oriented approach* which is semantically more complete and better adapted to the early stages of system development e.g. requirements engineering and system analysis.

An *activity-oriented approach* implies a linear view of processes which is not the case for analysis processes which are, by essence iterative. The process is prescribed as a plan of predefined and ordered actions i.e. a list of steps which, finally are never followed by the analysts.

Conversely a *decision-oriented* approach looks to conceptual modeling as a process of making decisions. It is more adapted to highly intellectual and creative activities for which analysts need to be guided and advised, locally to handle the particular situation they are faced to, and globally, to monitor the flow of decisions they have to make. The required support is based on heuristic knowledge, for example, suggestions on how to proceed in a certain context, providing alternatives for achieving a goal and advice to refine a goal.

Decision-oriented process models explain not only how the process proceeds but also when and why actions happen. They provide a more complete knowledge about the process which is particularly useful for capturing design heuristics, reusing developed products and design decisions, and for dealing with backtracking issues.

In the TOOBIS project, we would like to follow a decision-oriented approach based on our experience of the definition and use of the O* way-of-working. We feel necessary to provide to analysts and stakeholders interacting with analysts a well defined set of methodological guidelines which explain how to work with the proposed set of concepts.

2.4.2.2 Steps and heuristical knowledge

2.4.2.2.1 Steps

The detailed Table 2 of the previous section show that the process model of methods is restricted to a list of *ordered steps* which are informally described using natural language phrases. In addition these phrases express large grained activities. They constitute no more than a *road map* to proceed globally at each level. The *granularity* of the process steps is too large and descriptions are not detailed enough to provide an explicit guidance. Consequently the analyst has to discover by himself how to proceed in detail. Such descriptions are not accurate enough to provide guidance allowing, for instance, to choose a bottom-up or top-down design approach or to decide if an object class should be decomposed or subtyped. Vice-versa, the O* process model is able to handle different levels of granularity with an homogeneous way of describing detailed decisions and large grained decisions. In addition it offers more flexibility because there not one predefined order in which decisions have to be made but different possible ones. The exact order is chosen dynamically when the process proceeds.

2.4.2.2.2 Heuristical knowledge

Another important aspect of process models is the *heuristical knowledge* they can encapsulate. We found that this point was unequally treated by methods. Some of them provide detailed heuristics which explain in informal terms :

- when a concept should be used,
- if a real world phenomena should be modeled by a specific concept
- when a model should/must be used,
- how to check some properties of the specification
- empirical advice such as "a module cannot have more than 10 objects".

Heuristics are useful and supportive for the analysts especially for novices. However we found that their description in the five methods other than O* was first, spread over the books in various chapters and not necessarily in the chapters dealing with the process model descriptions. Second the heuristics are expressed in various forms : they can be examples only, natural language explanations, very often ambiguous or stories. Third, guidelines do not cover all aspects of the process. For example the Object Flow Diagram of OOA&D is not treated in the process part. Fourth, most often the guidelines are not globally consistent. We reached this conclusion when we gathered all heuristics in one single list and tried to organize the list to group together rules dealing with the same subject.

Current methods seem to be completely disorganized in their way of presenting methodological guidelines. We are in front of a kind of mess which does not really help the analysts who have to work with these methods. Finally, there is no possible automatic support of these methods on the process side. It is obvious that a software tool cannot guide a process with a disorganized list of informal, ambiguous and conflicting guidelines.

As a result of tackling process modeling in the similar way product modeling is tackled, O* uses the same precise formalism for describing global steps and detailed heuristics. In addition, these two aspects are related to each other by refining or decomposing higher decisions to more detailed ones. O* provides, in one hand, a process description guiding analysts from the beginning of their tasks until the end. In the other hand, a single process modeling formalism accommodates a wide range of process granularity in an homogeneous fashion. Moreover, based on the process model description, analysts can be supported guided by a CASE environment during the performance of the process.

In TOOBIS project, we intend to follow the O* approach in order to provide :
- a complete process model dealing with a wide range of process granularity in an homogeneous fashion ,
- a modular process description through the notion of *context* (intention to fulfill in a given situation)
- a complete and well organized set of heuristics,
- a process approach permitting to keep track of design rationales and to provide product traceability through process traces.

2.4.2.3 Meta-modeling

The last concern with process modeling is the manner to develop one specific process model i.e. meta-modeling. meta-modeling is the process of modeling process models. Traditional information systems methodologies were defined in an "ad-hoc" manner. This has a number of disadvantages among which the fact that the process model is depending of the experience of his designers.

The Information Systems Engineering (ISE) community has been recently paying attention to this problem adopting a *method engineering approach* [Lyytinen89], [Wynehoop93], [Brinkkemper90] in which the process model is considered as the product of an engineering process. The SOCRATES project [Hofstede90], [Wijers91] and the CAME (Computer Aided Methodology Engineering) approach [Martiin94] are early and promising works dealing with an automated support for creating situation specific process models.

The area of *Method Engineering* has emerged in response to an increasing feeling that methods are not well-suited [Lyytinen87] to the needs of their users, the application engineers. In particular, it is necessary to change methods from one business situation [Hidding94] to another. Situational method engineering [Kumar92] is the construction of methods which are tuned to specific situations of development projects. The situational method spectrum [Harmsen94] organizes approaches to situational method engineering according to their degree of flexibility in meeting situational needs and places them on a scale ranging from 'low' flexibility to 'high'. At the 'low' end of this spectrum are rigid methods whereas at the 'high' end is modular method construction [Harmsen94]. Rigid methods such as OMT, OOSE, and so on, are completely pre-defined and leave little scope for adapting them to the situation at end. In situational method engineering, a method is viewed as a collection of method fragments. A fragment can be a product fragment or a process fragment. In the former case, it captures product related knowledge of methods whereas in the latter case, it captures activity related knowledge. Method fragments are available in a method base to be assembled together to form a method.

We believe that such approach is interesting in the TOOBIS environment. A library of time dependent guidelines would be defined independently of any specific methodology. These reusable modules of temporal knowledge could be, then, integrated in various existing methodologies such as OMT or OOA&D.

A method engineering approach is our suggestion for the TOOBIS project.

2.5 Proposed features for the TOOBIS method

We sum up in this section the proposed features for the TOOBIS method that we have progressively introduced in this conclusive section. The summary is presented in two tables (Figure 26 and Figure 27) dealing with the product and the process aspects respectively.

characteristics	proposed features
structure abstraction	concepts of object class, properties, association, composition and inheritance links following the OMT terminology but with a formal semantics borrowed from O*.
behavior abstraction	event driven approach which seems semantically more powerful and particularly adapted to temporal applications.
function abstraction	encapsulation principle : functions are integrated in class definition
intra-level abstraction	one model single model for the analysis level. i.e. full integration of the three types of abstractions : structure, behavior and function abstractions.
inter-level abstraction process	expansion mode i.e the design specification results of the augmentation of the conceptual definition of classes
inter-level abstraction : traceability	the O* approach : the traceability mechanism is provided through process traces.
organizational level	a "scenario" based approach to initiate the analysis process

Figure 26: features of the TOOBIS method

characteristics	proposed features
process modeling approach	the decision-oriented approach borrowed from O* to provide a flexible process guidance advising on which decision to make in the situation at hand.
process model formalism	formal process description language allowing a modular description of the TOOBIS way-of-working.
heuristical knowledge granularity	the O* approach in order to provide : - a modular process description at different levels of detail - a complete and well organized set of heuristics, - traceability of design decisions

Figure 27: features of the process modeling approach

2.6 References

[Armenise93] : S. B. Armenise P., C. Ghezzi, A. Morzenti, *A survey and assessment of software process representation formalisms,* Int. Journal of Software Engineering and Knowledge Engineering, 3(3), 1993.

[Bandinelli93] : A. F. Bandinelli S., S. Grigoli, *Process Modelling in the large with SLANG*, Proc. of the 2nd Int. Conference on Software Process, (Berlin, Germany), pp. 75-93, 1993.

[Batini92] : S. C. Batini C., S. Navathe, *Conceptual Data Base Design : An ER Approach* , Benjamin/Cummings (Pub), 1992.

[Belkhatir94] : W. L. M. Belkhatir N., *Supporting Software Development Processes in Adele*,, Computer Journal, 37(7), pp. 621-628, 1994.

[Boehm88] : B. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 21(5), 1988.

[Brinkkemper90] : S. Brinkkemper, *Formalisation of Information Systems Modelling*, PhD Thesis, Thesis Publishers, Amsterdam, University of Nijmegen, 1990.

[Brunet93] : J. Brunet, *Analyse Conceptuelle orientée-objet*, PhD Thesis, University of Paris 6, 1993.

[Chen76] : P. P. S. Chen, *The Entity-Relationship Model : Towards an unified view of data*, ACM Transactions on Database Systems, 1(1), 1976 (March).

[Coad90] : Y. E. Coad P., *Object Oriented Analysis,* Prentice Hall, englewoods Cliffs, New Jersey, 1990.

[Coad91] : Y. E. Coad P. , *Object Oriented Design,* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Curtis88] : M. K. Curtis B., J. Over, *A Field Study of the Software Design Process for Large Systems*, Com. ACM, 31(11), 1988.

[Curtis92] : M. K. Curtis B., J. Over, *Process Modeling, Communications of ACM*, 35(9), pp. 75-90, 1992 (September).

[De-Champeaux91] : D. De-Champeaux, *A Comparative Study of Object Oriented Analysis Method*,, HP Laboratories, Research Report 1991 (April).

[Dowson88] : M. Dowson, *Iteration in the Software Process,* Proc 9th Int. Conf. on "Software Engineering", 1988.

[Dowson93] : M. Dowson, *Software Process Themes and Issues*, IEEE int. conf., 1993.

[Dowson94] : M. Dowson, *Towards Requirements for Enactment Mechanisms*, European worshop on Software Process Technology WSPT'94, pp. 90-106, 1994.

[Finkelstein90] : K. J. Finkelstein A., Goedicke M., *ViewPoint Oriented Software Development*, Proc. Int. Conf. on "Le Génie Logiciel et ses Applications", 1990.

[Franckson91] : C. P. Franckson M., *Specification of the Object and Process Modeling Language*, ESF Report n° D122-OPML-1.0, 1991.

[Goor92] : B. S. Goor G. van den, Hong S., *Formalization and comparison of six Object Oriented Analysis and Design Methods*, University of Twente,, Report Method Engineering Institute 1992.

[Gotel94] : A. C. W. F. Gotel O.C.Z., *An Analysis of the Requirements Traceability Problem*, IEEE transactions on software engineering journal, pp. 94-100, 1994 (May).

[Griethuysen82] : J. J. V. Griethuysen, *Concepts and Terminology for the concptual Schema and the Information Base*, ISO TC97/SC5/WG3, 1982.

[Harmsen94] : B. S. Harmsen F., Oei H., S*ituational method Engineering for Information System Project Approaches*, IFIP Int. Conf. on Methods and associated Tools for the Information Systems Life, 1994.

[Henderson-Sellers90] : J. M. E. Henderson-Sellers B., T*he Object-Oriented Systems Life Cycle, Communication of the ACM*, 1990 (September).

[Hidding94] : G. J. Hidding, *Methodology information: who uses it and why not ?*, WITS-94, (Vancouver, Canada), 1994.

[Hofstede90] : T. F. V. Hofstede A.H.M., G.M. Wijers , S. Brinkkemper, *The SOCRATES project*, proc. of NGCT'90, (Noordwijkerhout, The netherlands), pp. X1-1-X1-7, 1990 (April).

[Humphrey90] : W. S. Humphrey, *Managing the Software Process,* Addison-Wesley, Eds., 1990.

[Iivari94] : J. Iivari, *Object-oriented Information Systems Analysis: a comparaison of six object-oriented analysis methods*,in E. S. B.V., Eds. , , 1994.

[Jacherri92] : J. O. L. Jacherri L., R. Conradi, *Sotware Process Modelling and Evolution in EPOS*, Proc. of the 4th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE'92), (Capri, Italy), pp. 574-589, 1992.

[Jacobson93] : I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Le genie logiciel orienté-objet, une approche fondée sur les cas d'utilisation,* A.-W. France, Eds., (Bordas, 1993.

[Jarke92] : M. J. Jarke M., Schmidt J.W., Vassiliou Y., *DAIDA - An environment for evolving Information Systems*, ACM Trans. on Information Systems, vol. 10, 1992.

[Jarke94] : P. K. Jarke M., Rolland C., Schmitt J.R., *Experienced-Based Method Evaluation and Improvement : A process Modeling Approach*, Int. IFIP WG8.1 Conference in CRIS series : Method and associated Tools for the Information Systems Life Cycle, 1994.

[Jarke93] : e. a. Jarke M. , *Requirements Engineering : An Integrated view of representation, Process and Domain*, 4th European Software Engineering Conference, Springer Verlag, 1993.

[Kaiser88] : B. N. S. Kaiser G.E., Feiler P.H., Schwanke R.W., *Database support for knowledge-Based Engineering Environments*, IEEE Expert, 3(2), pp. 18-32, 1988.

[Kumar92] : W. R. J. Kumar K., Eds., *Methodology Engineering: A proposal for situation-specific Methodology Construction,* 1992.

[Lee94] : S. P. Lee, *Formalisation et aide outillée à la modelisation conceptuelle*, PhD Thesis, Paris 1, 1994.

[Loucopoulos92] : Z. R. Loucopoulos P., *Conceptual Modeling, Database and Case* Wiley, 1992.

[Lubars93] : P. C. Lubars M., Richter C., *A review of the state of the practice in Requirements Modeling*, Int. Symposium on Requirements Engineering, 1993.

[Lyytinen87] : K. Lyytinen, *Different Perpectives on Information Systems : problems and solutions*, ACM Computing surveys, 19(1), 1987.

[Lyytinen89] : S. K. Lyytinen K., Tahvainen V-P., Modeling CASE *Environments in systems Work*, CASE'89 conference papers, (Kista, Sweden), 1989.

[Martiin94] : P. Martiin, *Methodology Engineering in CASE shells : Design Issue and current Practice, Computer science and information systems reports* PhD thesis, Technical report TR-4, 1994.

[Martin92] : J. Martin, J. J. Odell, *Object-Oriented Analysis and Design ,* P.-H. I. Editions, Eds., 1992.

[Olle91] : H. J. Olle W., Macdonald I.G., Rolland C., Sol H.G. Assche F.J.M.V., Verrijn-Stuart A.A., *Information Systems Methodologies,* Addison-Wesley, 1991.

[Potts89] : C. Potts, *A Generic Model for Representing Design Methods*, 11th Int. Conf. on Software Engineering, 1989.

[Ramesh92] : D. V. Ramesh B., *Supporting Systems Development by Capturing Deliberations During Requirements Engineering*, IEEE transactions on Software Engineering, 18(6), pp. 498-510, 1992.

[Ramesh94] : E. M. Ramesh B., *Issues in the Development of a Requirements Traceability Model,* Int. Conf. on Requirements Engineering (ICRE), (Colorado Springs, USA), 1994.

[Rolland95] : S. C. Rolland C., Moreno M., *An approach for defining Ways of Working*, Information Systems Journal, 1995.

[Rolland96] : P. V. Rolland C., *Using Generic Method Chunks to Generate Process Models Fragments*, 3rd Int. Conf. on Requirements Engineering (ICRE'96), 1996.

[Royce70] : W. W. Royce, *Managing the Development of Large Software Systems*, IEEE WESCON 08/1970++, 1970.

[Rumbaugh91] : J. Rumbaugh, M. Blaha, W. Premerlani, F.Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* P.-H. I. Editions, Eds., 1991.

[Shlaer92] : S. Shlaer, S. J. Mellor, *Object lifecycles. Modeling the world in states ,* Y. Press, Eds., 1992.

[Shlaer88] : M. S. J. Shlaer S., Eds., *Object Oriented Systems Analysis ,*Yourdon Press, 1988.

[Sugiyama94] : Y. Sugiyama, *Producing and Managing Software Objects in Process Programming Environment OPM*, IEEE Asia-Pacific Software Engineering Conference (APSEC), pp. 268-277, 1994.

[Tominaga94] : T. T. Tominaga K., *Constraint-Centered Descriptions for Automated Tool Invocation*, IEEE Asia-Pacific Software Engineering Conference (APSEC), pp. 92-101, 1994.

[Wijers91] : G. M. Wijers, *Modeling Support in Information Systems Development*, PhD Thesis, PhD thesis, 1991.

[Wynekoop93] : R. N. Wynekoop J., *System Development Methodologies: Unanswered Questions and Research-Practice Gap*, 14th Int. Conf. Information Systems, (Orlando), pp. 181-190, 1993.

3. THE OBJECT ORIENTED METHODOLOGY (OOM)

Contents

3.1 Introduction

This document is the reference manual of the basic OO method. The temporal extension is defined in Chapter 4. The content of the OO method follows the guidelines induced by the State of Art : A comparison of six object-oriented analysis methods (Chapter 2).

If the reader is interested in the existing OO methods and the justifications of our choices in terms of concepts and model, he should read the State of Art first, otherwise he can start directly by this report.

The purpose of this document is to familiarize the reader with the modeling technique in general and more precisely with the concepts defined in the OO method. As it has been proposed in the State of Art, OOM aims to be a modular method constructed by assembling fragments of popular OO methods. It is based on one model integrating the three abstractions : structure, function and behavior. The concepts are presented through these three abstractions.

The document is structured as follows. *The modeling technique* , introduced in Section 2, gives to the reader a general understanding and terminology about the modeling activities. Section 3 presents the model and describes all the concepts of this method. Section 4 explains in details the concepts through the specification language associated to the model. Finally, two annexes are provided; one for the syntax of the specification language and the other one for a case study.

3.2 The modeling technique

A model is an abstraction of "the real world" for the purpose of understanding it before building it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity. To build complex systems, the developer must abstract different views of the system, build models using precise notations, verify that the model satisfy the requirements of the system.

• *Abstraction* is the selective examination of certain aspects of a problem. The goal of abstraction is to identify and describe aspects that are important for some purpose. Many abstractions of the same thing are possible, depending on the purpose for which they are made.

In [Olle91] an Information System Methodology has to deal with three orthogonal aspects or perspectives : structure (data), function (process) and behavior (control).

These three abstractions can be considered as three related but different viewpoints, each capturing important aspects of the system.

The *structure* abstraction has to describe the structure of objects managed by the system.

The *function* abstraction allows to define the processes or operations applied to objects of the system.

The *behavior* abstraction permits to explain the behavior of the system through the events from which it has to react.

These three viewpoints have to be strongly related in order to provide a complete description of the system. The relation between the structure and function abstractions describes the effects of the processes or operations on objects of the system. The link between the function and behavior abstractions permits to define the reaction of the system to an event, in terms of processes to trigger. Then, the link between the structure and the behavior abstractions allows to describe events in terms of state changes of objects.

Modeling a system requires to use the three basic concepts : objects, operations and events. Consequently, the designer has to abstract from the real phenomena of the real world, those representing objects, operations and events. The methodology we propose, follows the encapsulation principle, it means that the operations modifying an object and the events representing state changes of an object are encapsulated into the object definition.

Even if these three viewpoints are encapsulated, the designer can begin his design by any of these three perspectives.

3.2.1 Instances and Types

Observing the real world permits to identify real phenomena important to model. The real phenomena to capture are objects, events and operations. For example, the *Patient Smith* is an object , the *arrival of the patient Smith to the London Hospital* is an occurrence of a particular event, and the *update of the weight of the patient Smith to 75 kilos* is an occurrence of an operation.

Such descriptions belong to the instance level. But these instances can be classified into types of phenomena. Such types compose the model of the application. The *patient Smith* is an instance of the object class *PATIENT*, whereas the *arrival of the patient Smith to the London Hospital* is an instance of the event-type *ARRIVAL OF A PATIENT TO A HOSPITAL* and then the *change of the weight of the patient Smith to 75 kilos* is an instance of the operation-type *WEIGHT_CHANGE*. In this case, such descriptions belong to the type level.

The analyst has to describe his application in terms of types of phenomena. Sometimes, such abstraction appears difficult to make, observing the reality at the instance level and describing a model of this application at the type level. The methodology we propose permits to define the application in both instance and type levels and methodological guidelines are provided in order to transform the conceptual specification at the instance level to one at the type level.

3.2.2 The Decomposition mechanism

Each main concept of the model have a precise definition which is often difficult to follow at an early stage of the analysis process. As a consequence, the main concept's definition is relaxed to capture the first version of the system. For instance, an event is not the state change of only one object and an operation does not modify only one object. The observation of the real world made by the analyst is not constrained by the model. The decomposition mechanism allows the analyst to refine his first perception to a well-formed description.

Considering in the library system, the *LOAN_ACCEPTANCE* operation consists in creation of a loan, updating the state of the borrowed book and increasing the number of current loans of the subscriber. The refinement by decomposition allows to refine coarser operations, events and object classes to finer ones.

3.3 The Model

The main concepts of the model, that are used to formalize real world phenomena, together with their underlying relationships are shown in Figure 28: Main concepts of the OO Model.

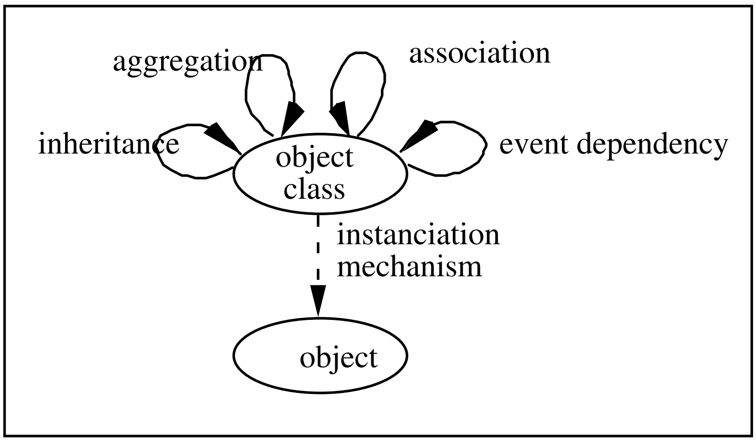


Figure 28: Main concepts of the OO Model

The two basic concepts of the model, namely object and object class, are adopted from the object-oriented paradigm. The main aim of the model is to adopt the concept of object from the analysis phase right through the implementation phase. Moreover, the model also takes into account the life cycles of objects for the determination of structural links (static links) between object classes, besides the inheritance links of the object-oriented paradigm. Other than the inheritance links, two types of structural links can be distinguished: associations and aggregations, where each link can be defined between two object classes after identifying the dependency between the life cycles of objects in these classes.

An object class is implicitly defined by an extent that stores objects representing the real world phenomena of the same nature. The extents of all the object classes form the persistent information of the IS. By default, an object of an object class refers to an object from the extent of the object class, which in turn, represents a persistent object of the IS. For specification purposes, the extent of object class is alternatively referred to as persistent collection, or more precisely, persistent set.

Other than the structural aspects, the model also considers the concepts of event and operation to take into account the behavioral aspects of ISs at the conceptual level. The causal dependency of events is related to the cause and effect in the occurrence of events. It can be explicated by the fact that the occurrence of an event is due to the previous occurrence(s) of some event(s).

Three groups of events are distinguished: external events, internal events and temporal events. External events correspond to the events occurring in the environment outside the IS. Internal events correspond to the internal state changes, or rather, the system responses to external changes. Temporal events that depend on the description of time, correspond to time occurrences. External events interact with the IS through messages initiated by an actor of an actor class. Each external event is attached to an actor class, while all temporal events are attached to the calendar

class. Internal events are attached to object classes. Each event is defined by an event predicate that indicates when the event occurs. The calendar class is simulated by the system's internal clock.
The concept of event is derived from the REMORA approach [Rolland82]. In this approach, propagation of state changes in the IS by basic operations is normally initiated by internal events as a result of the interception of an external event occurred in the environment outside the IS. Besides this, state changes can also be initiated by the occurrence of a temporal event. The concept of external event provides a more powerful way of communication than message passing mechanism by providing a mean for explaining when a certain phenomenon occurs.
Operations are defined in object classes. They are classified under basic operations and queries. A basic operation defined in an object class represents an action that will act on an object of this class, as a result, causing a state change to that object. The execution of basic operations is triggered by events. Queries merely provide access services to other objects. Derived attributes are realized as queries without input parameter.
Triggering conditions and/or triggering factors can be defined in the operation triggering of a trigger. They will be activated when the event occurs. A triggering condition is the condition that must be verified for activating a basic operation, while a triggering factor evaluates the set of objects on which the basic operation will be executed.

3.3.1 The structure abstraction

The structure abstraction allows to capture the static structure of a system as well as its environment.

The static structure of the system is described in terms of *objects which are considered internal*s. An object is an instance of an *object class* which is described in terms of *properties and constraints*. The definition of an object class can be done by deriving it from existing classes (inheritance mechanism).

The environment of the system is composed of external objects which have an impact on the system. External objects are classified in two groups : actors and clocks. Actors correspond to humans or devices interacting with the system through messages whereas clocks allow to wake up the system when relevant time instants occur. Actors are instances of an actor class whereas clocks are instances of a calendar class.

3.3.1.1 Objects Vs Classes

Objects represent real phenomena of the application. For instance, *the patient Smith, the GlaxoWellcome company, the truck number 4477 GB 75*. are objects.

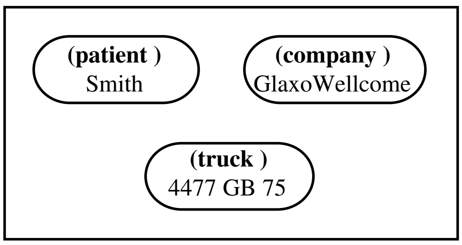


Figure 29: examples of objects

Object identity provides a means for differentiating an object from other objects in the same class. It implies two points :

• existence of an object is independent from the value that characterizes the object; and

• equality of two objects in terms of values does not mean equality of the objects.

The object identity is intrinsic to the object and it is insignificant to the user's point of view. Though the identity of an object is invariant across time, the object evolves in terms of its value with respect to time.

An object has its own identity. Each object can be differentiated from others, even if its attribute values are the same.

When we want to precise and refer exactly one thing, we will use the phrase *object*. We will use the phrase *class* to refer to a group of similar things.

The *classification mechanism* allows objects of the same structural and behavioral characteristics to be classified into object classes.

For instance, the objects *patient Martin* and *patient Smith* are classified into the object class PATIENT , which describes the structural and behavioral characteristics of potential patients.

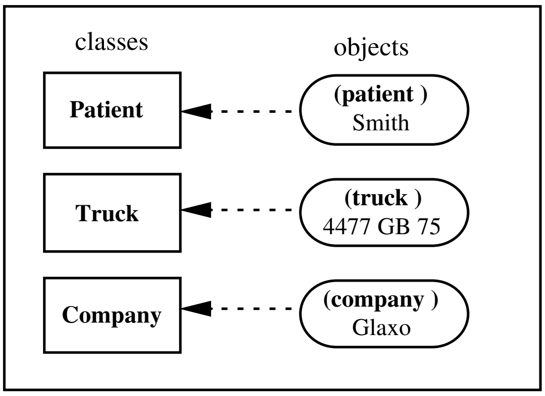


Figure 30: examples of instanciations

• Class is a way to group objects having similar properties (attributes) and common behavior (operations, events).

In the model, there are three kinds of classes, namely *object class*, *actor class* and the *calendar class*. The object classes represent relevant information managed by the system whereas actor and calendar classes compose the environment of the system.

3.3.1.2 *Object class*

An <u>*object class*</u> *describes and groups a set of objects of the same nature in terms of the structural and behavioral characteristics of the objects. It is identified by a unique name and has an extent consisting of the set of objects at a given time.*

An object class has a type that is defined by the description of the structural and behavioral characteristics of its objects. The structural characteristics are defined in the object perspective and the behavior characteristics are handled by the functional and the dynamic perspectives.

For instance *PATIENT, COMPANY, TRUCK* are classes and are represented graphically by a rectangle notation as:

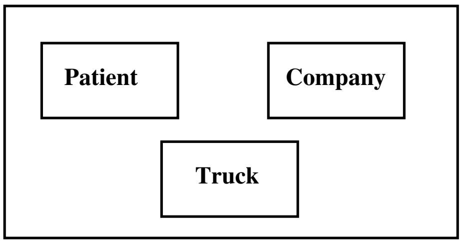


Figure 31: Examples of classes

3.3.1.3 *Actor class*

• *Object classes describe the internal structure of the system to model whereas actor classes represent the environment of the system to model. By modeling actors, we are interested to show the interaction between the system and its environment. The interaction existing between an actor and the system is described in terms of external events and external operations.*

This means that actors interact with the system by producing stimuli to the system in terms of messages. Their reception implies a reaction of the system which is described in the external event. In addition the system can interact with actors by triggering external operations.

The properties of an actor class are attributes useful to identify a particular actor within an actor class. For example, the class *CLIENT* can be viewed as object class but also as actor class because he interacts with the hospital system. In this case, the *Client* is represented in the application as an *actor class* as well as an *object class*. The *Client Smith* is an instance of the actor class *Client*. For instance, in the library system, the actor class *Client*, is represented graphically as follows:

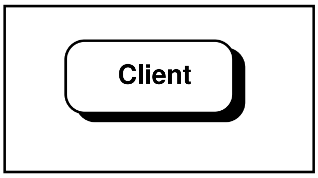


Figure 32: Example of an actor class

The description of an actor class consists of three parts:

- *attributes* are useful to identify an actor in the set of actors belonging to the same actor class. They are detailed in the section "*Properties*".
- *external events* describe how an actor which can affect the internal structure of the system. They are detailed in the section "*behavior abstraction*",
- *external operations* allow the system to send message to an actor. The set of external operations define the set of messages received by an actor. They are detailed in the section *"function abstraction"*.

3.3.1.4 Calendar class

The *calendar class* allows to represent clocks which affect the system because of temporal business rules that are performed when a specific *time instant* occurs.

Usually, there is only one calendar class called CLOCK which is related to the Gregorian calendar. All temporal events are attached to this pre-defined calendar class. The environment of the system is composed by one Gregorian clock used to trigger temporal events.

Relevant time occurrences may induce a temporal event occurrence which will propagate state changes to the objects in the IS. Figure 3.6 shows the graphical representation of the calendar class CLOCK,

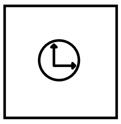


Figure 33: Example of the calendar class CLOCK

The description of the calendar class is composed by the list of temporal events having effects on the system.

Details about the temporal aspects are provided in the companion document "TOOM".

3.3.1.5 Properties

Properties are used to characterize a class. The property can be classified in two groups : *attributes* and *relationships* between objects.

An *attribute* is a characteristic of a class. The value set of an attribute is defined with the notion of *domain*

Relationships between objects are of two kinds : *association* and *aggregation*.

3.3.1.5.1 Attributes and domains

A class is characterized by *attributes* which are related to *domains*. Two categories of attributes can be defined : *object* attributes and *class attributes*.

3.3.1.5.1.1 Attributes

An ***attribute*** is a value characterizing objects of a class. *NAME, BIRTHDATE, WEIGHT and WEIGHTDATE* are attributes of the class *PATIENT*. Each attribute has a value for each object of a class. Attributes are defined in the class, and each attribute name can be followed by optional details such as the domain and the default value. The type is introduced by a colon and the default value by the equal sign. An attribute is regarded as a mapping function that gives for each object a value of a given value domain. For example, WEIGHT is a mapping function providing, for each patient , his weight.

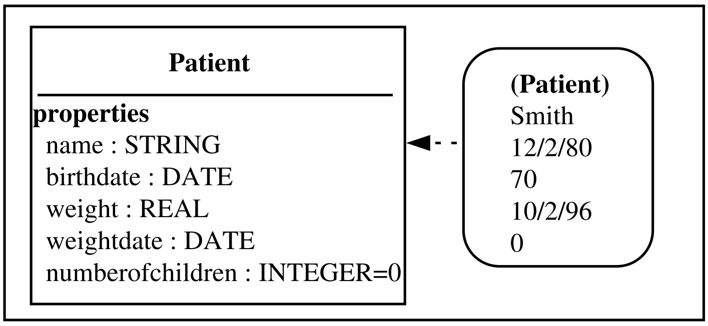


Figure 34: The instanciation of the attributes of the class PATIENT

An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have identity. For example, all occurrences of the integer 70 are indistinguishable, as are all occurrences of the string *Smith*.

Due to the fact that each object has its own identity, explicit object identifiers are not required. But notice that internal identifiers are purely an implementation convenience and have no meaning in the problem domain.

3.3.1.5.1.2 Domain

A ***domain*** is a data that contributes to the state of an object. It is described by a type that may be ***pre-defined*** *or* ***user-defined***.

The domain of an attribute may be a pre-defined data value, an enumerated value, a complex value, or other domain value, depending on the definition of the domain associated. A domain definition describes a static aspect of an object and contributes to the description of an information system. The concept of domain is not as important as the concept of class.

Integer, char, string, symbol, boolean are examples of pre-defined domain. But, the attribute *patient-state* can be defined in the following domain {"registered", "followed", "under_treatment", "dead", "lost"}. This domain is an enumerated domain which is user-defined.

Patient

properties
name : STRING
birthdate : DATE
weight : REAL
weightdate : DATE
patient_state : PATIENT-STATE
patient_address : ADDRESS

Figure 35: examples of domains in the object class PATIENT

The domain can be also ***simple*** or ***complex***. For example, integer , *state*, *degree*, *sea-miles* are examples of simple domains, whereas *address*, *position* and *periods* are examples of complex domains.

domain state : {"registered", "followed", "under_treatment", "dead"}

domain degree : [0..359]

domain sea-miles : **REAL**

domain position : [x : sea-miles, y : sea-miles]

domain address : [number : integer,
street : string,
city : string,
city code : string,
country : string]

domain periods : **set(**[period : [start : date, end : date]]**)**

3.3.1.5.1.3 Class attributes Vs object attributes

• If attribute has a value for each object of a class, it is called ***object attribute*** but when its value is common to all objects of a class, it is called ***class attribute.*** A class attribute is noted with a '$' mark.

Patient
properties Name : STRING birthdate : DATE weight : REAL weightdate : DATE patient_state : PATIENT-STATE patient_address : ADDRESS numberofpatients : INTEGER

Figure 36: Example of class attribute in the class PATIENT

For example, the attribute *numberofpatients* is a class attribute, it represents the number of patients existing in the extent of the *PATIENT* class. The attribute *name* is an object attribute because its value is different for each patient.

The class attributes can evolve during the life of the application.

3.3.1.5.2 Aggregation

• An ***aggregation*** is a specific association between classes. This link represents the composition of objects. It establishes a strong structural relationship between each object of the former class, referred to as composite object, and an object or several objects of the latter class, referred to as component object(s).

The aggregation link between two classes verifies the following properties :

- non-sharing : a component object cannot be shared by several composite objects;

- life cycle inclusion : the life-cycle of a component object is more or less included in the life cycle of its composite objects;

- life cycle similarity : at least one component object has the same life cycle duration as the composite object.

For instance in the library system, the fact that a subscriber has an address is represented by an aggregation link originated from the object class *SUBSCRIBER* and terminated in the object class *ADDRESS* as shown in Figure 3.15. The role *sub_address* precises which property takes its value into the set of objects of the class *ADDRESS*.

The aggregation link is expressed with a hollow diamond standing on the component side of the link, as represented on the following example.

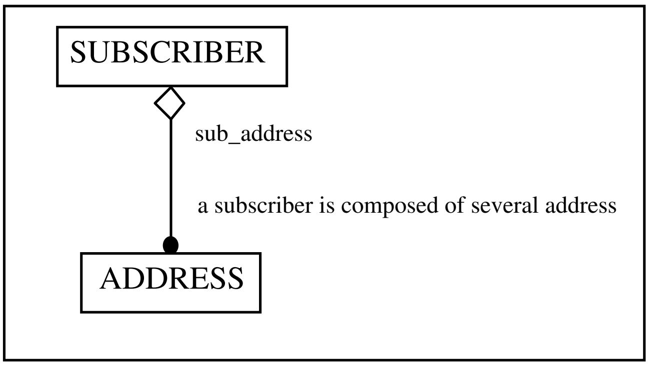


Figure 37: Aggregation link between the object classes SUBSCRIBER and ADDRESS

For instance, the life cycle duration of the object *"address of Martin"* corresponding to the address of the subscriber called *Martin* (where *Martin* is represented by the object *"subscriber Martin"* in *SUBSCRIBER*) can never be greater than the one of the object *"subscriber Martin"*.

The reasons for representing the address of a subscriber by an object instead of a domain are two fold. Firstly, the address of a subscriber is perceived to be a phenomenon of the real world that is relevant to the development of the desired IS. The second reason aims at relating the autonomy of the object representing the address so that its information content is independent of its composite object representing the subscriber. As such, the information content of the object will be managed by the object itself and not by its composite object.

Aggregation link can be associated with one of the following cardinalities : [0,1]; [1,1]; [0,N] and [1,N]. The cardinality constraint is graphically represented in the same way the association link is represented.

For instance, the fact that there are several copies of a book is represented as shown in Figure 38: Aggregation link between the object classes BOOK and COPY. The property copies represents the set of copies for a book. The cardinality mentions that a book is composed of zero or more copies.

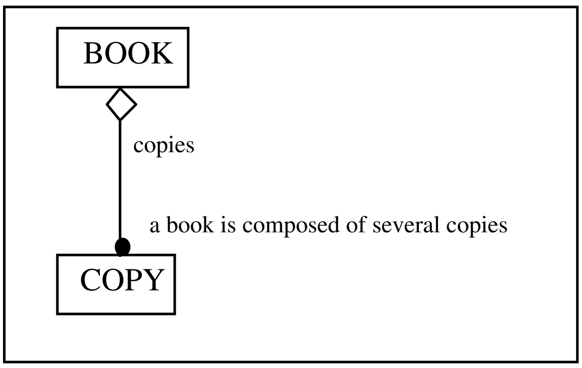


Figure 38: Aggregation link between the object classes BOOK and COPY

For instance, the object "book OMT" has at least one component object in COPY having the same life cycle duration as it due to its initial creation, together with either one or several of its copies. These ones are perceived to be phenomena of the real world that are relevant to the development of the desired IS.

Figure 39 shows graphically the different cardinalities allowed for an aggregation link.

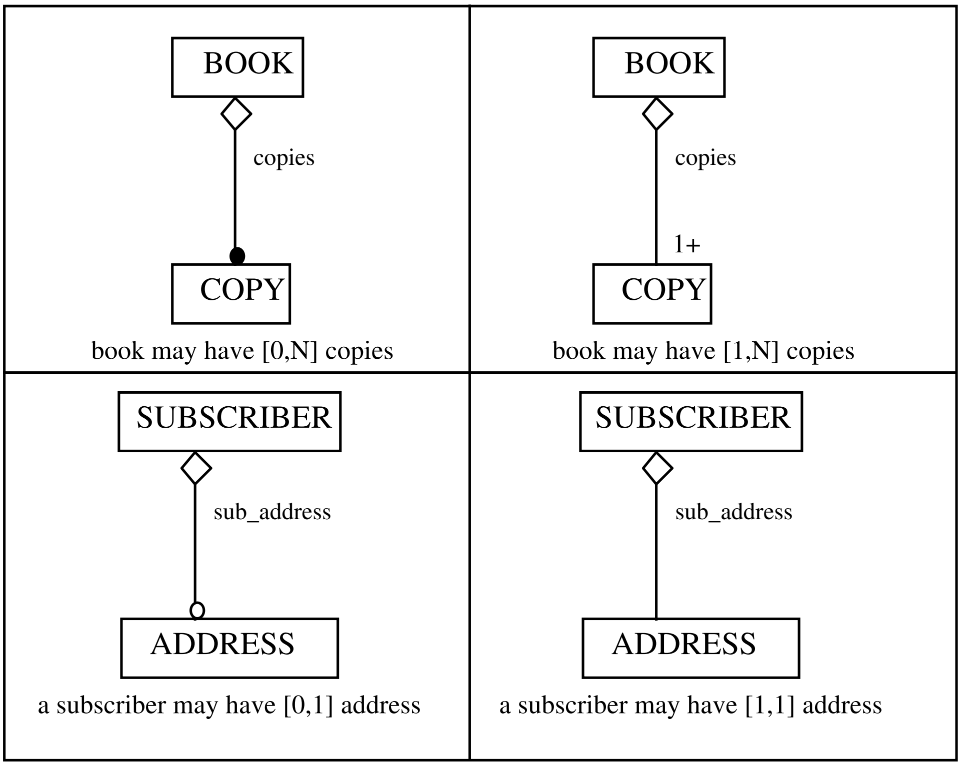


Figure 39: Cardinalities on aggregations

An aggregation link between two object classes is defined internally in the composite class by an attribute whose name corresponds to the name of the aggregation link. The inverse link from the component class to the composite class is managed implicitly by ***inverse of BOOK::copies*** from an object of the class COPY.

Semantically, the aggregation link expresses a strong coupling of behavior of the composite object with its component object(s). This coupling of behavior can be summarized as follows:

- the creation of a composite object implicitly implies the creation of at least one component object;
- the modification of a composite object may imply the creation, modification, or deletion of its component object(s);
- the deletion of a composite object implicitly implies the deletion of all its component objects.

These semantics are reflected in the signature of the composite object's basic operations.

3.3.1.5.3 Association

Associations allow to group relationships, between objects, which have the same structure and semantic. For instance, one object of the class *PATIENT is followed by* one object of the class *DOCTOR*. An association is considered as a mapping function that gives for each object of a class a set of objects of a given type. For example *IS FOLLOWED BY* is a mapping function providing, for each patient , the doctor who follows him.

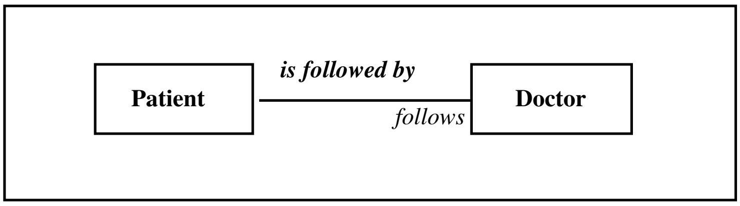


Figure 40: An example of association

Connections between classes are represented in terms of association. Relations are bi-directional in nature, even if at the implementation level one direction is not useful.

3.3.1.5.3.1 Properties of the association

We distinguish the principal direction (mandatory) and the inverse direction (optional).

For instance, the association between the classes *PATIENT* and *DOCTOR* describes the role *is followed by* as a principal direction of the association whereas the inverse role is*follows*. The referenced objects belongs to the class *DOCTOR* whereas the referring objects belongs to the class *PATIENT*.

The principal direction is described by a role in bold. The inverse role, when introduced, is in normal style.

An association has to verify the following properties :

- sharing : a referenced object can be shared by several referring objects;

- life cycle inclusion : the life cycle duration of a referring object is included in the life cycle duration of its referenced object;

- life cycle disparity : each referenced object has at least one referring object having an inferior life cycle duration.

Semantically, the association expresses a weak coupling of behavior of the referring object with its referenced object(s). The referenced object can be related to the several roles it plays, through different associations originated from some referring objects of different classes. For instance, an employee may play several roles related to his involvement in a project. Compared to the aggregation , the association has a trivial coupling of behavior between objects. It can be summarized as follows:

- the creation of a referring object may imply its binding to one or several referenced objects according to the type of association (simple or multiple);

- the deletion of a referenced object is allowed only when there isn't any referring link referring to it.

Note that it is possible to create a referring object initially without referring it to any referenced object. The binding can be done after the creation of the referring object. For instance, a new project is first created in an organizational IS without assigning it to any employee.

In order to ensure consistency, by default, the deletion of a referenced object is not allowed while there exists a referring link referring to it. This can be reasoned out simply through the library system, such that deleting a copy of book while there is still reservation of it will lead to inconsistency in the IS. However, if the reservation refers to a copy which is not referred by any other objects, then in this case, it is possible to delete the reservation together with the copy.

Cardinalities can be defined on associations. They are used to express how many instances of a class are connected to one instance of the associated class at a given time.

For instance, a patient is followed by only one doctor but a doctor follows up many patients.

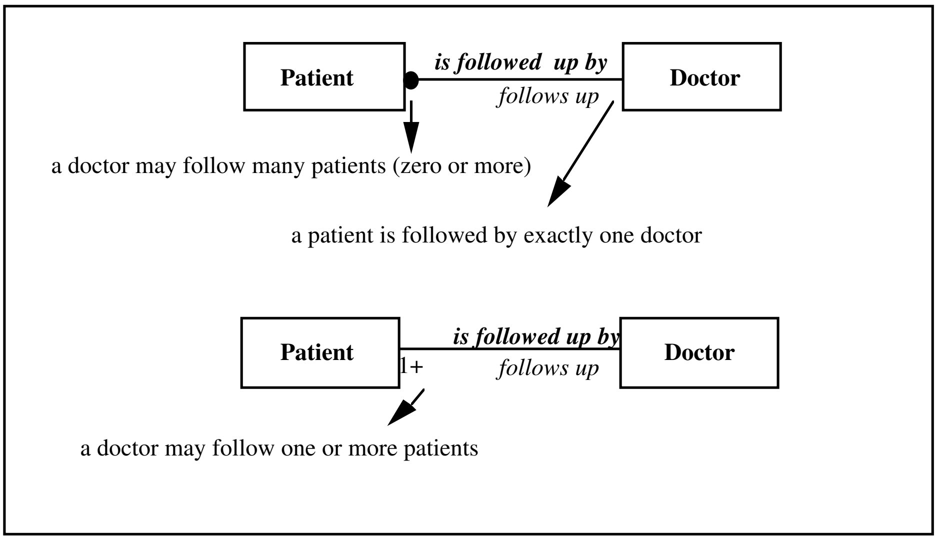


Figure 41: Cardinalities on associations

The connection may be optional. For instance, a doctor may not work for an hospital.

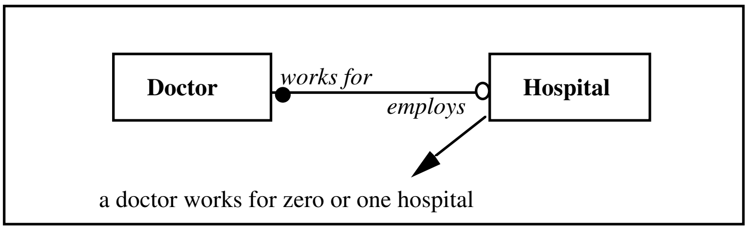


Figure 42: An optional cardinality on associations

The definition prohibits two cases allowed in the E/R modeling approach:

- case 1 : attributes and operations defined on an association,

- case 2 : more than two roles on an association.

3.3.1.5.3.2 How to model case 1 :

If you need to couple attributes and operations to associations, model association as a class.

For instance, if you want to describe the association *WORKS FOR* as follows :

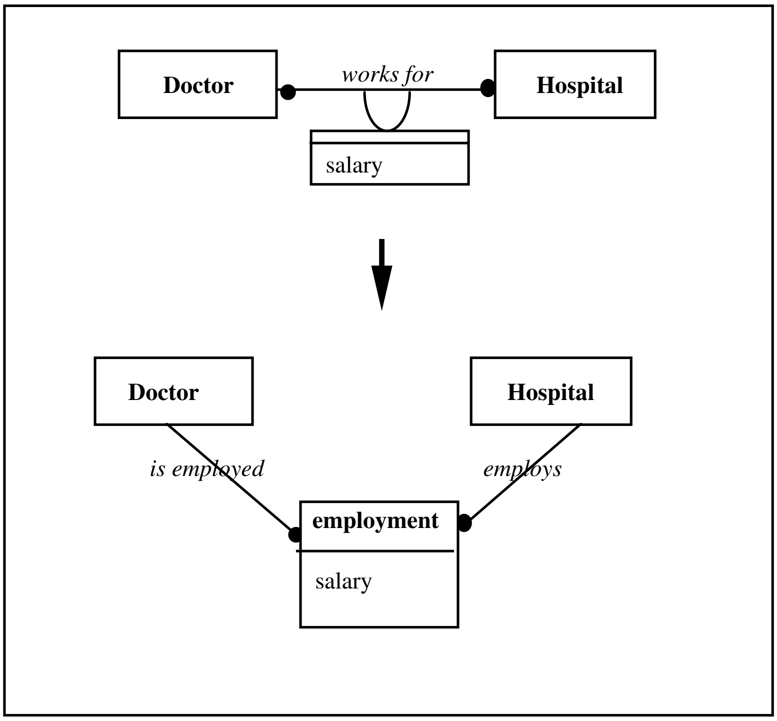


Figure 43: transformation of an association

The transformation mechanism leads to retype the association into the class *EMPLOYMENT.*

3.3.1.5.3.3 How to model case 2:

Each role of the association must be transformed into a binary association between two classes. Considering the following ternary association between the *STUDENT, PROFESSOR* and *UNIVERSITY* classes, the mapping process leads to define the class *COURSE* with an association for each of these classes.

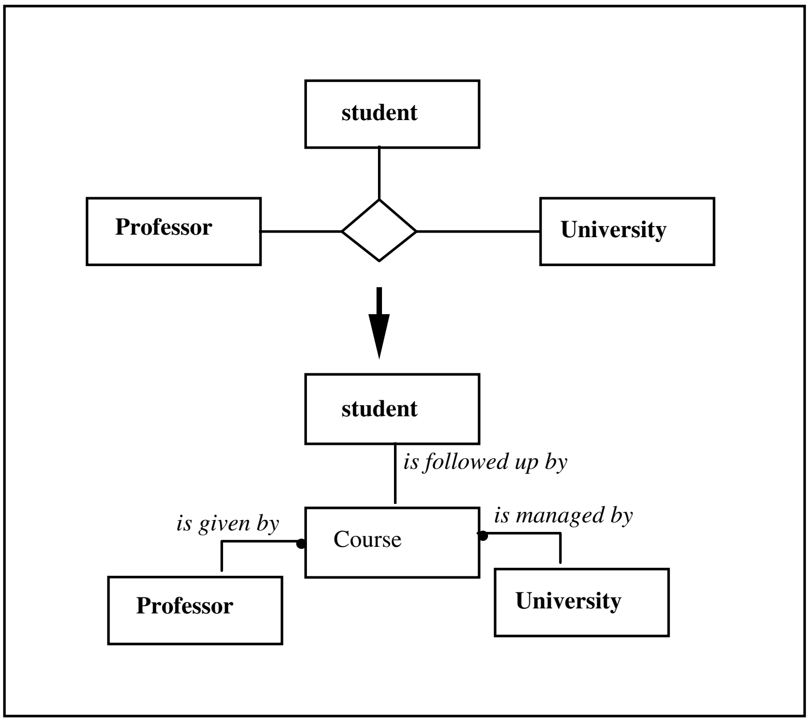


Figure 44: transformation of a ternary association

These restrictions are motivated by two reasons : the object orientation requires the IS structure to be a set of classes (not as a set of classes and associations) and for compliance to the ODMG report which excludes operations and attributes on associations.

3.3.1.6 Inheritance link

The ***inheritance*** and the generalization are powerful abstractions allowing to share common properties between classes and to preserve their differences. For instance, each person has a name, an address and a phone number. In addition, a patient has a weight and a doctor has a speciality.

The inheritance link is expressed with a hollow triangle, as represented on the following example.

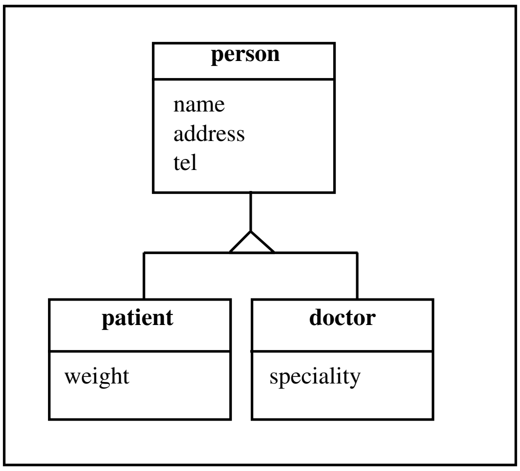


Figure 45: Example of inheritance link

T he *PERSON* class is considered as a superclass and the *PATIENT* and *DOCTOR* classes are subclasses.

Attributes, associations, aggregations and behavior characteristics defined in the superclass are inherited to subclasses.

An inheritance constraint restricts the possibility of creating specialized objects for a given generalized object. It is declared in the generalized class. Three types of inheritance constraints are distinguished: *partition constraints, disjunction constraints* and *covering constraints*. An inheritance constraint can cover a set of specialized classes.

3.3.1.6.1 Inheritance Constraint

A disjunction constraint covering a set of specialized classes specifies that every generalized object can correspond only to an object belonging to at the most one of the specialized classes. Therefore, the intersection of the extents of all specialized classes covered by the constraint is empty. For instance, a disjunction constraint defined on the object classes *VEHICLE* that covers the two specialized classes *CAR* and *TRUCK* indicates that a vehicle can either be a car, or a truck, or neither a car nor a truck (i.e., non-shaded portion), but not both a car and a truck (i.e., shaded portion) as shown in Figure 3.19.

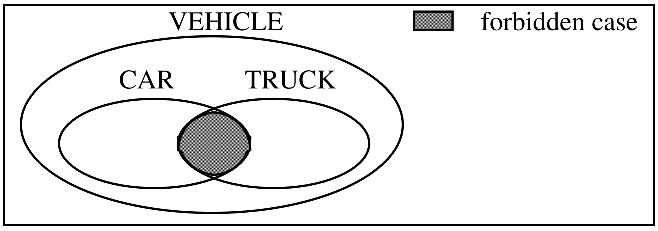


Figure 46: Disjunction constraint covers the specialized classes CAR and TRUCK

A covering constraint covering a set of specialized classes specifies that every generalized object must always correspond to at least an object belonging to one of the specialized classes. Moreover, the union of the extents of all specialized classes covered by the constraint is equal to the extent of the generalized class. For instance, a covering constraint defined on the object class *PERSON* that covers the two specialized classes *STUDENT* and *TEACHER* indicates that a person must either be a student, or a teacher, or both a student and a teacher (i.e., non-shaded portion) and nothing else as shown in Figure 3.20.

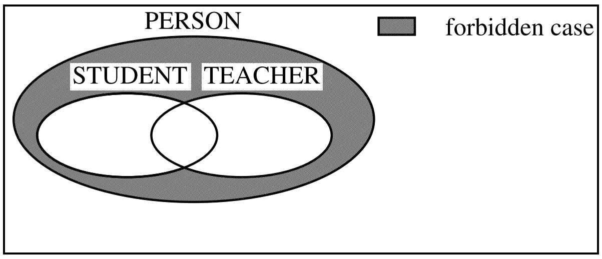


Figure 47: Covering constraint covering the specialized classes STUDENT and TEACHER

A partition constraint covering a set of specialized classes is at the same time a disjunction constraint and a covering constraint. This means that every generalized object must correspond exactly to an object belonging to one of the specialized classes. For instance, a partition constraint defined on the object class *PERSON* that covers the two specialized classes *MAN* and *WOMAN* indicates that a person must either be a man or a woman (i.e., non-shaded portion) and nothing else (i.e., shaded portion) as shown in Figure 3.21.

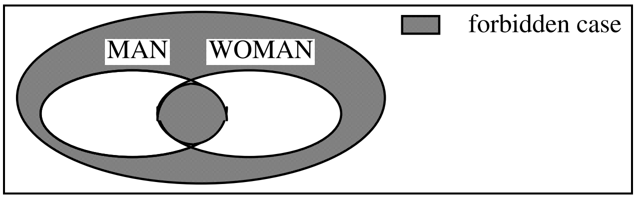
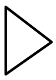
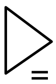


Figure 48: partition constraint covering the specialized classes MAN and WOMAN

 The solid triangle means that the specialized classes are overlapping (or not disjoints).

 A simple triangle means that the specialized classes are not overlapping (disjoint constraint).

 The equal symbol beside the triangle means that the specialized classes cover the generic classes (covering constraint).

 The combination of a solid triangle and the equal symbol expresses the partition constraint.

3.3.1.6.2 Clustering

The specialized classes of a generalized class may be grouped into *clusters*, each of the clusters is conditioned by an *inheritance constraint* . As a result of specialization through criteria of the inheritance constraints, the same specialized class may be constrained by more than one inheritance constraint. For instance, persons are grouped into two clusters, respectively consisting of the object classes *MAN* and *WOMAN*, and the object classes *SINGLE*, *MARRIED*, and *DIVORCED*.

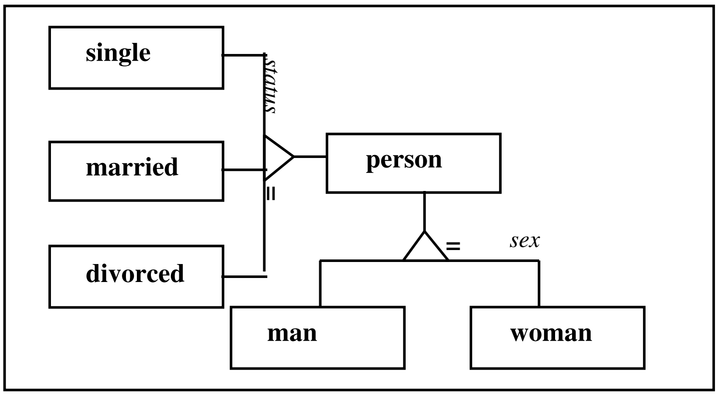


Figure 49: Clusters of the class PERSON

• Properties of the inheritance link :

Objects participating in such a relationship own the same object identity. The concept of inheritance to be proposed is based on the *multi-instantiation principle*.

The inheritance relationship between two object classes verifies the following properties:

- identity inclusion: each specialized object has the same identity as its generalized object;
- life cycle inclusion: the life cycle duration of a specialized object is included in the life cycle duration of the corresponding generalized object;
- class description inclusion: the description of a specialized object implicitly includes the description of the corresponding generalized object.

As a result of the multi-instanciation principle, *multiple inheritance* is allowed but is limited to the condition where the generalized classes inherited by an object class must be originated from a unique ancestor in the inheritance hierarchy. This also means that an object class can inherit the characteristics from several object classes.

Semantically, the inheritance link induces some behavioral dependencies between specialized objects and generalized objects as follows:

- creation of a specialized object implies simultaneous creation of its corresponding generalized object(s), provided that the latter still has not existed yet;
- deletion of a generalized object implies simultaneous deletion of its specialized objects;
- deletion of a specialized object having several generalized objects implies deletion of at least one of these generalized objects due to the principle of multi-instanciation.

The inheritance link is associated to a mechanism called augmentation mechanism, which is related to the execution of those basic operations or events having the same name defined in both the specialized class as well as its generalized class. For instance, if the basic operation of the specialized class is invoked, the mechanism causes this operation to be executed in addition to the basic operation having the same name defined in the corresponding generalized class.

The example on simple inheritance is described by the fact that the objects of the generalized class *PERSON* are specialized into objects corresponding to men, women, people who are single, people who are married and people who are divorced. Hence, the specialized classes *MAN*, *WOMAN*, *SINGLE*, *MARRIED* and *DIVORCED* inherit the structural and behavioral characteristics of *PERSON*. Furthermore, each object of a specialized class such as *MAN* has the same object identity as the corresponding object in the generalized class *PERSON*. This implies that the same person can be viewed as a man in another perspective.

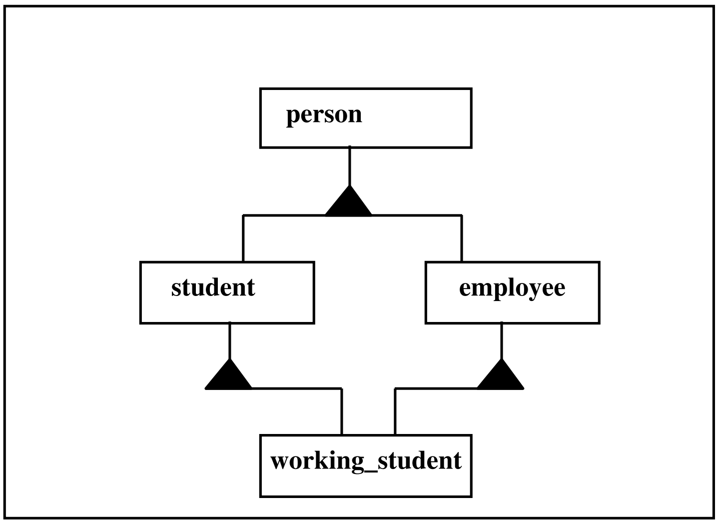


Figure 50: Multiple inheritance

The example on multiple inheritance illustrates the fact that a working student who is both a student and an employee is a person. The restriction of the multiple inheritance is reflected by the acyclic inheritance graph requiring a common ancestor (the object class *PERSON*) to be issued from the two generalized classes *STUDENT* and *EMPLOYEE* of the specialized class *WORKING_STUDENT*. The same explanation can be argued on multi-instanciation, a working student is an instance of *STUDENT* at one point of view and an instance of *EMPLOYEE* at a different point of view. These two instances are different perspectives of the working student, who is viewed as a person at a more general perspective.

3.3.1.7 Constraints

Several constraints are still embedded in the concept definitions. For instance, referential constraints and cardinalities constraints are defined in the relationship between object classes : association and aggregation. In addition, the inheritance constraints are defined in the inheritance link.

Additional constraints can be described in the definition of an object class. These constraints are classified in two groups : *attribute* constraint and *uniqueness* constraint.

3.3.1.7.1 Attribute Constraint

An attribute constraint is a property assertion that restricts either the attribute values of an object, or the cardinalities of the attributes defining related objects or domains according to a certain criterion. It is expressed as a boolean assertion. For instance, the invariant stating that *the weightdate of a patient must be greater than the birth date* is specified in the object class *PATIENT* as shown in figure 3.24.

Patient

properties
name : STRING
birthdate : DATE
weight : REAL
weightdate : DATE
patient_state : PATIENT-STATE
patient_address : ADDRESS
numberofpatients : INTEGER
constraints
weightdate > birthdate

Figure 51: Attribute Constraint on the class PATIENT

An attribute constraint can also be defined so that an inherited attribute of a composite specialized object takes its value from a component specialized object, where both objects are related via an association or an aggregation defined between their generalized objects. For instance, an attribute constraint defined in the specialized object class *STUDENT* restricts its property named *period* to take its value from the specialized object class *STUDENT-CARD*, where both classes are related via an association property define between *PERSON* and *CARD*.

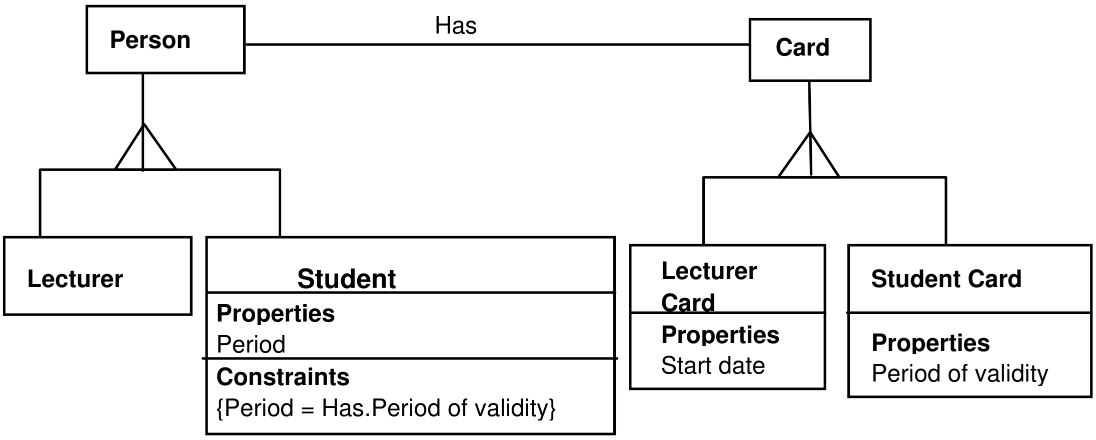


Figure 52: Example of an attribute constraint

3.3.1.7.2 Uniqueness Constraint

A uniqueness constraint ensures uniqueness of objects in an object class based on the uniqueness of values for one or several attributes of the objects. It corresponds to the notion of key in the relational model. All uniqueness constraints are declared as invariants. For instance, the uniqueness of objects in the object class *CAR* can be identified via the pair of attributes (*country, registration_number*) as shown in figure 3.26.

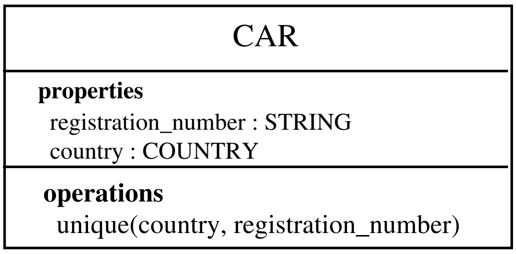


Figure 53: Uniqueness constraint of objects on the attributes country and registration_number

The function *unique()* is pre-defined for verifying the uniqueness of one or several attributes in an object class. The reference attribute country and the domain attribute *registration_number* refer to an object of the referenced class *COUNTRY* and a car registration number respectively. The invariant indicates that the pair of values corresponding to the country and the registration number must be unique for all objects of *CAR*.

3.3.1.8 Summary :

The structure abstraction deals with three classes which are *object class, actor class* and *calendar class*. Each of these classes are characterized by *attributes*. Only object classes are defined by a set of *properties* which are attributes and relationships. Two kinds of relationships between object classes are supported :*association* and *aggregation*.

A property is regarded as a mapping function that gives for each object a value of a given domain or extent of an object class. For example, weight is a mapping function providing, for each patient , his weight. Constraints on attributes can be defined on object class.

Domains associated to attributes can be *pre-defined* or *user-defined* domain.

Associations express relationships between objects. Such associations are restricted to a reference link from one class to another. It is embedded to the object class definition of the source class. An association from A to one or more objects B describes a main direction and the inverse link from B to A can be managed if it is necessary. *Cardinalities constraints* are expressed on association.

Aggregation is supported between object classes. It expresses that an object class A is a part of an object class B. The aggregation allows as it is in the association link to access the composite object from the component object. Cardinalities constraints are defined for the composing object class.

The *inheritance* link between object classes is supported. It expresses that an object class is a specialization or a generalization of another object class. *Cluster* is a group of specialized classes which forms a partition of a superclass. Three inheritance constraints can be expressed : *disjunction, covering and partitioning constraints*.

3.3.2 The function abstraction

The function abstraction emphasizes the concept of *operation*. This concept provides a mean for locally describing the set of possible actions to be performed on objects.

3.3.2.1 Operations

An ***operation*** is a function or a transformation that may be applied on or by one object of a class. Operations are described in a class definition. They allow to define the behavior of objects of this class.

For instance, *WEIGHT_CHANGE* , *ADDRESS_CHANGE, AGE* and *$AVERAGE_AGE* are operations defined in the *PATIENT* class as shown in Figure 54.

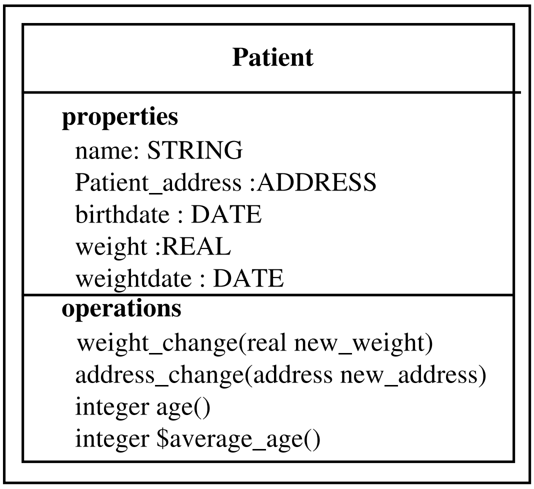


Figure 54: Operations of the class PATIENT

The operation has a target object as an implicit argument. The behavior of the operations depends on the class of its target. The same operation name can be reused in different classes. In this case, the right implementation of the operation is derived from the class of the target object. This mechanism is called *polymorphism*.

The operation may have arguments in addition to the target object. Such arguments are defined in the signature of the operation. To simplify the semantic complexity, the same operation can not be defined in the same class with different signatures (as it is done in O2).

The signature of an operation is composed of the operation name, the argument list (limited by parenthesis) and the result domain (introduced before the operation name).

Each operation in an object class has a unique name. There are two kinds of operations that may be defined in an object class. There are classified under *basic operations* and *queries*. During modeling, it is useful to distinguish operations which are side effect on objects and those which compute a functional value without modifying objects.

3.3.2.1.1 Basic operations

Basic operations correspond to actions that change object states. A basic operation is activated by an event trigger, and thus causing a state change to an object at a given time.

Basic operations are encapsulated in object classes. The activation of a basic operation is triggered by an event, as a result, a side effect (state change) will be produced in the encapsulating object. In addition to an operation name and an operation text, a basic operation is also characterized by an operation type: *create*, *delete* or *update*. Explicit specification of operation type enables the verification of the consistency of the behavior of the application. As a result, a basic operation may be either an operation of creation, an operation of update or an operation of deletion, referred to as *create operation*, *update operation* and *delete operation* respectively. A basic operation may be triggered by one or several events at different time. Figure 3.28 shows the various basic operations defined in the object class *REQUEST OF LOAN.*

Request of loan

properties

dateofrequest : DATE
subscriber : asso(onemany, SUBSCRIBER)
book : asso (onemany, BOOK)
state : REQUEST STATE

operations

request acceptance (id sub:string, id book : string)
request wait(id sub:string, id book : string)
notification ()
cancellation()

Figure 55: Basic operations encapsulated in the object class REQUEST OF LOAN

Hence, the create operation type is associated to the basic operation *acceptance* and *wait*, the *delete* operation type to the basic operation *cancellation*, and the *update* operation type to the basic operation *notification*.

3.3.2.1.2 Queries

A query does not cause state changes to objects. It only provides services to other objects either by retrieving the appropriate object, or by carrying out computation from the information of an object or some objects from an object class, and returns the computed result. For instance, a query named *average_age* can be defined in the object class *PATIENT* to calculate the average age of subscribers as shown in Figure 56.

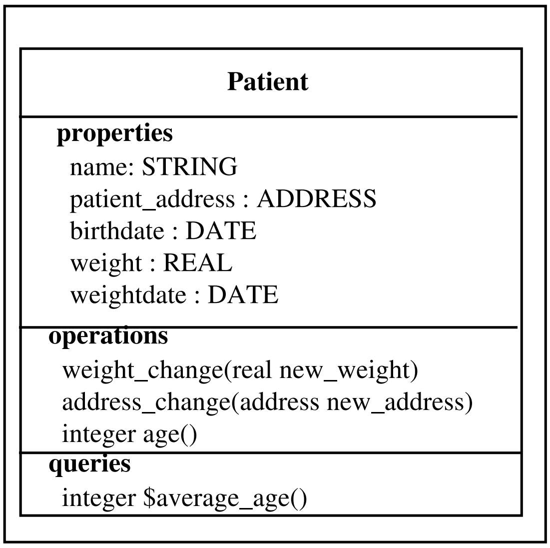


Figure 56: Query named average_age defined in the object class PATIENT

The query needs only to access the values of the attribute *birthdate* of all objects of *PATIENT* without producing any side effect and returns the average value calculated from these values.

Queries with no arguments can be considered as ***derived attributes***. For example, the operation age is considered as a derived attribute, which compute the age of the patient from the attribute *birthdate*.

3.3.2.1.3 Object operations Vs Class operations

Orthogonally to these categories of operations, we can distinguish two levels of operations : ***Object operation*** and ***Class operation***.

An *Object operation* is applied on one object of the class whereas a *Class operation* is applied to the extent of the class or to class attributes. It means that a class operation computes a value requiring all objects of the target class. For example, the operation *average_age* is a class operation computing the average of the derived attribute age of all objects of the class *PATIENT*. A Class operation has an implicit argument which is the target class.

The Class operation is mentioned in the object class definition with the '$' mark.

3.3.2.2 Links between operations

Aggregation and inheritance links between object classes imply aggregation and inheritance links between operations of these object classes.

3.3.2.2.1 Aggregation between operations

Aggregation between object classes imply aggregation between operations. As a result of the strong coupling of object behavior implied by an aggregation link between the object classes *ORDER* and *ORDER LINE* (with onetomany cardinality) the creation of an object of *ORDER* will invoke the creation of a set of objects of *ORDER LINE*, while the deletion of an object of *ORDER* will invoke the deletion of all its component objects in *ORDER LINE*. Consequently, the operations *ORDER_CREATION, ORDER_DELETION* are aggregate operations. They are composed of a set of the operation *ORDERLINE_CREATION* or respectively *ORDERLINE_DELETION* .

```
Order Order_creation(...): aggr (many, Orderline::orderline_creation(...))
Order_deletion(...) : aggr (many, Orderline::orderline_deletion(...))
```

3.3.2.2.2 Inheritance between operations

The inheritance link between object classes is associated to a mechanism called *augmentation mechanism*, which is related to the execution of those basic operations or events having the same name defined in both the specialized class as well as its generalized class. For instance, if the basic operation of the specialized class is invoked, the mechanism causes this operation to be executed in addition to the basic operation having the same name defined in the corresponding generalized class.

As a result of the inheritance link between object classes, a creation of an object of the specialized class implies the creation of the object of the superclass with the same identity. For instance, the operation *MAN_CREATION* is an operation which is derived from the operation *PERSON_CREATION*.

```
Man Man_creation(): Person::Person_creation()
Man_deletion() : Person::Person_deletion()
```

The augmentation mechanism is allowed by deriving operation of the specialized class from operation of the generalized class.

3.3.2.3 External Operations

Actor classes are used to describe the environment of the system. The impact of an actor on the system is illustrated in terms of external events described in the section *the behavior abstraction*. The interaction of the system to its environment is shown in terms of external operations.

External operations allow to describe how messages are sent to an actor. The external operation *SEND_NOTICE* allows to the system to send a notice to a subscriber when his waiting request is selected. The external operation is described with its signature.

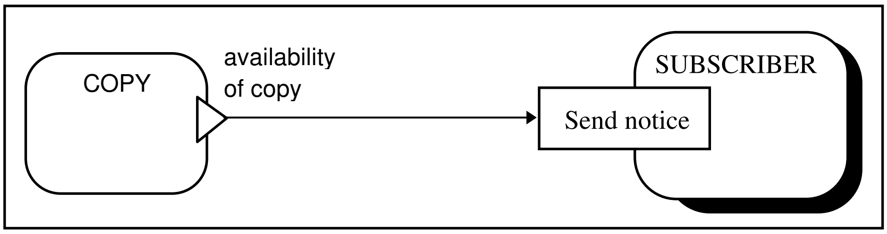


Figure 57: Example of an external operation

3.3.2.4 Summary :

The function abstraction deals with the concepts of *operation*.

Operations are classified according to their effect and their scope.

If the operation modifies an object or an attribute, it is called a *basic operation* otherwise it is called a *query*.

If the scope of the operation is the class, it is named a *class operation* whereas it is considered as an *object operation*.

Links between operations are coming from the aggregation and inheritance links between object classes. Aggregation between operations are used to emphasize the behavioral consequences of the aggregation link between object classes.

Inheritance link between operations are allowed to define an operation of a specialized class by augmentation of an operation of a generalized class.

External operations are attached to actor classes for showing the interaction of the system to its environment. The system can send messages to actors through the external operations. They are described with their signature.

3.3.3 The behavior abstraction

The behavior abstraction deals with the concept of *event*.

Three kinds of events are defined : *internal*, *external* and *temporal*. Internal events are defined in object classes, external ones are related to actor classes and finally temporal events are embedded in calendar classes.

The *event description* details the treatment to execute after the occurrence of an event. This treatment is composed by operations to perform. Each triggering operation can be conditional or iterative. It is conditional, a *triggering condition* is specified whereas if it is iterative, a *triggering factor* is defined.

The *dynamic transition* is associated to each event. It represents a unit of atomicity, consistency and integrity.

Two relationships between events are defined : the induction and the concurrence. The *induction link* between two events expresses that an event occurrence of an event may imply the occurrence of another event. Events are *concurrent* when their executions implies a conflicting access to at least one object.

3.3.3.1 Event

An event occurrence is a projection of a real world phenomenon whose occurrence at a given instant generates an impact on some objects of the IS.

For instance, the phenomenon *"arrival of the patient Smith to the hospital London"* and *"arrival of delivery for the order number 88"* are two event occurrences, that generate impacts on the objects *"patient Smith"* and *"order number 88"* respectively. Event occurrences are differentiated from objects in the sense that they are instantaneous while objects have life duration.

An event is the result of the classification process from event occurrences with similar characteristics.

Hence, an event describes a set of event occurrences of the same nature. The description includes the cause of the occurrence of an event occurrence and its impact on objects of the IS. There are three types of events: ***external events, internal events and temporal events***. Thus an external event groups external event occurrences of the same nature, an internal event groups internal event occurrences of the same nature, while a temporal event groups temporal event occurrences of the same nature. An event represents either a state change of an object (internal event), an arrival of message made by an actor (external event), or the occurrence of time (temporal event).

The occurrence of an external event corresponds to the arrival of a valid message made by a particular actor, the occurrence of an internal event corresponds to the state change of an object, while the occurrence of a temporal event corresponds to the verification of a temporal assertion. An event is represented graphically in the dynamic graph, by a triangular notation. For instance, the external event *ARRIVAL_OF_PATIENT_TO_AN_HOSPITAL* is attached to the actor class *PATIENT,* the internal event *NEW_AVAILABILITY* to the object class *COPY*, and the temporal event *EACH_WEEK_AFTER_EXPIRATION_OF_LOAN* to the calendar class *CLOCK*.

The behavior abstraction is based on three kinds of classes : ***object class, actor class and calendar class***.

An object class is defined by the description of the structural and behavioral characteristics of its objects. The *behavior aspects* are described in terms of the events through which state changes (evolution) of objects of that type can be ascertained.

Object classes describe the internal structure of the system to model whereas actor classes represent the environment of the system to model. By modeling actors, we are interested to show the interaction between the system and its environment. The interaction existing between an actor and the system is described in terms of external events.

Actors interact with the system by producing stimuli to the system in terms of messages. Their reception implies a reaction of the system which is described in the trigger part of the external event. The behavior aspects of an actor class is described in terms of *external events* initiated by an actor of that type which have effects on the internal structure of the system.

The calendar class can be simulated as the system's internal clock.

It describes time occurrences. There is usually one calendar class which is associated to the Gregorian calendar called *CLOCK*. This predefined class has only one instance *clock*. Particular occurrences of time induce the occurrence of temporal events which will propagate state changes to the objects in the IS.

The list of temporal events having effects on the system are defined in the calendar class *CLOCK*.

The occurrence of an event has an impact on the states of some objects in an IS. An event is defined by an event predicate and a trigger part.

3.3.3.1.1 Typology of events

• ***An internal event*** is related to an object class because it represents a particular state change of one object of this class. The event *AVAILABILITY_OF_COPY* occurs when a copy becomes available. Thus, the predicate was false and become true due to the effect of an operation applied to a copy. The triggering part of the event is executed. The context execution of an event depends on the type of event (external, internal and temporal). For an internal event, the context is the object on which the event has just been ascertained by the internal event. The internal events are described in the object class definition.

An internal event is represented with a white triangle and it is associated to an object class.

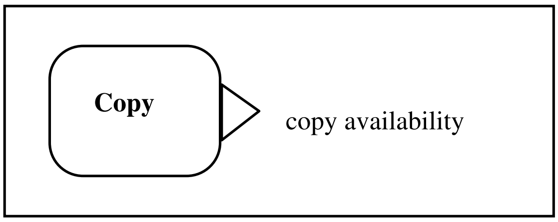


Figure 58: Internal event

The *Inheritance between object classes* can leads to define an internal event of the specialized class by augmentation of an internal event of the generalized class. The specialized event inherits the predicate and the trigger part from the generalized event. A specialized event can occur also when the generalized event with the same name occurs. The predicate and the trigger part of the specialized event are defined by augmentation of the generalized one.

The *aggregation between object classes* implies that the internal events associated to the component object class are available in the composite object class.

• An ***external event*** is related to a message sent by a particular actor to the system. The *RETURN_LOAN* event occurs when the system receives a message occurrence, recognizes the type of this message, but also when the predicate of the associated external event is true. The external event predicate is composed of verification conditions on the message content. The triggering part of the event is executed, after the verification of the predicate. For an external event, the context is the message received. The external events are described in the actor class definition.

An external event is represented with a solid triangle associated to an actor class.

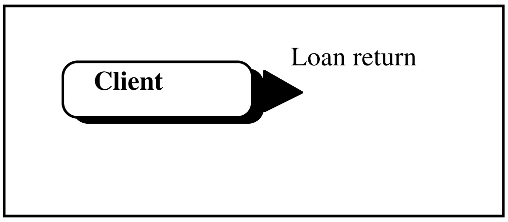


Figure 59: External event

• A ***temporal event*** is related to the calendar class *CLOCK*. The *EXPIRATION_OF_*LOAN event occurs when the temporal assertion defined in the predicate is true. The internal clock evolves due to the time passing. When an occurrence time verifies the temporal event predicate, its triggering part is executed. The context of it is the time occurrence and the object used as a reference time in the predicate (event or object in the system). The temporal events are described in the calendar class definition.

The temporal event is represented graphically with a grey triangle as follows :

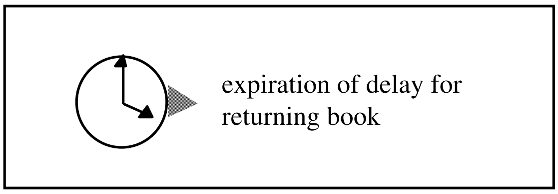


Figure 60: Temporal event

3.3.3.1.2 The event description

• ***The event description*** includes the description of its predicate part and its trigger part. The event predicate describes the condition for the occurrence of the event, while the event trigger describes how basic operations will be triggered on objects when the event occurs. In a global behavior graph, an operation triggering is represented by an arrow originated from an event and terminated in the operation to be triggered.

Let us take the example of the library IS for illustrating the description of an event on the availability of a copy of book. When a copy of book is available, the reservation for the copy will be made if there is a waiting request for it. The description reflects the representation shown in the following figure by an internal event named new_availability for ascertaining if it exists an available copy.

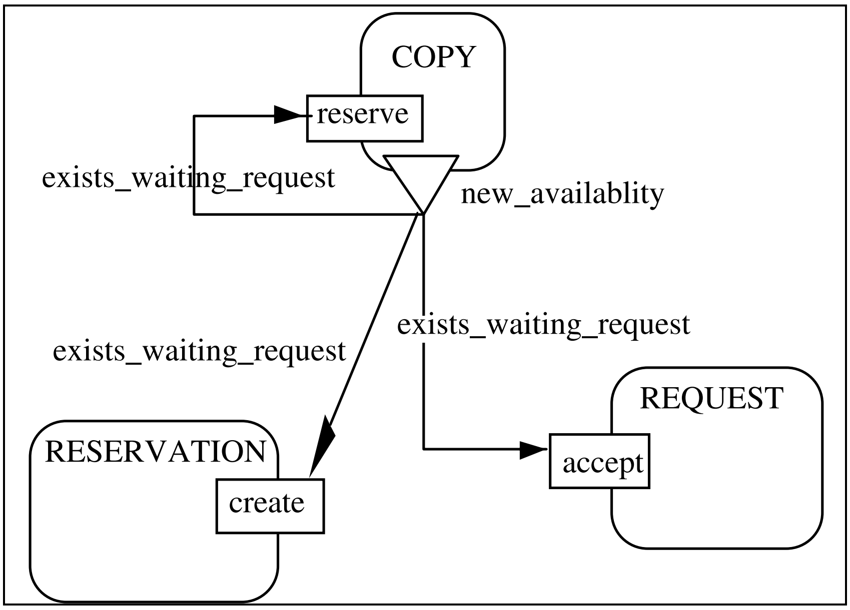


Figure 61: Description of the internal event new_availability

The event predicate of *new*_availability describes the availability of a copy represented by the fact that the old state of the copy is not available and the new state is available. The figure also shows the trigger part of the event. A trigger describes how the basic operations will be triggered. There are three operation triggering associated to the trigger. If the event occurs, the three operation triggering will be activated in parallel. An operation triggering defines the basic operation to be triggered, and it may also be associated by a triggering condition and/or a triggering factor. In the figure, only a triggering condition named exists_waiting_request is defined for all the operation triggering. Triggering factors will be explained later.

If the event occurs, the event context will be passed to the triggering conditions/triggering factors associated to the operation triggering. The context of an event contains the information needed during event triggering. It depends on the type of event: external, internal or temporal. For an external event, the context is the message object representing the message contents received from the external environment. For an internal event, the context is the object on which a state change has just been ascertained by the event. In the example, the context is the object corresponding to the copy ascertained by the internal event, *new_availability*, whose old state is not available and whose new state is available. For a temporal event, the context is the time occurrence of the calendar.

• An ***event predicate*** describes when an event occurs, i.e., the occurrence condition of the event. There are three types of event predicates: ***external event predicates, internal event predicates*** and ***temporal event predicates.*** The external event predicate is described by using the message attributes defined in the external event description, the internal event predicate by using the attributes and/or the states of the objects of an object class, and the temporal event predicate by a temporal assertion.

The structure of the ***event trigger*** is common for all kinds of events. The trigger involves the verification of the associated triggering condition and the evaluation of the associated triggering factor for each operation triggering. The basic operation of an operation triggering is activated only if the associated control condition is true. If a triggering factor is also associated to the operation triggering, the triggering factor will be computed first and the basic operation will be executed repeatedly for each element of the triggering factor.

• ***A triggering condition :*** before the operation is actually invoked, its control condition is checked. If the results of evaluating the condition are true, its operation is invoked. If false, the operation is not invoked.

Figure 62: A graphical representation of a triggering condition

• ***A triggering factor*** is expressed as a function returning a set of objects. The operation is iterated for each object of this set. The factor may be used by several events. It describes the parameters for the iterative execution of a basic operation. This means, it determines the set of objects of an object class on which a particular basic operation must be applied when the triggering condition of an event occurrence is verified.

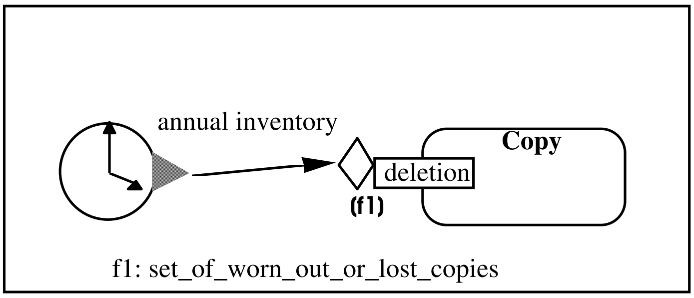


Figure 63: A graphical representation of a triggering factor

• ***Triggering operation*** **:** an operation is an action that can be processed alone in the organization and can change the state of its objects. This concept is represented with a box having the operation name inside as shown in Figure 64: Operations of classes. An operation may change the state of an object.

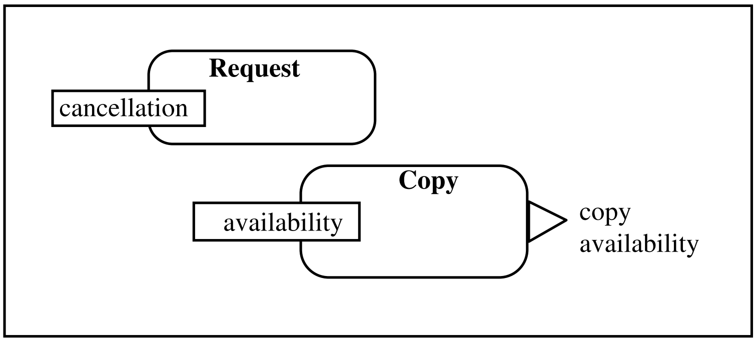


Figure 64: Operations of classes

An operation is invoked by an event but an operation may be invoked by one or more events.

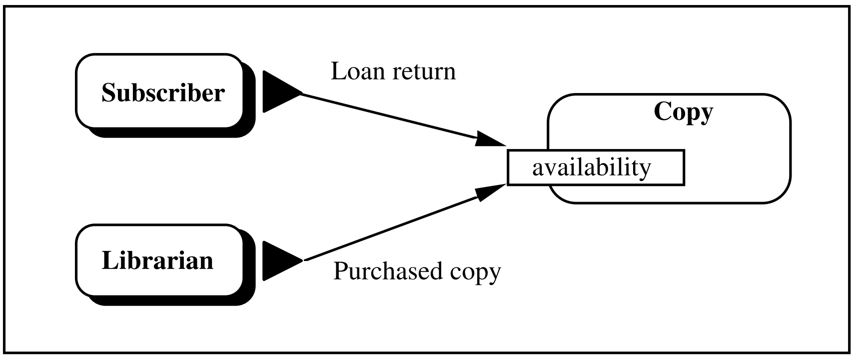


Figure 65: Operation triggering

3.3.3.2 Dynamic transition

• A *dynamic transition* [Rolland82] is made up of an event, a set of basic operations triggered by the event and a set of persistent objects manipulated by these operations. A dynamic transition is analogous to an elementary transaction of databases, that is, a unit of atomicity, consistency and integrity. In a wider perspective, a dynamic transition may be defined globally and explicitly by the sequence of event occurrences caused by the occurrence of an external or a temporal event. The execution of a dynamic transition must lead the database from a coherent state to a coherent state.

In a dynamic transition, the basic operations that manipulate persistent objects are triggered in parallel by the event. Therefore, any creation, update or deletion of persistent objects within the dynamic transition must be completely done before any other process can be carried out by/on any of these persistent objects. In this sense, a dynamic transition can easily be mapped into a run-time elementary transaction.

In conclusion, the dynamic transition describes the effect of the event occurrence. When a dynamic transition occurs, one or several basic operations is(are) triggered in parallel on objects. The event together with its trigger part make up a dynamic transition whose interpretation is equivalent to a synchronization of parallel processes for triggering basic operations on objects. The dynamic transition indicates that any update made to an object as a result of an operation triggering does not appear to other operation triggering accessing the same object in the same dynamic transition. It provides a more conceptual view of atomicity, consistency and integrity of objects.
The representation of the behavior of the business activities has been summarized as shown in Figure 66.

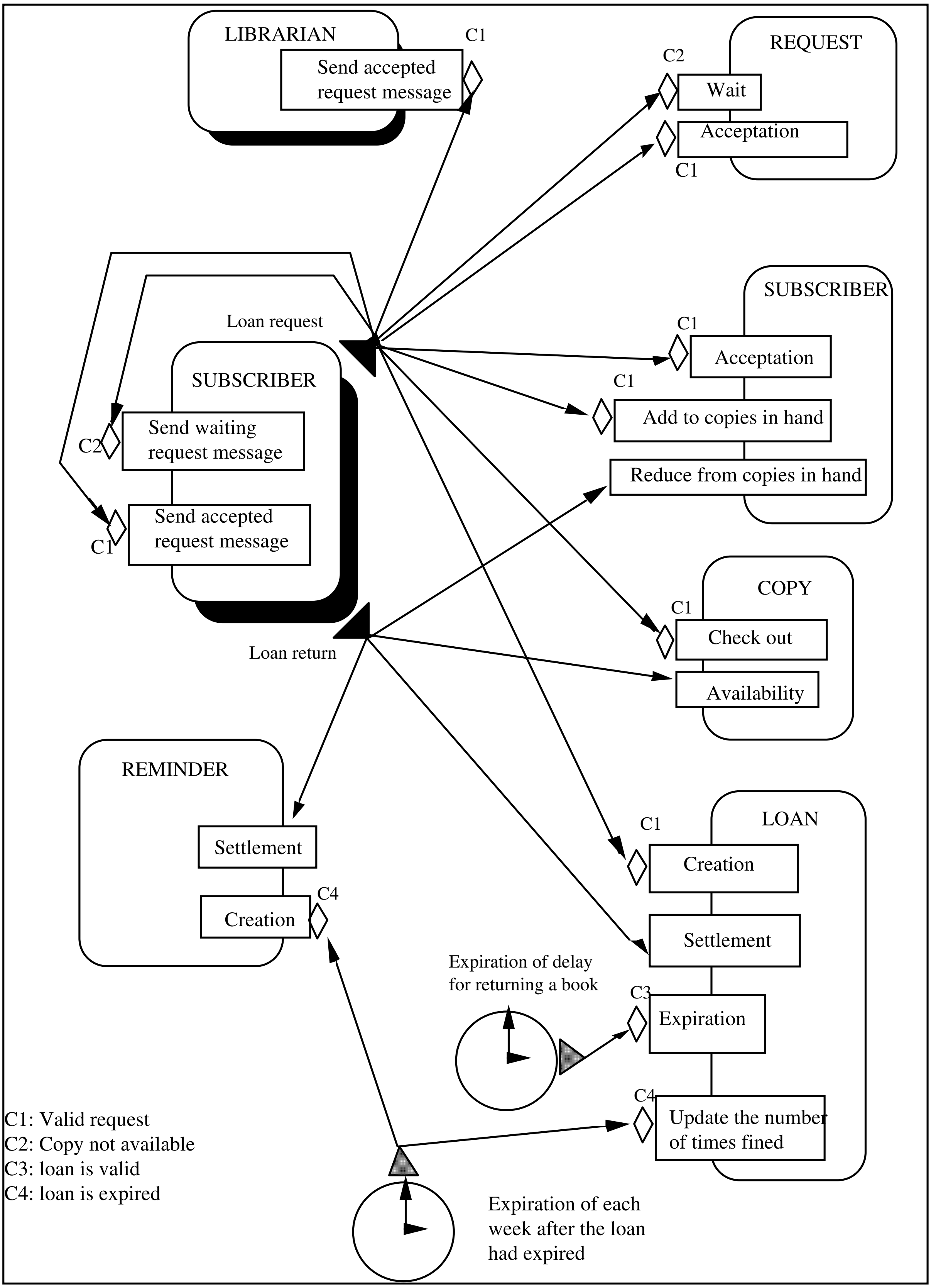


Figure 66: A global behavior graph part

3.3.3.3 Event Induction Graph

The behavior of the application is expressed by a set of events. The application requires a particular sequence between these events which are described in the behavior schema but in an implicit manner. The event induction graph makes the causalities between events explicit. The induction means that an event may or must imply the occurrence of another event. This link is defined for the three types of events : internal, external and temporal.

The *induction* link is defined between two events. This relationship is characterized by the following syntax :

<induction link>::=<modality> [**if** <condition_name>] [**with delay** <interval>] <cardinality> <event_name>
<modality>::=**may/must**
<cardinality>::=**one/more**

The induction link is characterized by :
- its modality : obligation or possibility,
- its existence : unconditional or conditional,
- its temporal constraint : immediate or delayed,
- its cardinality : simple or multiple.

The modality of the induction link comes from the distinction between permission and obligation made in the modal logic [Hughes88] (permission and obligation). Obligation is associated to the obliged induction whereas permission corresponds to possible induction. The ***obliged induction*** means that each occurrence of the source event <u>must</u> induce an occurrence of the target event. This constraint has to be verified for prohibiting something classified as "bad" from happening. For instance, the external event *RETURN_OF_LOAN* <u>must</u> imply systematically an occurrence of the internal event *AVAILABILITY_OF_COPY*. In the same way the external event *RESPONSE_OF_NOTIFICATION* <u>must</u> induce the occurrence of the internal event *AVAILABILITY_OF_COPY* (if the condition *SUBSCRIBER_IS_NOT_VALID* is true).

The ***possible induction*** means that each occurrence of the source event <u>should</u> induce an occurrence of the target event. This constraint is derived from the permission used in the modal logic, indicating something permissive but not sufficient is allowed to occur. For instance, the external event *REQUEST_FOR_*LOAN may induce an occurrence of the external event *RETURN_OF_LOAN*. In the same way, the internal event *AVAILABILITY_OF_*COPY should induce an occurrence of the external event *RESPONSE_OF_NOTIFICATION*. The target event of a possible induction link is an external event because it expresses an expected behavior of an actor of the system.

The existence of an induction link from the event *EV1* to EV2 can be unconditional or conditional. The existence of this link is conditional if *EV2* occurs only when a certain condition is true, after the performance of an occurrence of ev1. For instance, the external event *RESPONSE_OF_*NOTIFICATION may imply the occurrence of the internal event *AVAILABILITY_OF_*COPY only if the condition *SUBSCRIBER_IS_VALID* is true. On the contrary, the external event *RETURN_OF_LOAN* imply systematically the occurrence of the internal event *AVAILABILITY_OF_COPY.*

The temporal constraint of the induction link allows to describe if the induction must be either immediate or delayed within a time limit. For instance, the induction link <*RETURN_OF_*LOAN must imply an occurrence of the internal event *AVAILABILITY_OF_COPY*> is an example of immediate induction. The induction link <*AVAILABILITY_OF_COPY* may induce in a delay of seven days an occurrence of the external event *RESPONSE_OF_NOTIFICATION* > is an example of delayed induction with a strict delay of 7 days. In this case, the target event should occur after the period of validity. Exception events can be defined to handle the expiration time.

The cardinality of the induction link permits to express that an occurrence of the source event induce one or many occurrence(s) of the target event. The induction is simple if the cardinality is one whereas it is called multiple if the cardinality is many. For instance, the external event *RETURN_OF_LOAN* must imply only one occurrence of the internal event *AVAILABILITY_OF_COPY* (simple induction). The event *NEW_SUBSCRIPTION* should induce many occurrences of the event *REQUEST_FOR_LOAN* in a delay of thirteen months (example of multiple induction).

The graphical notation of the induction link is shown by Figure 67: event induction graph.

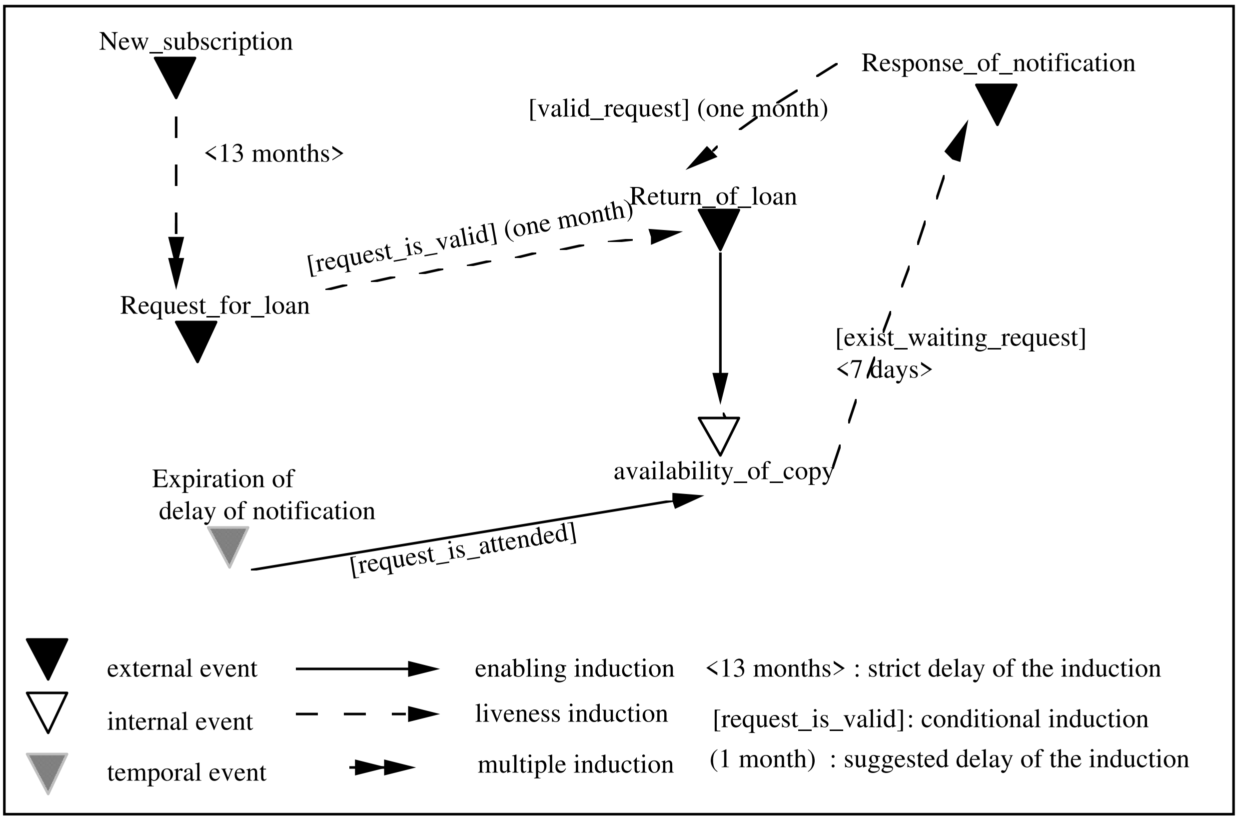


Figure 67: event induction graph

3.3.3.4 Event Concurrence Graph

Two events are concurrent if their executions implies conflicting accesses to the same object (i.e. read/write or write/write).

The concurrence link is bi-directional. The solving mechanism consists of sequencing their execution. For instance, the external event *REQUEST_FOR_LOAN* is concurrent to the external event *RETURN_OF_*LOAN because the former requires to read the number of current loans of the subscriber whereas the latter update it. The analyst can define priorities for concurrent events. For instance, the event *REQUEST_FOR_LOAN* has a lower priority than *RETURN_OF_LOAN* (The return of loan will be executed first). Figure 68 shows the notation used for representing the concurrence between two events.

Figure 68: A concurrence link between events

3.3.3.5 Summary :

The behavior abstraction emphasizes the concept of event. The event definition is encapsulated into the class definition. A typology of classes allows to have a typology of events. An *internal* event is a particular state change of an object of an object class. An e*xternal event* is a message sent to the system by an actor of an actor class. Finally, a *temporal event* is a particular state changes of a clock of the calendar class.

The *predicate of an event* is a boolean assertion describing when the event has to occur. If the predicate is true, the event occurs and its trigger part is executed. A trigger part of an event is composed of operations to perform. Each of these operations can be conditional or iterative. A conditional operation means that the operation is performed only if a specific *triggering* condition is true. An iterative operation represents the fact that the operation must be applied on several objects. The set of objects modified by the operation is specified by the *triggering factor*.

The *dynamic* transition is composed of the recognition of the event and the execution of its trigger part. It represents a unit of atomicity, consistency and integrity. The end of each dynamic transition sets the database in the consistent state.

Managing the event execution requires to specify for each event the induced events and the concurrent events. The *event induction graph* allows to express the induction relationships between events. The induction means that an event must/may/should imply the occurrence of another event. This link is defined for the three types of event : internal, external and temporal.

The *concurrence* between two events corresponds to events using during their executions same objects with conflicting access types (i.e. read/write or write/write). This link is defined for the three types of event : internal, external and temporal. Concurrent events implies a sequencing of their execution. A priority can be specified for determining the execution order.

3.3.4 Diagrams associated to the model

The structural aspects (static aspects) of the IS are described in the ***global structure graph***, that provides the global structural perspective of the application.
The behavioral aspects (dynamic aspects) of the IS are described in some ***global behavior graphs***, that provide the global behavioral perspective of the application, through the hypothesis that the behavioral aspects are responsible for causing state changes to the structural aspects of the IS. The global behavior graph is completed by the event ***induction graph*** and the ***event concurrence graph***.
The local structural aspects and local behavioral aspects of the objects in the individual object classes are described in ***local class views***. These views correspond to the local structural and behavioral perspectives of the objects defined in the application.

3.3.4.1 The global structure graph

The global structure graph describes graphically all the object classes with their static relationships : aggregation, association and inheritance.

For instance , the global structure graph of the library system is graphically shown in Figure 69.

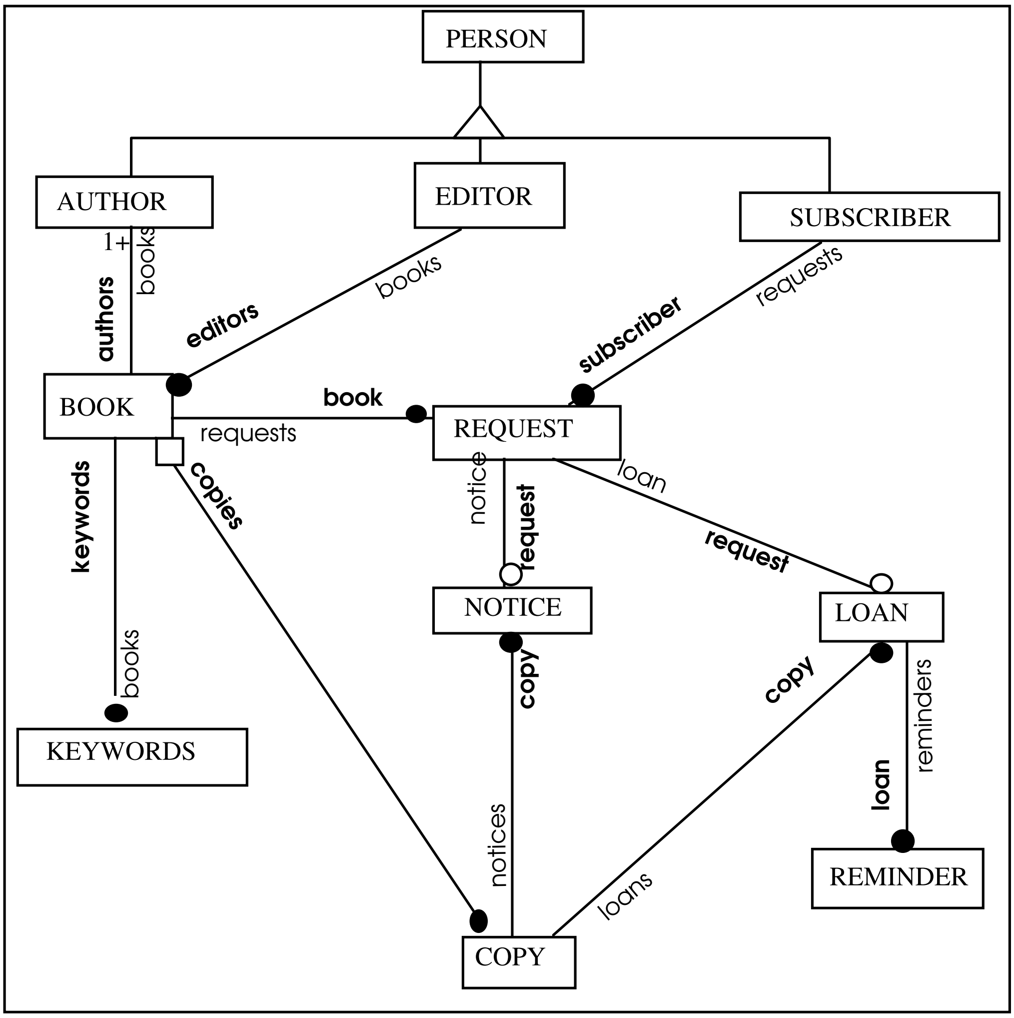


Figure 69: The Global structure graph of the library system

The global structure gives to the analyst the general map of the persistent information of the system.

3.3.4.2 The global behavior graphs

The global behavior graphs describe the dynamic of the application in terms of events, triggers and operations embedded in classes such as object classes, actor classes and calendar classes. This global behavior graph allows to see how an event is ascertained and what are the effects of an event.

Ideally it will be better to manage one global behavior graph, but in the reality it could be unreadable. It is why we suggest to split the global graph into several behavior graphs.

An example of a global behavior graph is shown in Figure 70.

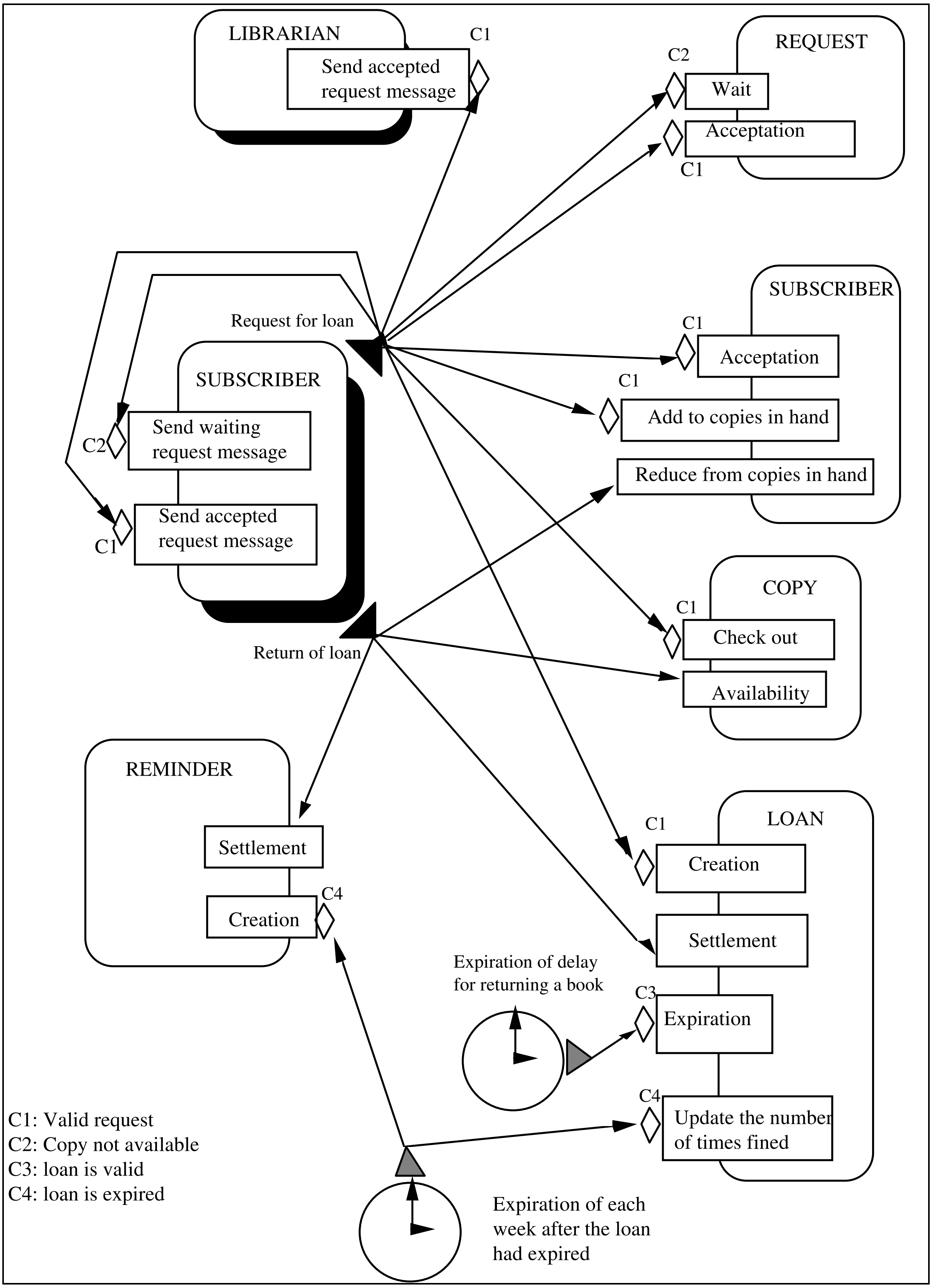


Figure 70: A global behavior graph

3.3.4.3 The event induction graph

The event induction graph completes the global behavior graph by defining the induction between any kinds of event : internal, external and temporal. The global behavior graphs provide indirectly the induction of the internal events.

For instance , the event induction graph of the library system is shown in Figure 71.

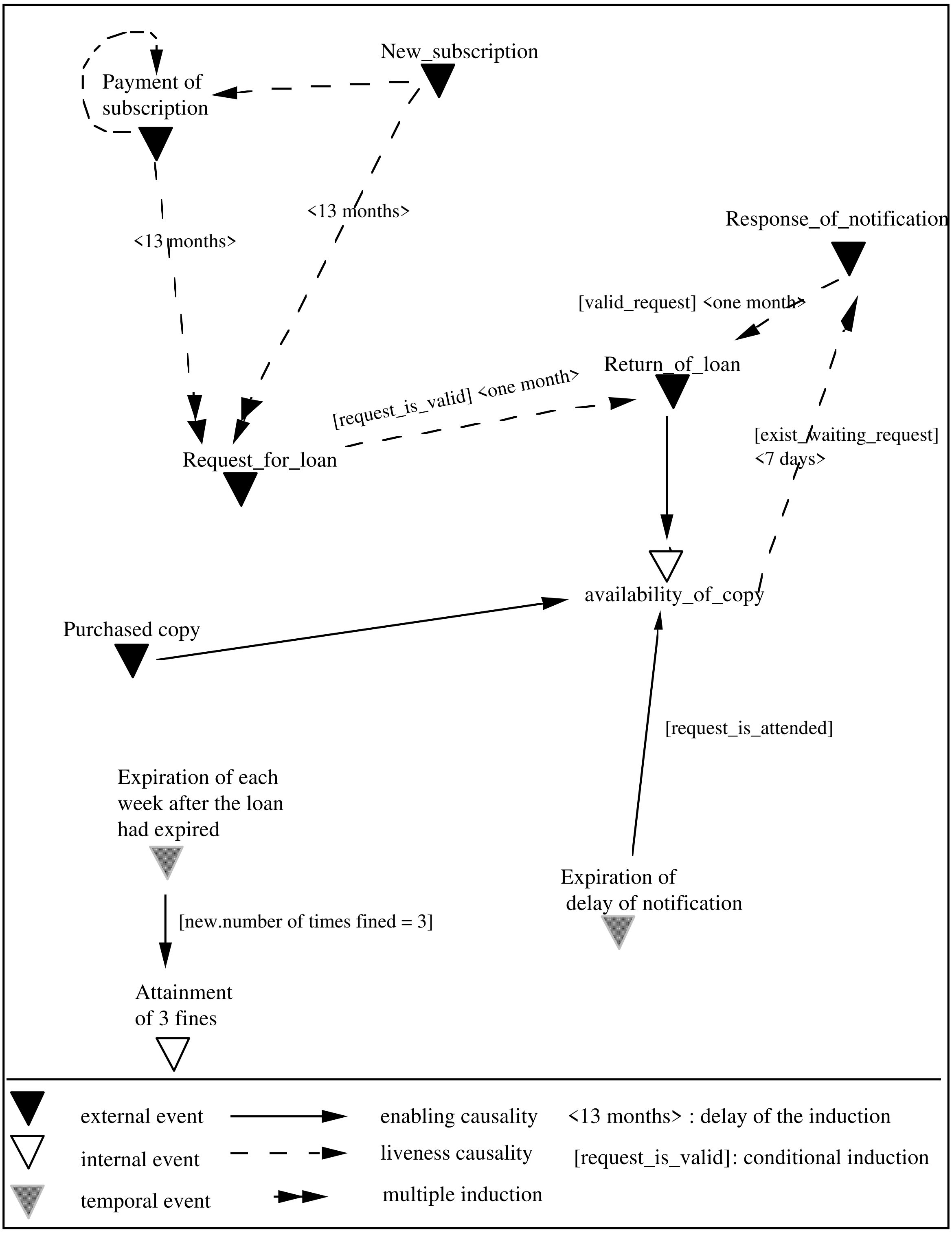


Figure 71: The event induction graph of the library system

3.3.4.4 The event concurrence graph

The event concurrence graph describes how events can be sequenced according to their concurrence.

An example of an event concurrence graph is shown in Figure 72.

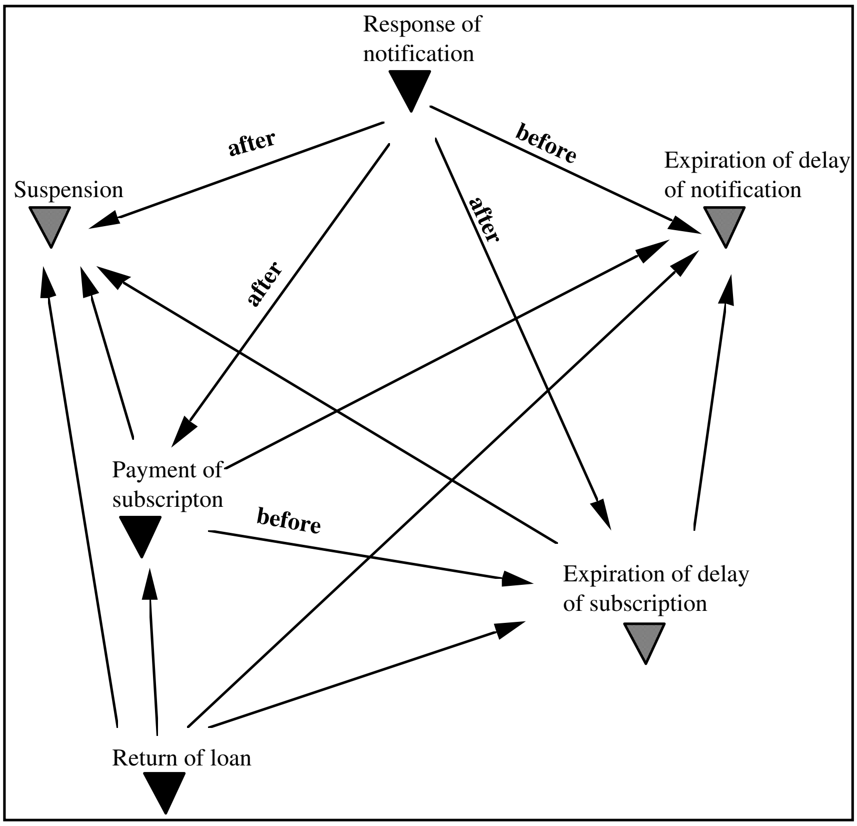


Figure 72: An event concurrence graph

3.3.4.5 The local class views

The local class views describe the static and the dynamic aspects of a class of any kind (object class, actor class and calendar class).

• The local view of an object class describes its attributes, its constraints, its operations or queries as well as its internal events. The local view of the object class *COPY* is graphically shown in Figure 73.

COPY
properties reference : STRING purchase date: DATE loan date : DATE state : COPY-STATE loans : inverse (zeromany, LOAN.copy) notices : inverse (zeromany, NOTICE.copy)
constraints unique(Copies,reference)
operations COPY purchase(c:structure(COPY)) reservation() availability() check out() wear out() loss() deletion()
queries boolean exist_copy_available()
internal-events $availability of copy

Figure 73: The local class view of the object class COPY

• The local view of an actor class is composed by its attributes, its external operations and its external events. The local view of the actor class *SUBSCRIBER* is shown graphically in Figure 74.

SUBSCRIBER (actor)
properties id of subscriber : STRING
external operations send_notice (n: NOTICE) send_reminder (r: REMINDER) send_waiting_request_message (r : REQUEST) send_accepted_request_message (r : REQUEST, c: COPY) send_cancelled_request_message (r: REQUEST) send_exclusion_message (s: SUBSCRIBER)
external events request for loan response of notification return of loan new subscription payment of subscription

Figure 74: The local class view of the actor class SUBSCRIBER

• The local view of a calendar class is only defined by its temporal events. The local view of the calendar class *CLOCK* is shown graphically in Figure 75.

CLOCK
temporal-events expiration of delay for payment of subscription expiration of waiting for request expiration of delay of notification moment of annual inventory expiration of delay for returning a book expiration of each week after the loan has expired suspension

Figure 75: The local class view of the calendar class CLOCK

3.3.5 Checking & Verification

The checking activity of the specification can be done manually or automatically. Validation is the activity aiming at assuring that the specification is valid by checking the correctness of the specification with respect to the model. The controls can be classified according to the following typology:

- *conformity controls* -- ensure that all concepts of the Model used are conformed to their definitions;
- *correctness controls* -- verify that the specification does not contain inconsistent or contradictory descriptions;
- *completeness controls* -- aim at verifying the completeness of the specification based on what has been specified;
- *reliability controls* -- detect the possible abnormal situations.

Validation controls are of the following nature:

- controls local to a class;
- controls on the static aspects of the specification;
- controls on the dynamic aspects of the specification;
- global controls on the specification.

In the following subsections, the controls of the first typology are used for the detailed description of the controls of the second typology. They are qualified by abbreviations, such as conformity control (fc), correctness control (cc), completeness control (pc) and reliability control (rc).

3.3.5.1 Controls local to a class

Rule 1 (fc): the names of all attributes are unique within a class.

Rule 2 (fc): the names of all operations are unique within an object class.

Rule 3 (fc): the names of all events are unique within a class.

Rule 4 (cc): every actor class contains at least one attribute.

Rule 5 (pc): every attribute is defined by a valid domain.

Rule 6 (pc): every association/aggregation property is defined by a valid object class.

Rule 7 (pc): an internal event predicate is associated to every internal event.

Rule 8 (pc): an external event predicate is associated to every external event.

Rule 9 (pc): a temporal event predicate is associated to every temporal event.

Rule 10 (cc): every object class contains at least a basic create operation.

Rule 11 (cc): every object class contains at least an (inherited) attribute.

Rule 12 (cc): every event triggers at least a basic operation.

Rule 13 (cc): every basic operation defined in an object class that is neither a component class nor a generalized class is triggered by at least an event.

Rule 14 (cc): every actor class contains at least an external operation or an external event.

Rule 15 (pc): every external operation defined in an actor class is triggered by at least an event.

Rule 16 (pc): every derived attribute defined in an object class is associated to a query operation with the same name (without argument).

3.3.5.2 Controls on the static aspects of the specification

Rule 17 (cc): an inheritance graph (i.e., graph whose nodes are object classes and whose arcs are inheritance links) must not contain a cycle.

Rule 18 (cc): an aggregation graph (i.e., graph whose nodes are object classes and whose arcs are aggregation links) must not contain a cycle.

Rule 19 (cc): a graph whose arcs are aggregation links and inheritance links must not contain a cycle.

Rule 20 (cc): in the case of multiple inheritance, the generalized classes inherited by an object class must be originated from a unique ancestor in the inheritance hierarchy.

Rule 21 (cc): an inheritance graph must not have a transitive path containing only an inheritance link.

Rule 22 (rc): a transitive path in a composition graph can be redundant (depending on the semantics of the aggregation links).

Rule 23 (rc): a transitive path in an association graph (i.e., graph whose nodes are object classes and whose arcs are associations) can be redundant (depending on the semantics of the associations).

3.3.5.3 Controls on the dynamic aspects of the specification

Rule 24 (cc): the basic operations of the same class triggered by an event must be exclusively executed or must be executed on different objects of the class.

Rule 25 (cc): the basic operations of the same class triggered by several dependent events (i.e., related by immediate induction relationship) must be exclusively executed or must be executed on different objects of the class.

Rule 26 (rc) : An internal event must be induced indirectly by at least one temporal or external event.

Rule 27 (pc) : Every triggering condition is triggered by at least one event.

Rule 28 (pc) : Every triggering factor is triggered by at least one event.

Rule 29 (fc): the names of all triggering conditions are unique within an application.

Rule 30 (fc): the names of all triggering factors are unique within an application.

3.3.5.4 Global controls on the specification

Rule 31 (fc): the names of all object classes or domains are unique within an application.

Rule 32 (fc): there is at least one global behavior graph (dynamic graph) within an application.

Rule 33 (fc): each class belongs to at least one global behavior graph.

Rule 34 (fc): the names of all actor classes are unique within an application.

Rule 35 (cc) : Every obliged induction between an event ev1 and event ev2 of the event induction graph is represented in the global behavior graph by an operation triggered by ev1 modifying an object of the object class whose ev2 is originated.

Rule 36 (cc) : Every obliged induction between an event ev1 and event ev2 with a condition has to verify one of these two cases :

- an operation triggered by ev1 modifying an object of the object class of the event ev2 is conditional.
- the predicate of the event ev2 is not systematically satisfied when the operation is executed, the condition embedded in the predicate must be verified.

3.4 The Specification language

A conceptual schema is identified by an application name. The definition part of each application is composed of :

object classes

actor classes

calendar classes

domains

triggering conditions

triggering factors

3.4.1 The application definition

The syntax of the definition of an application is given as follows :

<application-definition>::=

application <application-name> <specification-part>

<specification-part>::=

definition <specification-list> **enddefinition**

<specification-list> ::=

<object-class-definition-list>

<actor-class-definition-list>

[<calendar-class-definition>]

[<domain-definition-list>]

[<triggering-condition-definition-list>]

[<triggering-factor-definition-list>]

The object classes and actor classes are the mandatory elements of an application.

3.4.2 Domains

Domains contribute to the description of object states. A domain is described by a type that may be *atomic (core type)*, *enumerated*, *complex*, or the *type of other domain*. The concept of domain is not as important as the concept of class, but it must be introduced in the first place.

A domain is identified by the **domain** keyword in the type declarations of the specification. Its name should be unique among the names of all domains and object classes declared within the specification. The domain of an attribute may be a *core literal*, an *enumerated literal*, a *complex literal*, or *other domain literal*, depending on the definition of the domain associated. A domain definition describes a static aspect that contributes to the context descriptions of different objects, and hence, they are the components that contribute to the description of classes. The following presents some examples of domain declarations.

domain day : **{**"Monday", "Tuesday", "Wednesday", "Thursday», «Friday", "Saturday", "Sunday"**}**
domain hotel_number : **[**0 .. 359**]**
domain end_point : **integer**
domain position : **[**x : **structure (**end_point), y : **structure** (end_point)**]**
domain address : **[**street : string, city : string, postal_code : string, country : string**]**
domain periods : **set(**[period : **[**start : date, end : date**]])**

The domain *day* is defined by an enumeration of strings, the domain *hotel_number* by an interval of integers and the domain *end_point* by a core type, **integer**. The type definition of the domains, namely position and address, are generated by a **tuple** constructor, while the type definition of the domain periods is generated by both **tuple** constructor and **set** constructor. Each field in the domain position, is defined by a tuple whose structure will be copied from the data type definition (structure) of the domain *end_point*.

3.4.3 Classes

The three kinds of classes, namely object class, actor class and calendar class are respectively introduced by the keywords **object-class, actor-class** and **calendar-class** in the type declarations. The declarations of each class define other related concepts such as properties in object classes and actor classes, events in all kinds of classes.

3.4.3.1 Object Classes

An object class has a type definition associated to it. The type definition corresponds to the *intentional* description of the common characteristics of the objects in the class. The intentional description is local to the object class. It is possible to relate several class extents to an object class. An object class is identified by the **object-**class keyword in the type declarations. Its name should be unique among the names of all classes and domains declared within the application. Its declarations end with the **endobject-class** keyword. The syntax of object class is given as follows:

<object-class-definition> ::=

object-class <object-class-name> **':'** <inherited-class-list>
extents <persistent-collection-list>
[**clusters** <cluster-list>]
[**properties** <object-properties>]
[**constraints** <invariant-list>]
[**operations** <operation-list>]
[**queries** <operator-list>]
[**internal-events** <event-list>]
endobject-class

The simple example below shows the type definition schema of the object class *STUDENT*.

object-class STUDENT : PERSON **extents** Students
properties
student_number : integer
...
...
endobject-class

The object class *STUDENT* is inherited from the object class PERSON. In this case, all properties and operations of *PERSON* will be inherited by *STUDENT*. It has an attribute defined by the type *integer* and an extent named *Students*.

3.4.3.1.1 Properties

The attributes of an object class are described in the ***attributes*** part of its type definition schema. An attribute name should be unique among all attributes declared in the class. An attribute can be considered as a named function of an object that returns a value. The attribute value returned makes up a portion of the object state. It is possible to explicitly declare the inverse link of an association or an aggregation via the inverse keyword, in order to allow navigation to be specified in a more readable form.

The syntax for aggregation properties and association properties in the following subsections use the **aggr** constructor for describing the aggregation relationship between objects from two different object classes, the **asso** constructor for describing the association between objects.

3.4.3.1.1.1 Attributes

A domain attribute is an attribute that takes its value from a domain. Its type can be either a *built-in core type*, an *enumerated type*, a *complex type*, or *other domain name*. A domain attribute of an object class is called *derived attribute* if it contains the **derived** keyword in its declaration. It takes the same name as a query operation defined in the class. In this case, the usage of the derived attribute is equivalent to the assignment of the activation result of the corresponding query operation.

<u>Example 1</u>: A person is identified by an identification number, a name and a derived attribute named age which is associated to the service operation having the same name.

object-class PERSON **extents** Persons
  properties
    id_number : integer
    name : name
    age : integer **derived**
  queries
    integer age() **type object service**
endobject-class

3.4.3.1.1.2 Aggregation property

An aggregation property defined in an object class declares an aggregation link from this class to a related object class.

<u>Example 2</u>: A car is composed of an engine. It is assumed that the object class ENGINE exists.

object-class CAR **extents** Cars
  properties
    engine : **aggr(one,** ENGINE)
    ...
endobject-class

Example 3: An hotel is composed of a set of rooms, each room has an implicit inverse link to the hotel (**inverse** of HOTEL::rooms). It is assumed that the object class *ROOM* exists.

```
object-class HOTEL extents Hotels
    properties
        name : string
        category : string
        rooms : aggr(onemany , ROOM)
endobject-class
```

3.4.3.1.1.3 Association property

An association property defined in an object class declares an association link from this class to a related object class.

Example 4: An order line is associated to a product by the property ordered_product. The inverse role orderlines is described in the object class *PRODUCT*. An association property is described by the cardinality, the referenced object class and the inverse role. The inverse role is defined in the referenced class with its cardinality and the name of the association property.

```
object-class ORDERLINE extents Orderlines
    properties
        ordered_product : asso (one , PRODUCT, inverse of orderlines)
        quantity : integer
endobject-class

object-class PRODUCT extents Products
    properties
        orderlines : inverse  (zeromany , ORDERLINE::ordered_product)
        quantity : integer
endobject-class
```

3.4.3.1.2 Constraints

The constraints defined in the object class are of three kinds : attribute constraints, uniqueness constraints and inheritance constraints

3.4.3.1.2.1 Attribute Constraints

The example below describes that the number of wheels of a car must be equal to 4.

object-class CAR **extents** Cars

properties

wheels : **aggr(many,** WHEEL)

constraints

boolean wheel_constraint() **comment** number of wheels = 4

endobject-class

An attribute constraint may be defined on inherited attributes. For instance, if age is an attribute of the object class PERSON, it is possible to define an attribute constraint on age in the object class *STUDENT* that inherits from *PERSON*. This constraint is defined to be an invariant and applies only to students.

object-class STUDENT **:** PERSON **extents** Students

properties

student_number : integer

constraints

boolean age_constraint() **comment** age > 0

endobject-class

The assertion function named *age_constraint* defines the attribute constraint age > 0 that requires a person to have an age superior to zero.

A constraint can restrict an inherited attribute. For instance, the inherited attribute *rooms* in *LUXURY*_HOTEL takes its value from the specialized object in *LUXURY_ROOM* is a constraint specified as follows:

object-class LUXURY_ROOM **:** ROOM **extents** Luxury_rooms

...

endobject-class

object-class HOTEL **extents** Hotels

properties

rooms : **aggr(onemany,** ROOM)

...

endobject-class

object-class LUXURY_HOTEL **:** HOTEL **extents** Luxury_hotels

...

constraints

boolean room_constraint() **comment** each room r in Luxury_rooms

endobject-class

3.4.3.1.2.2 Uniqueness Constraints

The pre-defined function named *unique()* can be called to verify the uniqueness of one or several attributes, that may be defined by aggregation or association links. The first argument of the function is the name of the class or the name of the aggregation/association property having a many cardinality. The rest of the arguments are the list of attributes on which the uniqueness constraint will be imposed in the class or the collection defined by the aggregation/association property. In other words, the list of attribute values of objects must be unique within the class or the collection defined by an association with a many cardinality. The example below shows the declaration of a uniqueness constraint to so that all objects are unique on the pair of attributes (*country, registration_number*).

```
object-class CAR extents Cars
properties
    country : asso( one, COUNTRY, inverse of cars)
    registration_number : string
constraints
        unique( Cars, country, registration_number)
endobject-class
```

The following uniqueness constraint states that room numbers are unique for each hotel (but rooms of different hotels may have the same room number).

```
object-class HOTEL extents Hotels
properties
    hotel_name : string
    rooms : aggr(onemany, ROOM)
endobject-class

object-class ROOM extents Rooms
properties
    number : integer
constraints
        unique(self. inverse of HOTEL::rooms.rooms, number)
endobject-class
```

The path leading to the composite object in *HOTEL* can be accessed by specifying the keyword self followed by **inverse of** and the name of the aggregation property. The path ***self.inverse of*** *HOTEL::rooms.rooms* is specified to ensure uniqueness of the room numbers in the hotel.

When object classes are related to each other by an association link, such as the object class ORDERLINE related to each of the object classes *ORDER* and *PRODUCT*, it is possible to say that an orderline is identified by its order and its product as described in the following type definition of *ORDER*:

```
object-class ORDER extents Orders
    properties
```

orderlines : **inverse** (**onemany**, ORDERLINE::order)

endobject-class

object-class PRODUCT **extents** Products

properties

orderlines : **inverse** (**zeromany**, ORDERLINE::product)

endobject-class

object-class ORDERLINE **extents** Orderlines

properties

product : **asso(one,** PRODUCT, orderlines**)**

order : **asso(one,** ORDER, orderlines**)**

quantity : integer

constraints

unique(Orderlines, product, order**)**

endobject-class

3.4.3.1.2.3 Inheritance Constraints

The three pre-defined functions corresponding to the three types of inheritance constraints: partition constraint, disjunction constraint and covering constraint defined in the previous chapter can be called in the following manner:

partitioning(<object class name1>, ..., <object class nameN>**)**

disjoint(<object class name1>, ..., <object class nameN>**)**

covering(<object class name1>, ..., <object class nameN>**)**

3.4.3.1.2.3.1 Disjunction Constraints

The disjunction constraint defined in the type definition of the object class *VEHICLE* given below states that a vehicle cannot be at the same time a car and a truck, but it can be another type of vehicle.

object-class VEHICLE **extents** Vehicles

...

constraints

disjoint(CAR, TRUCK**)**

endobject-class

object-class CAR **:** VEHICLE **extents** Cars

...

endobject-class

object-class TRUCK **:** VEHICLE **extents** Trucks

...

endobject-class

3.4.3.1.2.3.2 Covering Constraints

The covering constraint defined in the type definition of the object class *PERSON* given below states that a person must be a student, a teacher, or both.

object-class PERSON **extents** Persons
...
constraints
covering(STUDENT, TEACHER**)**
endobject-class

object-class STUDENT **:** PERSON **extents** Students
...
endobject-class

object-class TEACHER **:** PERSON **extents** Teachers
...
endobject-class

3.4.3.1.2.3.3 Partition Constraints

The partition constraint defined in the type definition of the object class *PERSON* given below states that a person must be either a man or a woman, but not both.

object-class PERSON **extents** Persons
...
constraints
partitioning(MAN, WOMAN**)**
endobject-class

object-class MAN **:** PERSON **extents** Men
...
endobject-class

object-class WOMAN **:** PERSON **extents** Women
...
endobject-class

3.4.3.1.2.4 Clusters

A cluster is identified by a cluster name. For instance, the following type definition schema illustrates how vehicles can be grouped into two clusters, namely *vehicle_propulsion* and *vehicle_by_medium*.

object-class VEHICLE **extents** Vehicles
  clusters
    vehicle_propulsion(WIND_PROPELLED, MOTORIZED, MAN_POWERED)
    vehicle_by_medium(AIR_VEHICLE, WATER_VEHICLE, LAND_VEHICLE)
  ...
  constraints
    covering(vehicle_propulsion, WIND_PROPELLED, MOTORIZED, MAN_POWERD)
    covering(vehicle_by_medium, AIR_VEHICLE, WATER_VEHICLE, LAND_VEHICLE)
  ...
endobject-class

In this case, an inheritance constraint is associated to each cluster by including the cluster name as the first parameter of the function *covering()* for the inheritance constraint.

3.4.3.1.3 Operations

An operation of a given type is specified by an operation signature. Each operation signature defines the name of the operation, the names and types of any arguments and the type of any return value.

3.4.3.1.3.1 Basic Operations

Input values must be passed to a create operation for creating an object along with its attribute values. For an update operation, input values might be passed for replacing the old attribute values of an object. No input values need to be passed to a delete operation. When an object is deleted, all its attribute values and relationships with other objects are also deleted. The signature of a basic operation includes the operation type denoted by either **type create**, **type update** or **type delete**.

By default, the *primitive delete procedure*, *delete()*, follows the default rule for the deletion of a referenced object. The default rule stipulates that a referenced object can be deleted only when there isn't any association refers to it. The violation of the default rule causes the abortion of the transaction. To make sure that a referenced object does not have any other associations applied to it, control must be explicitly defined in a delete operation for verification before applying the primitive delete procedure. For instance, the type definition schema of the object class *ROOM* defines two separate delete operations for taking care of the deletion of an object of *ROOM*.

object-class ROOM **extents** Rooms

properties

number : integer

availabilities : **inverse** (**zeromany**, AVAILABILITY::room)

reservations : **inverse** (**zeromany**, RESERVATION::room)

operations

delete_room() **type delete comment** deletion of the room

control_delete_room() **type delete**

comment if the inverse links *availabilities* and *reservations* are empty

then deletion of the room

endobject-class

The delete operation *delete_room* applies the default rule for the deletion of an object of *ROOM*, i.e. deletion of the object only if no other association applied to it. The violation of the default rule will cause the abortion of the implicated transaction. On the other hand, the delete operation *control_delete_room* verifies that there isn't any association link associated to the object before deleting it. This avoids abortion of the transaction by integrating control in the delete operation for taking care of the object still referred by other objects.

3.4.3.1.3.2 Strong Coupling of Basic Operations implied by Aggregation Links

Some kind of specifications has been defined for taking into consideration the strong coupling of behavior implied by an aggregation link, i.e., the dependency of component objects on their composite object. Such specifications are defined in a declarative way on the signatures of the basic operations of composite classes by including the *aggr clause* right after the operation type (**type create** and **type delete**). The dependencies are specified as follows:

<operation name> **(...)** <operation type> **: aggr(cardinality,** <component class name>**::**<operation name>, ...)

The *aggr clause* can take a list of operations from different object classes. The list may include a specific cardinality for each operation, such as,

<operation name> **(...)** <operation type> **: aggr(one,** <component class name1>**::**<operation name1>, **many,** <component class name2>**::**<operation name2>,...)

Two cases may be summarized as follows:

- the invocation of the create operation of the composite class will automatically generate the invocation (or iterative invocation) of the create operation(s) for its component class(es);
- the invocation of the delete operation of the composite class will automatically generate the invocation (or iterative invocation) of the delete operation(s) for its component class(es).

The invocation of the create/delete operation of a component class will be iterative if the composite object is related to more than one component objects. In other words, the composite class and the component class are related with each other via an aggregation link. Our main aim in the example hereafter is to demonstrate how to define basic operations between a composite object and its component objects. The object classes *HOTEL* and ROOM are related to each other via the aggregation properties rooms defined in *HOTEL*. *HOTEL* is the composite class while ROOM is the component class. Some basic operations are defined in the following type definition schema of ROOM for creating an object of *ROOM* via *create_room*, for updating an object of *ROOM* via *update_room*, as well as for deleting an object of *ROOM* via *delete_room*.

object-class ROOM **extents** Rooms
  properties
  number : integer
operations
  ROOM create_room(num : integer) **type create comment** insertion of a room in rooms
  delete_room() **type delete comment** deletion of a room
  update_room(new_number : integer) **type update**
      comment update of the number of the room
endobject-class

object-class HOTEL **extents** Hotels
  properties
    name : string
    category : string
    rooms : **aggr(onemany,** ROOM**)**
  operations
    HOTEL create_hotel(name : string,
          category : string,
        rms : set(structure(ROOM)) **type create : aggr(many,**ROOM::create_room**)**
    delete_hotel() **type delete : aggr(many,** ROOM::delete_room**)**
    add_rooms(rooms : set(structure(ROOM))) **type update**
    delete_rooms(room_numbers : set(integer)) **type update**
endobject-class

The above example intends to show coupling of behavior between a composite object in *HOTEL* and related component objects in *ROOM* via the create operation (respectively delete operation). As a result of declaration of the *agg clause* right after the operation type of *create_hotel* (i.e., ***aggr(many,*** *ROOM::create_room*), the basic operation *create_room()* defined in *ROOM* will be automatically invoked in an iterative way for any creation performed by the basic operation *create_hotel()* of *HOTEL*.

3.4.3.1.3.3 Queries

Queries can be invoked in the trigger part of events. They can be seen as encapsulated functions returning a value. A class operation service is mentioned with a '$' mark before the name of the service. In the following example, there is a query called age defined in the object class *PERSON* for calculating the age of a person from his date of birth, and a query named *average_age* for calculating the average age of all the people. The operation *age()* is a query calculating the value for the derived attribute age.

```
object-class PERSON extents Persons
    properties
        name : string
        birth_date : date
        age : integer derived
    queries
    integer age()
    integer $average_age()
endobject-class
```

Notice that the object operation age() will be invoked in the following way via an object of PERSON represented by p:

```
p.age()
```

while the class operation *average_age()* which deals with the persistent set Persons in the implementation, will be invoked in the following way:

```
Persons.average_age()
```

Moreover, if an attribute of the type **derived** is declared in the class and possesses the same name as the service operation age(), the actual age of each person can be directly access via the attribute without activating the query operation, as this is implicitly done via the attribute having the same name as a query. The following query returns the earliest waiting request for a book from the object class *REQUEST*.

```
object-class REQUEST extents Request
    ...
    queries
        REQUEST $find_earliest_request(book : BOOK)
endobject-class
```

This query attached to *REQUEST* deals with the persistent set Requests and could be invoked as follows:

```
Requests.find_earliest_request(b)
```

where *b* is an object of *BOOK*.

A query may also deal with an object, in this case, it is invoked via the object. For instance, the query *find_request* is attached to COPY, from which a request for a copy of a book can be accessed.

```
object-class COPY extents Copies
    ...
    queries
        REQUEST find_request()
endobject-class
```

It can be invoked in the following way for retrieving the request of the copy c.

```
c.find_request()
```

3.4.3.2 Actor classes

An actor class has a type definition associated to it. The type definition of an actor class corresponds to the intentional description of actors belonging to the same class. An actor class is identified by the **actor-**class keyword in the type declarations. Its name should be unique among the names of actor classes of an application. Its declaration ends with the **endactor-class** keyword. The syntax of actor class is described as follows:

```
<actor-class-definition> ::=
    actor-class <actor-class-name>
        properties <object-properties>
        external operations <external-operation-list>
        external-event <external-event-list>
    endactor-class
```

The example below shows the type definition schema of the actor class *SUBSCRIBER*.

```
actor-class SUBSCRIBER
    properties
        id_subscriber : string
        subscriber_name : string
        ...
endactor-class
```

The attributes allows to characterize a subscriber when he interacts with the system.

3.4.3.2.1 external operations

An external operation is specified by its signature. Each operation signature defines the name of the operation, the names and types of any arguments. The arguments of this kind of operations represent information send by the system to the actor.

actor-class SUBSCRIBER
    properties
        id_subscriber : string
        subscriber_name : string
  external operations
      send_notice (n: NOTICE)
      send_reminder (r: REMINDER)
      send_waiting_request_message (r : REQUEST)
      send_accepted_request_message (r : REQUEST)
      send_cancelled_request_message (r: REQUEST)
      send_exclusion_message (s: SUBSCRIBER)
      send_suspension_message (s: SUBSCRIBER)

        ...
endactor-class

For instance, the external operation *SEND_NOTICE* permits to send a notice to the subscriber. The message is the argument of the operation.

3.4.3.2.2 External messages

An actor impacts the system through external events. An external event represents the arrival of a particular message which requires a reaction of the system. The external message is described by its structure. The example below shows the definition of the message attached to the event *RETURN_LOAN*.

actor-class SUBSCRIBER
    properties
    ...
    external-events
        return_loan
        message RETURN_LOAN
            s_family_name : **string**
            s_name : **string**
            copy_reference : **string**
        ...
endmessage-class

Three attributes *code_reference, s_family* and *s_name*, compose the message belonging to the external event *return_loan*. External events will be described separately in the next section.

An attribute of a message may be described by a domain type or a structure defining other type via the **structure** constructor. An attribute that takes its value from a domain is called a *domain attribute*. An attribute described by a structure of a type takes the data structure (data type definition) of that type, and it is referred to as *structure attribute*. The example below shows the definition of the message *PURCHASED_*COPY associated to the external event *PURCHASED_COPY* .

actor-class LIBRARIAN
  properties
  ...
  external-events
    purchased_copy
    message PURCHASED_COPY
      b : **structure**(BOOK)
      date put out for loan : **date**
  ...
 endactor-class

The structure type indicates that the b attribute defines the data type of the object class BOOK. Hence, the b attribute can be elaborated as,
b : [reference : **string,**
  title : **string,**
  edition date : **date,**
  last edition : **string,**
  price: **integer,**
  copies : **set** (COPY)**,**
  editor : EDITOR,
  authors : **set** (AUTHOR)**,**
  keywords : **set** (KEYWORD)**,]**

3.4.3.3 The calendar class

The calendar class has a type definition associated to temporal events. It is identified by the **calendar-class** keyword in the type declarations and its declaration ends with the **endcalendar-class** keyword. The syntax of calendar class is described as follows:

<calendar-class-definition> ::=
  calendar-class <calendar-class-name>

temporal-events <event-list>

endcalendar-class

Temporal events will be described separately in the next section. The example below shows the definition of the calendar class CLOCK, describing two temporal events, namely *each_week_after_expiration_of_loan* and *at_the_end_of_each_month*.

calendar-class CLOCK

temporal-events

each_week_after_expiration_of_loan

...

at_the_end_of_each_month

...

endcalendar-class

3.4.4 Events

An event represents either a state change of object, an arrival of message or an occurrence of time, depending on the type of the event. It is defined by an event predicate and a dynamic transition.

3.4.4.1 Event Predicates

The predicate is recognized by the keyword **predicate**. If the predicate is a boolean expression. If the predicate is true, an event will occur.

3.4.4.1.1 Internal Event Predicates

Internal event predicates are the event predicates of internal events. The event predicate of the internal event *availability of copy* is described by a predicate, for verifying if there is a copy available. Its description is given in the following type definition schema of *COPY*.

object-class COPY **extents** Copies

...

internal-events

availability of copy

predicate (not(old.state.available**) and new.state.**available**)**

...

endobject-class

Notice that the keywords **old** and **new** are normally used in internal event predicates. They refer respectively to the old object and the new object. The assertion verifies if the state of the old version of the object is not "available", and the state of the new version is "available". If the predicate is verified, the event will occur.

3.4.4.1.2 External Event Predicates

External event predicates are the event predicates of external events. The predicate in the external event is used for verifying message attributes.

For instance, the predicate of the event *RETURN_LOAN* verifies if the subscriber exists, the copy exists and the loan for this subscriber and that copy exists. If the predicate is false, the message is considered invalid and there is no occurrence of the external event.

3.4.4.1.3 Temporal Event Predicates

Temporal event predicates are the event predicates of temporal events. The temporal predicate is a temporal assertion which corresponds to a particular state change of the object *clock*. Three different kinds of temporal assertions are allowed : absolute time, relative time and periodic time.

3.4.4.1.4 Absolute time

The absolute time represents a particular time instant. For instance, *15th January 1995* is an absolute time.

3.4.4.1.5 Relative time

The relative time is a time instant which is expressed relatively to a temporal mark. The temporal mark could be a time occurrence of an event or a temporal attribute of a persistent object class.

For instance, ***today*** *= NOTICE::date of notification + 7* ***days*** is an example of relative time based on a temporal attribute. In this case, the temporal mark is the attribute *date of notification* of the object class *NOTICE*. For each value of this attribute in the *Notices* collection , the assertion is/was/will be true.

The same example can be described as follows :

today = COPY::availability of copy.time+ 7 days

In this example, the temporal mark is the time occurrence of the internal event *availability of* copy of the object class *COPY*. For each occurrence of the internal event, the temporal assertion is/was/will be true. These two examples are equivalent because the notice is created by the internal event *availability of copy* and the value of the attribute *date of notification* **is** today (time occurrence of the event).

3.4.4.1.6 Periodic time

The periodic time is a temporal assertion which becomes true when a certain period of time is expired from the last time the assertion was true. For instance, **each** day or **each** end_of_year are examples of periodic time. A periodic time is introduced by the each keyword.

3.4.4.2 Event Triggers

The direct mapping of the event trigger from the diagram can be expressed in two general forms for all kinds of events. In the case where no triggering factor is defined in the operation triggering, the statement of the operation triggering in the trigger part can be expressed in the following description:

trigger
    if (*triggering_condition(***context***)***) then**
            *query_operation(***context***)***.***basic_operation(...)*
    ...
endtrigger

If there is a triggering factor defined in the operation triggering, the statement of the operation triggering in the trigger part corresponds to the following description:

trigger
    if (*triggering_condition(***context***)***) then**
        foreach *object* **in** *triggering_factor(***context***)* **do**
            *query_operation(object)***.***basic_operation(...)*
    ...
endtrigger

The statements within the keywords **trigger** and **endtrigger** describe operation triggering. The activation of the triggering condition is described in the if clause by its invocation, *triggering_condition(***context***)*. The service request of the basic operation is represented by the invocation of the service operation, *query(**context**),* if the triggering factor is not defined in the operation triggering, and *query(object)*, if a triggering factor is defined. The result of this invocation returns the requested object. The invocation of the basic operation on the requested object (result of the service request) corresponds respectively to the two statements described as follows:

query(**context**).basic_operation(...)
query(object).basic_operation(...)

For each operation triggering, if the triggering condition is verified, the context will be passed to any triggering factor associated for the evaluation of the appropriate objects, for which the basic operation will be performed repeatedly. If there is no triggering factor associated, the context will be passed to the query operation of the basic operation whose operation type is either *update* or *delete*, for retrieving the appropriate object. A basic operation whose operation type is *create* does not have any query operation.

- **Triggers Corresponding to the Three Types of Events**

- The trigger part is integrated in the internal event *availability_of_copy* as:

object-class COPY **extents** Copies
properties
...
operations
COPY purchase(c:structure(COPY)) **type create**
reservation() **type update**

queries
boolean exist_copy_available() **type class service**

internal-events
availability of copy
predicate(not(old.state = "available"**) and (new.state** = "available"**))**
trigger
if exist_waiting_request(**context.inverse of** BOOK::copies) **then context**.reservation()
if exist_waiting_request(**context.inverse of** BOOK::copies) **then**
Requests.find_earliest_request(**context**.**inverse of** BOOK::copies).notification()
if exist_waiting_request(**context.inverse of** BOOK::copies) **then** Notices.creation()
if exist_waiting_request(**context.inverse of** BOOK::copies) **then context**.reservation()
if exist_waiting_request(**context.inverse of** BOOK::copies) **then**
Requests.find_earliest_request(**context**.**inverse of** BOOK::copies).subscriber.add to copies in-hand()
if exist_waiting_request(**context.inverse of** BOOK::copies) **then**
Requests.find_earliest_request(**context**.**inverse of** BOOK::copies).subscriber.
send_notice(**context.**notice)

```
        if exist_waiting_request(context.inverse of BOOK::copies) then
            Librarian.send_notice(context.notice)
        endtrigger
endobject-class
```

If the internal event occurs, it is said that the object *COPY* has been ascertained by the event. This object corresponds to the context of the event and it is the available copy. The context of the event is denoted by the context keyword. The context will be passed to the triggering condition *exists_waiting_request* for determining if there exists a waiting request

If the triggering condition is verified, the context will be passed to the service operation *find_earliest_request*, defined in *REQUEST*. It is the query operation for the basic operation accept, defined in *REQUEST*. It provides the service of retrieving the earliest waiting request for the basic operation accept. Since the basic operation reservation, defined in *COPY*, reserves the available copy and will be performed on the object that is the context itself, it does not need to have a service operation. The basic operation *creation*, creates an object of *NOTICE*, so there is no service operation associated to it.

In the example, different ways have been shown for the activation of basic operations, for instance, the basic operation reservation is invoked on the object that is the context itself, the basic operation notification is preceded by the invocation of the service operation described by *Requests.find_earliest_request(**context.inverse of** BOOK::copies)* and the basic operation creation is preceded by the persistent set named *Notices*.

- The context of an external event is the message coming from the environment.

Let us consider the external event *new_person*, defined in the message class *NEW_PERSON,* for creating a new person in the IS. For instance, the external event *new_person,* is defined by the following specification:

```
external-event  new_person
    message new_person
     name : string
     address : structure(ADDRESS)
    predicate ( the person does not exist)
    trigger
        Persons.create(context.name, context.address)
    endtrigger
```

The predicate is ascertained if there is an arrival of a valid message, by verifying that there isn't any object with the same name as the context already exists in the persistent set named *Persons* The context of the external event is the message just intercepted. If the predicate is verified, the trigger of the event will be activated. The activation begins by passing the context of the event to the basic operation create defined in the object class *PERSON* for invoking the creation of an object corresponding to the person in the persistent set Persons.

3.4.4.3 Triggering conditions

Like an operation, a triggering condition has a declaration part describing its signature. A triggering condition can be seen as a non-encapsulated function returning a boolean. Its name should be unique among the names of all triggering conditions defined in the specification. The triggering condition defined below determines if there exists some waiting requests of a book made by the subscribers who have less than three copies in-hand.

triggering-condition
  boolean exists_waiting_request(c : COPY)
 comment does there exist a request waiting for the book became available which can be satisfied ?
endtriggering-condition

3.4.4.4 Triggering factors

Like the triggering conditions, a triggering factor has a declaration part describing its signature. It can also be seen as a non-encapsulated function returning the set of objects on which will be applied the basic operation. Its name should be unique among the names of all triggering factors defined in the specification. To see how a triggering factor can be used in the trigger part, consider an example in which a decision can be activated for deleting those copies of books that have been declared as worn-out. With this, we consider the description of the following external event named *decision_for_worn_out_copies* attached to the message *D_WORN_OUT_COPIES*.

decision_for_worn_out_copies
  message D_WORN_OUT_COPIES
    copy_refs : **set(**string**)**
  predicate (the copies must exist)
  trigger
    foreach c **in** set_of_for_worn_out_copies(**context**) **do** c.wear out()
  endtrigger

The predicate determines if all reference labels of the books in the message correspond to those found in the object base. If it is verified, the context will be passed to the triggering factor named *set_of_worn_out_copies* for retrieving the set of the appropriate copies to be deleted. The declaration and implementation of the triggering factor *set_of_worn_out_copies* is given as follows:

triggering-factor
  set(COPY) set_of_worn_out_copies(m : D_WORN_OUT_COPIES)
endtriggering-factor

Notice that the **foreach** statement retrieves each object from the set of worn-out copies returned by the triggering factor and passes it to the basic operation *wear_out*.

3.4.5 Event Induction Graph

The event induction graph is expressed in the event definition. The syntax is declared as follows :

<event-name>
[**induction** <induction-list>]
predicate '(' <text> ')'
<trigger>

3.4.5.1 obligation

The obliged induction allows to characterize if an event must induce the occurrence of other events. The following example expresses that the external event *RETURN OF LOAN* must induce an occurrence of the internal event *AVAILABILITY OF COPY*.

induction

Loans.find_loan(**context**).copy::availability of copy()

The variable **context** is the execution context of the event *RETURN OF LOAN*. The target event is specified with the returned copy which is the execution context of the internal event *AVAILABILITY OF COPY*.

The conditional induction can be expressed as follows :

induction

IF request_is_attended (**context**) **context**.copy.availability of copy()

If the condition *request_is_attended* is true, the temporal event *expiration of delay of notification* must induce an occurrence of the internal event *AVAILABILITY OF COPY*.

3.4.5.2 permission

The possible induction allows to characterize if an event may or should induce an occurrence of other events. For instance, the event *NEW_SUBSCRIPTION* and *PAYMENT_OF_SUBSCRIPTION* may induce several occurrences of the external event *REQUEST_FOR_LOAN* within one year.

induction
with delay 1 year **more** **self**.request_for_loan ()

The induction can concern several occurrences of the target event, it is mentioned with the **more** keyword. The strict delay of the induction link is mentioned after the keyword **with delay** The induced event concerns the same actor, it is why the keyword **self** is used.

3.4.6 Event Concurrence Graph

The concurrence between events is described in the event definition. Each event has a list of concurrent events. If a priority is required the keyword **before** or **after** can be specified.

3.4.6.1 concurrent event

The concurrence between events is expressed through concurrent link defined in each event description. For instance, the *REQUEST_FOR_LOAN* and *RETURN OFLOAN* events are concurrents. The description is as follows :

request_for_loan
...
concurrent events
after return of loan

This example expresses that an occurrence of the *REQUEST_FOR_LOAN* event has to be performed after the occurrence of the *RETURN OF LOAN* event.

3.5 REFERENCES

[Hughes68] G. E. Hughes, M. J. Cresswell, “An Introduction to Modal Logic”, Methuen, 1968.

[Olle91] T. W. Olle, J. Hagelstein, I. G. Macdonald, C. Rolland, H. G. Sol, F. J. M. V. Assche, A. A. Verrijn-Stuart, “Information Systems Methodologies”, Addison-Wesley, 1991.

[Rolland82] C. Rolland, C. Richard, “The Remora Methodology for Information Systems Design and Management”, IFIP Conference on Comparative Review of Information Systems Design Methodologies, North-Holland, 1982.

[Rolland88] C. Rolland, O. Foucaut, G. Benci, “Conception des Systèmes d’Information”, EYROLLES, 1988.

[Wieringa89] R. Wieringa, J-J. Meyer, H. Weigand, “Specifying Dynamic and Deontic Integrity Constraints”, Data & Knowledge Engineering 4, North-Holland. 1988.

3.6 ANNEX I

Syntax of the specification language

<u>Lexical conventions</u> :

<u><comment></u> :

- comment which can appear anywhere :

/* <text> ***/**

- comment which can appear only between atomics OOPL :

// <text> **end-of-line**

<<u>identifier</u>> : an identifier is an arbitrary long sequence of letters or digits. The first character must be a letter; the underscore is counted as a letter. Upper and lower case letters are different. All characters are significant. The identifier with a double underscore is reserved for use by implementation and standard libraries should be avoided by users.

<<u>keywords</u>> :

single character : **^ * ( ) - + = { } [ ] / ; ' : " < > , . **

character combinations : **<= >= := <> ::**

<<u>literals</u>> :

<char-constant> : a character constant is one or more characters enclosed in single quotes : 'a',

<string-constant> : a string constant is a sequence of characters surrounded by double quotes : "assss",

<integer-constant> : a sequence of digits which is in base ten : 100

<real-constant> : consists of an integer part a decimal point and a fractional part

3.6.1 Application definition

<application-definition>::=

application <application-name> <specification-part>

<specification-part>::=

definition <specification-list> **enddefinition**

<specification-list> ::=

<object-class-definition-list>

<actor-class-definition-list>

[<calendar-class-definition>]

[<domain-definition-list>]

[<triggering-condition-definition-list>]

[<triggering-factor-definition-list>]

<domain-definition-list> ::=

<domain>

| <domain-definition-list> <domain>

<object-class-definition-list> ::=

<object-class-definition>

| <object-class-definition-list> <object-class-definition>

<actor-class-definition-list> ::=

<actor-class-definition>

| <actor-class-definition-list> <actor-class-definition>

<triggering-condition-definition-list> ::=

<condition-definition>

| <triggering-condition-definition-list> <condition-definition>

<condition-definition> ::=

triggering-condition

<boolean-fonction-signature>

comment <text>

endtriggering-condition

<triggering-factor-definition-list> ::=

<factor-definition>

| <triggering-factor-definition-list> <factor-definition>

<factor-definition> ::=

triggering-factor

<factor-signature>

comment <text>

endtriggering-factor

3.6.2 Domain definition

<domain>::=

domain <domain-name> '**:**' <domain-definition>

[**operations** <operation-list>] **enddomain**

<domain-name>::=

<identifier>

<domain-definition>::=

<enumerated-domain-type>

| <domain-elaborated-type-specifier>

<domain-elaborated-type-specifier> ::=

<domain-final-type>

| <constructor> '(' <domain-elaborated-type-specifier> ')'

| '[' <domain-field-specifier-list> ']'

<domain-final-type> ::=

<simple-type-name>

| <domain-name>

| '**[**' <domain-field-specifier-list> ']'

<constructor> ::=

set | **bag** | **list** | **multilist**

<domain-field-specifier-list> ::=

<domain-field-specifier>

| <domain-field-specifier-list> '**,**' <domain-field-specifier>

<domain-field-specifier> ::=

<field-name> **':'** <domain-definition>

<enumerated-domain-type>::=

'{' <string-constant-list> '}'

| '[' <integer-constant> '**..**' <integer-constant> ']'

| '[' <real-constant> '**..**' <real-constant> ']'

| '[' <char-constant> '**..**' <char-constant> ']'

| '[' <date-constant> '**..**' <date-constant> ']'

<string-constant-list>::=

<string-constant>

| <string-constant-list> '**,**' <string-constant>

3.6.3 Declarations of object class

<object-class-definition> ::=

object-class <object-class-name> '**:**' <inherited-class-list>

extents <persistent-collection-list>

[**clusters** <cluster-list>]

[**properties** <object-properties>]

[**constraints** <invariant-list>]

[**operations** <operation-list>]

[**queries** <operator-list>]

[**internal-events** <event-list>]

endobject-class

<persistent-collection-list>::=

<persistent-collection-name>

| <persistent-collection-list> '**,**' <persistent-collection-name>

<inherited-class-list>::=

<object-class-name>

| <inherited-class-list> '**,**' <object-class-name>

<cluster-list>::=

<cluster>

| <cluster-list> <cluster>

<cluster>::=

<cluster-name> '(' <object-class-list> ')'

<cluster-name> ::=

<identifier>

<object-class-list>::=

<object-class-name>

| <object-class-list> '**,**' <object-class-name>

<invariant-list>::=

<invariant>

|<invariant-list> <invariant>

<invariant>::=

<boolean-fonction-signature>

| **unique** '(' <expression-list> ')'

| **partitioning** '(' <expression1-list> ')'

| **covering** '(' <expression1-list> ')'

| **disjoint** '(' <expression1-list> ')'

<expression1-list>::=

<expression1>

| <expresion1-list> <expression1>

<expression1>::=

<expression>

| <cluster-name>

<operation-list>::=

<operation>

| <operation-list> <operation>

<operation> ::=

<signature-operation> [**':' aggr '('**<operation-reference-list> **')'**]

[**':'** <operation-reference-list>] [**comment** <text>]

<signature-operation> ::=

<fonction-signature>

| <procedure-signature>

<cardinality_agg> ::=

one | many

<operation-reference-list> ::=

<operation-spec>

| <operation-reference-list> **','** <operation-spec>

<operation-spec> ::=

<cardinality_agg> **','** <operation-reference>

<operation-reference> ::=

<operation-name>

| <object-class-name> **'::'** <operation-name>

<operator-list> ::=

<function-signature>

| <operator-list> <function-signature>

<event-list> ::=

<event>

| <event-list> <event>

3.6.3.1 Attributes of object class

<object-properties>::=

<attribute-definition>

| <attributes> <attribute-definition>

<attribute-definition> ::=

<attribute-name> '**:**' <attribute-type-specifier> ['=' <default_value>]

<default_value> ::= <constant>

<attribute-type-specifier> ::=

<attribute-final-type> [**derived**]

<attribute-final-type> ::=

<simple-type-name>

| <domain-name>

| **aggr** '(' [<cardinality> '**,**'] <object-class-name> ')'

| **asso** '(' [<cardinality> '**,**'] <object-class-name> [**',' inverse of** <attribute-name>] ')'

| **inverse** '(' [<cardinality> '**,**'] <object-class-name> '**::**' <attribute-name> ')'

<cardinality> ::=

one | zeroone | many | zeromany

3.6.3.2 Function and procedure signatures

<fonction-signature> ::=

<type-specifier> <fonction-name> '(' [<argument-list>] ')' [**type create**] **|**

<type-specifier> **$** <fonction-name> '(' [<argument-list>] ')'

<procedure-signature> ::=

<operation-name> '(' [<argument-list>] ')' [**type update | type delete**]

<boolean-fonction-signature> ::=

boolean <boolean-fonction-name> '(' [<argument-list>] ')'

<factor-signature> ::=

<constructor> '(' <type-specifier> ')' <factor-name> '(' [<argument-list>] ')'

<fonction-name> ::=

<operation-name>

| <conversion-function-name>

| <operator-function-name>

<boolean-fonction-name> ::=

<invariant-name>

<condition-name>

<argument-list> ::=

<argument-declaration>

| <argument-declaration> '**,**' <argument-list>

<argument-declaration> ::=

[<variable-name> '**:**'] <type-specifier> ['**:=**' <expression>]

<operation-name> ::=

<identifier>

<factor-name> ::=

<identifier>

<invariant-name> ::=

<identifier>

<condition-name> ::=

<identifier>

3.6.3.3 Events

<event> ::=

<event-name>

[**induction** <induction-list>]

[**concurrent events** <concurrent-list>]

predicate '(' <text> ')'

<trigger>

<event-name> ::=

<identifier>

<trigger> ::=

trigger <statement-list> **endtrigger**

<statement-list> ::=

<statement>

| <statement-list> <statement>

<statement> ::=

[**IF** <call-condition> **THEN**] [**FOREACH** <variable-name> **in** <call-factor>] <call-operation>

3.6.4 Declarations of actor class and calendar class

<actor-class-definition> ::=

actor-class <actor-class-name>

properties <object-properties>

external operations <external-operation-list>

external-event <external-event-list>

endactor-class

<actor-class-name> ::=

<identifier>

<external-event-list> ::=

<external-event>

| <external-event-list> <external-event>

<external-event> ::=

<event-name>

message <message-name> <message-attributes>

[**induction** <induction-list>]

[**concurrent events** <concurrent-list>]

predicate '(' <text> ')'

<trigger>

<external-operation-list>

<external-operation>

| <external-operation-list> <external-operation>

<external-operation> ::=

<operation-name> '(' [<argument-list>] ')'

<calendar-class-definition> ::=

calendar-class <calendar-class-name>

temporal-events <event-list>

endcalendar-class

<calendar-class-name> ::=

<identifier>

3.6.4.1 Declaration of a message of an external event

<message-attributes>::=

<message-attribute-definition>

| <message-attributes> <message-attribute-definition>

<message-attribute-definition> ::=

<attribute-name> '**:**' <message-type-specifier>

<message-type-specifier> ::=

<message-final-type>

| <constructor> '(' <message-type-specifier> ')'

<message-final-type> :

<simple-type-name>

| **structure** '(' <OODL-type-name> ')' // O*-type-name : object class or domain

| '[' <message-field-specifier-list> ']'

<message-field-specifier-list> ::=

<message-field-specifier>

| <message-field-specifier-list> '**,**' <message-field-specifier>

<message-field-specifier> :

<field-name> '**:**' <message-type-specifier>

3.6.5 Call of conditions, factors and operations

<call-condition> ::=<condition-name> '**(**' [<expression-list>] '**)**'

<call-factor> ::=<factor-name> '**(**' [<expression-list>] '**)**'

<call-operation> ::= <procedure-expression> '**(**' [<expression-list>] '**)**'

<procedure-expression> ::=

<name2>

| <object-expression> '**.**' <name2>

<object-expression> ::=

<variable-name>

| <object-expression> '**.**' <name1>

| <object-expression> **inverse of** <name1>

| <procedure-expression> '**(**' [<expression-list>] '**)**'

<name1> ::= // name of attribute for an object class

<attribute-name>

| <qualified-class-name> '**::**' <name1>

<name2> ::= // name of an encapsulated function or a procedure

<operation-name>

| <operator-function-name>

| <conversion-function-name>

| <qualified-class-name> '**::**' <name2>

<expression-list> ::=

<expression>

| <expression-list> '**,**' <expression>

<expression> ::=

<primary-expression>

| <postfix-expression>

<postfix-expression> ::=

<final-expression>

| <direct-fonction-name> '**(**' [<expression-list>] '**)**'

| <postfix-expression> '**.**' <fonc-name> '**(**' [<expression-list>] '**)**'

| <postfix-expression> **'.'** <name>

| <postfix-expression> **inverse of** <name1>**'.'** <object-class-name>

<final-expression> ::=

<variable-name>

| <persistent-collection-name>

| context | old | new | self

<primary-expression> ::=

<literal>

| '('<expression> **')'**

<literal> ::=

<integer-constant>

| <character-constant>

| <symbol-constant>

| <date-constant>

| <floating-constant>

| <string-constant>

| <boolean-constant>

| <reference-constant>

<boolean-constant> ::=

true | false

<reference-constant> ::=

nil

3.6.6 Event induction graph & Event concurrence graph

<induction-list>::=

<induction-link>

| <induction-list> <induction-link>

<concurrent-list>::=

<concurrent-link>

| <concurrent-list> <concurrent-link>

<induction-link>::=<modality> [IF <call-condition>] [**with delay** <quantity-of-time>] [<cardinality>] <call-event>

<modality> ::= **may** | **must**

<cardinality> ::= **one** | **more**

<concurrent-link>::= [**before** | **after**] <event-name>

<call-event>::= <proc-expression> '**(**' [<expression-list>] '**)**'

<proc-expression> ::=

<namef>

| <object-expression> '**.**' <namef>

<object-expression> ::=

<variable-name>

| **CLOCK**

| <object-expression> '**.**' <nameatt>

| <object-expression> **inverse of** <nameatt> '**.**' <object-class-name>

| <proc-expression> '**(**' [<expression-list>] '**)**'

<nameatt> ::= // name of attribute for an object class

<attribute-name>

| <qualified-class-name> '**::**' <nameatt>

<namef> ::= // name of an encapsulated function or a procedure

<event-name>

| <qualified-class-name> '**::**' <namef>

3.7 ANNEX II

CASE STUDY OF THE LIBRARY SYSTEM

In this annex, a case study on the functioning of the library system will be conducted to demonstrate how to define an IS as a representation of the real world library system.

3.7.1 Management of the library system

The text below shows an example of the functioning of a library system, in which the purpose of the system, the organizational choices and the management rules have been described separately.

3.7.1.1 The Statements

The aim of the project is to organize the functioning of a library stressing on an automated IS. The IS should allow a coherent functioning of the library and inform the librarian of the present or former state of this functioning, bringing with it, for example, information on:

- the inventory of books by subject, or author, or year of entry or keywords;
- the inventory of books borrowed at a given moment;
- the inventory of present or former subscribers, or for a given period;
- the list of overdue books;
- the books borrowed at a given moment, by a given subscriber;
- the frequency of the reading of the books;
- the average duration of the waiting for a book;
- the eagerness with which readers come to look for an available book;
- the proportion of subscribers who have gone past the period of loan allowed by the rules ...

The functioning of the library stresses on a set of organizational choices and of management rules, that can be set down as follows after discussion with the managers.

3.7.1.2 The Organizational Choices

The library consists of a collection of books, each with a unique reference. There may be several copies of the same book in the library. In the actual management, a card is created for each new book. On this card are written the reference number of the book, the name of the author(s), the date of edition, the reference number of the last edition, a summary in the form of a list of keywords, and the number of copies for loan with the date they are put out for loan. This card is updated for any purchase of new copies or any deletion of copies. Each subscriber of the library has a card on which the librarian writes down the date of each loan, the reference of the book borrowed, and the date of return, together with the reference of the book.

The librarian has, at his disposition, two other cards:

- the card for requests of loans that could not be satisfied immediately and that are on the waiting list;
- the card for notifications of the subscribers, that inform them that they can come and take from the library the book that they have been waiting for and that is available from now on.

3.7.1.3 The Rules of Management

Below is a list of the principal rules of the functioning of the library:

- To subscribe, a subscription must be paid and it is renewable every year. The period of subscription is 12 months from the date of payment of the subscription.
- The maximum delay allowed for the payment of a renewed subscription is a month from the normal date of this renewed subscription.
- The subscriber who has not paid his subscription a month after the normal delay cannot borrow any book. It is said that he is suspended.
- A subscriber can, at the most, borrow 3 books at the same time.
- Any book on loan must be returned to the library, at the latest, at the end of the delay of a month for the loan.
- The requests for loan non-satisfied are placed on the waiting list, if the subscriber wishes it, and the requests would be satisfied according to the chronological order of their registrations.
- When a book on the waiting list demanded by a subscriber becomes available, he is informed by notice. He has a week to come and to take it. At the end of this delay, the book will no longer be reserved for him.
- The subscriber who has not returned the books borrowed within the normal delay period will be reminded one week after the expiring date of return, and will be fined at the end of two weeks of delay. The fine is equivalent to 20% of the price of the book. Each additional week of delay gives rise to a new reminder and to a fine of the same amount.
- A subscriber who has not returned the borrowed book to the library two weeks after the allowable period of a month will be suspended from borrowing any book from the library. If he has been fined thrice for the same loan, he can no longer benefit from the services of the library. It is said that he is excluded.
- A request on the waiting list for 3 months should be canceled.
- At the moment of the annual inventory, the librarian can decide to delete the copies of books too worn out and proceed to request new ones. He can decide to purchase additional copies of books much in demand.

3.7.2 The various object classes identified for the library application schema are:

1. *BOOK* - a class of all books kept in the library.
2. *COPY* - a class of all copies of books.
3. *KEYWORD* - a class of all keywords used in the books.
4. *PERSON* - a class of all people relevant for the management of the library system.
5. *SUBSCRIBER* - a class which is a specialization of *PERSON* for representing all subscribers who borrow books.
6. *AUTHOR* - a class which is a specialization of *PERSON* for representing all authors of books.
7. *EDITOR* - a class which is a specialization of *PERSON* for representing all editors of books.

8. *LOAN* - a class grouping all loans of copies requested by the subscribers.
9. *REQUEST* - a class grouping all requests of books.
10. *NOTICE* - a class grouping all notices sent to the subscribers whose requests for loan have been put into waiting state.
11. *REMINDER* - a class grouping all reminders sent to the subscribers who have been fined for not returning the borrowed books after the allowable period.

3.7.3 Global Structure view

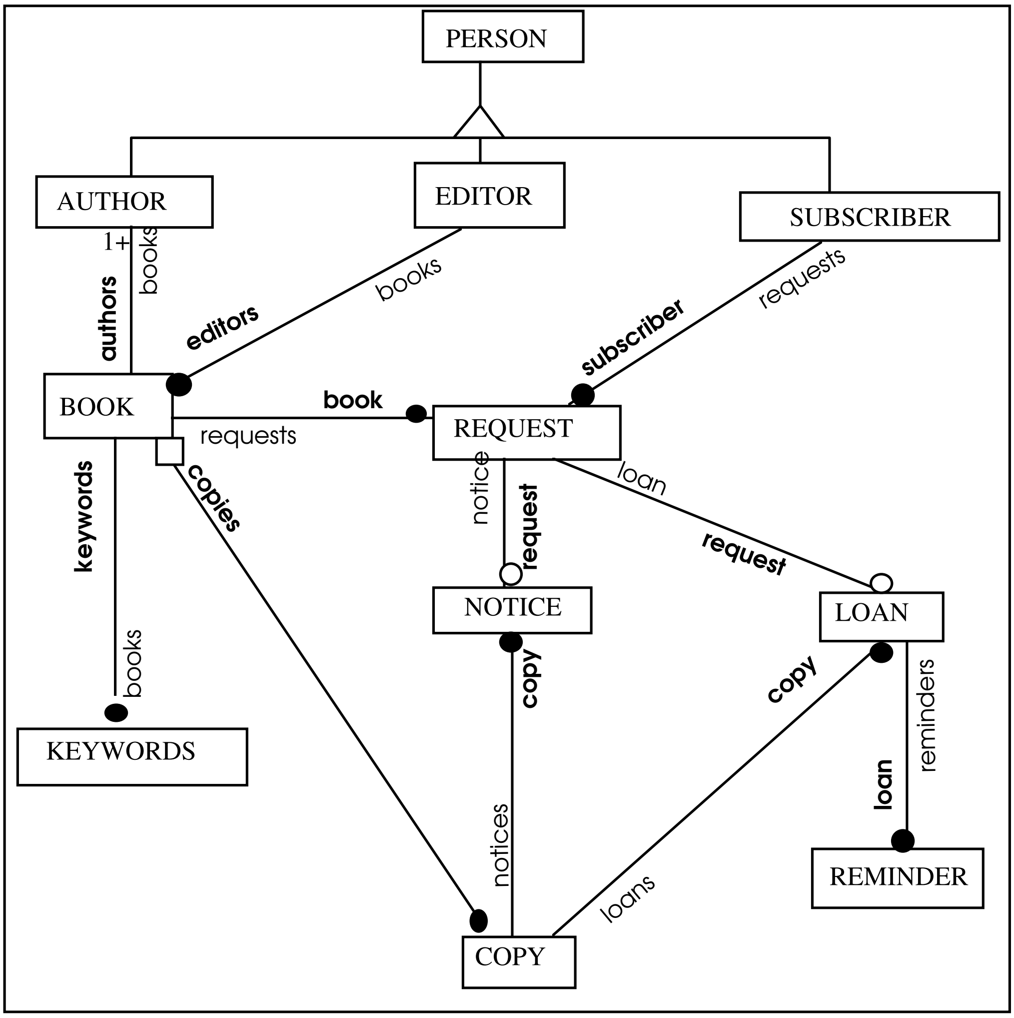


Figure 76: Global Structure View

3.7.4 Global behavior view

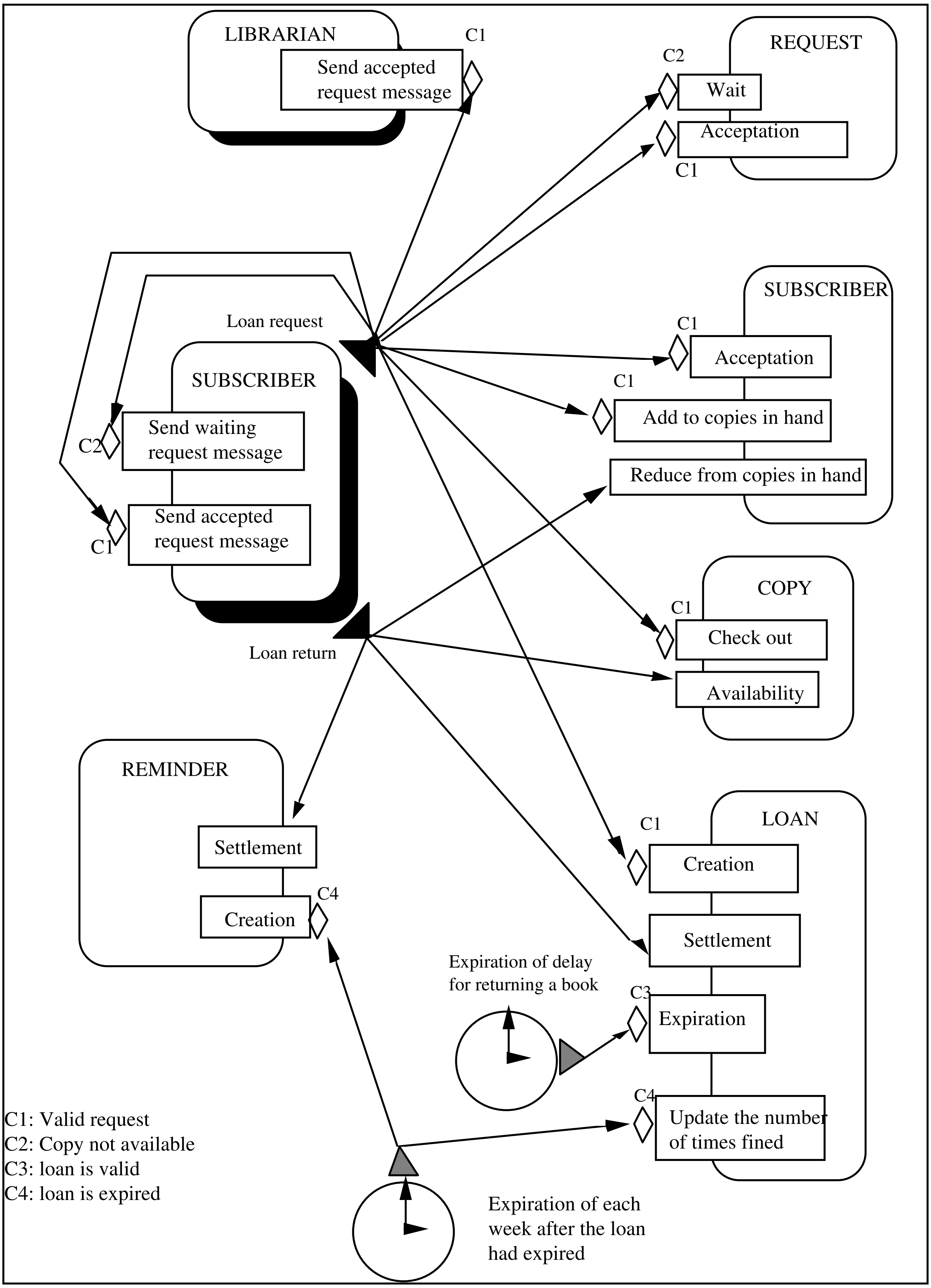


Figure 77 : Global behavior view (part 1)

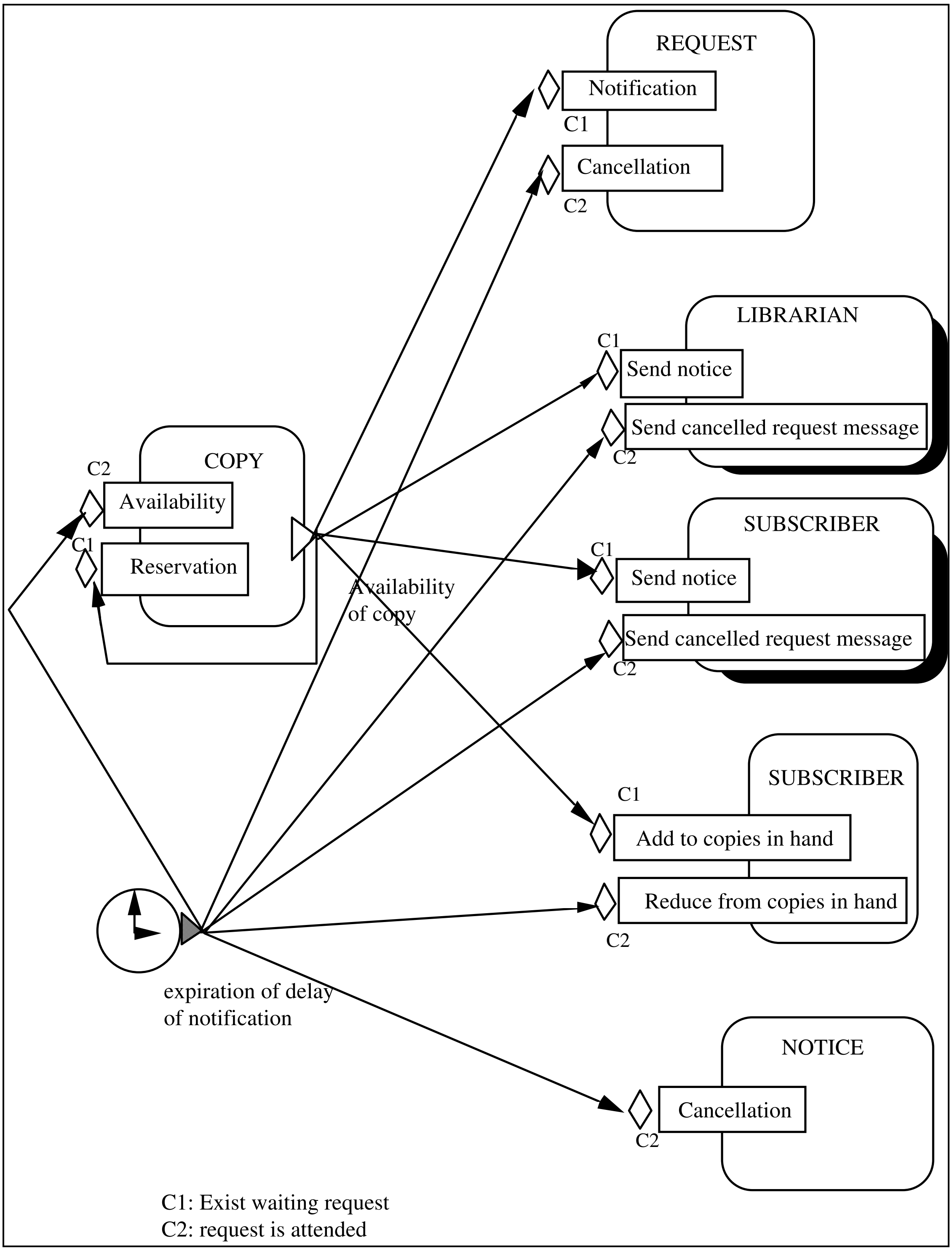


Figure 78 : Global behavior view (part 2)

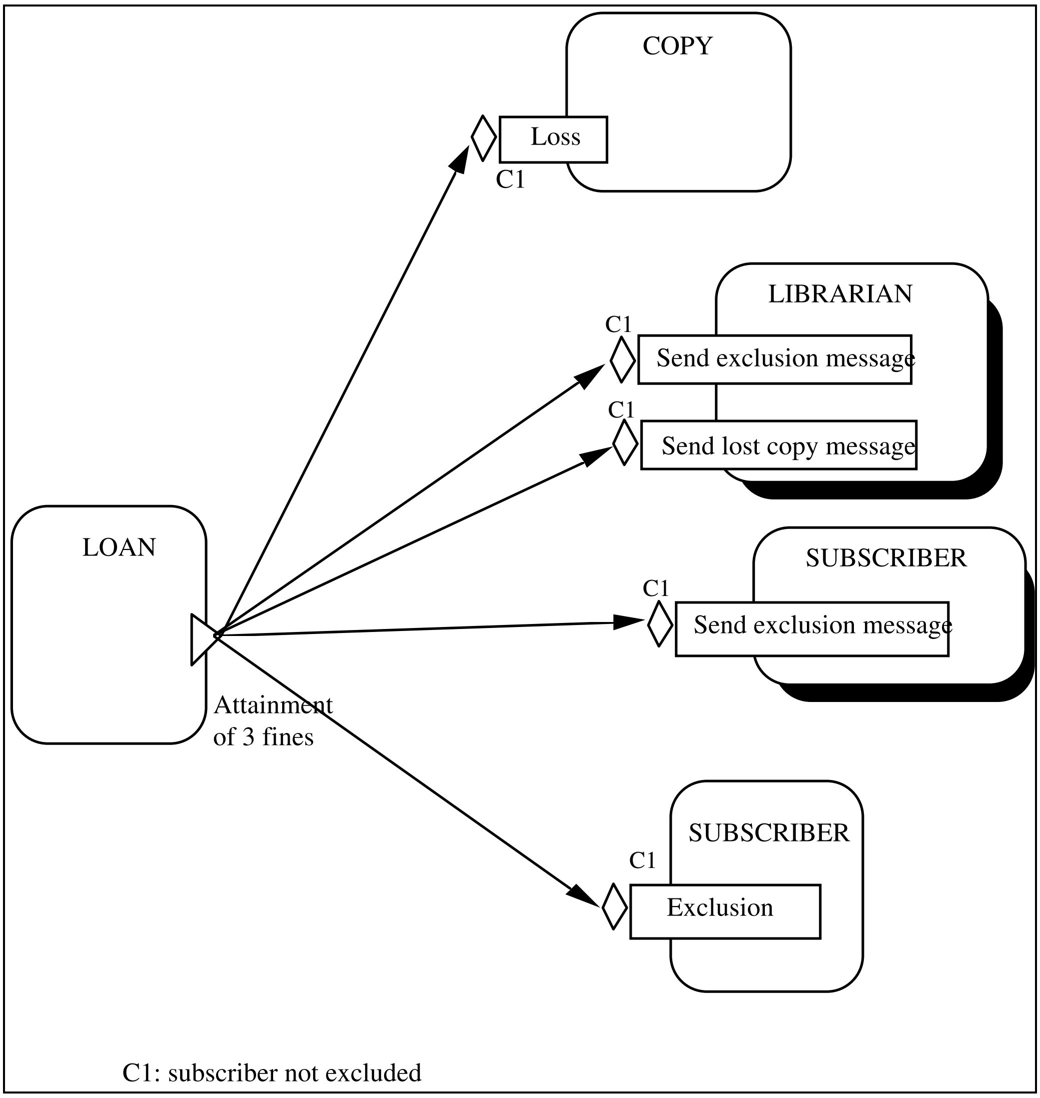


Figure 79 : Global behavior view (part 3)

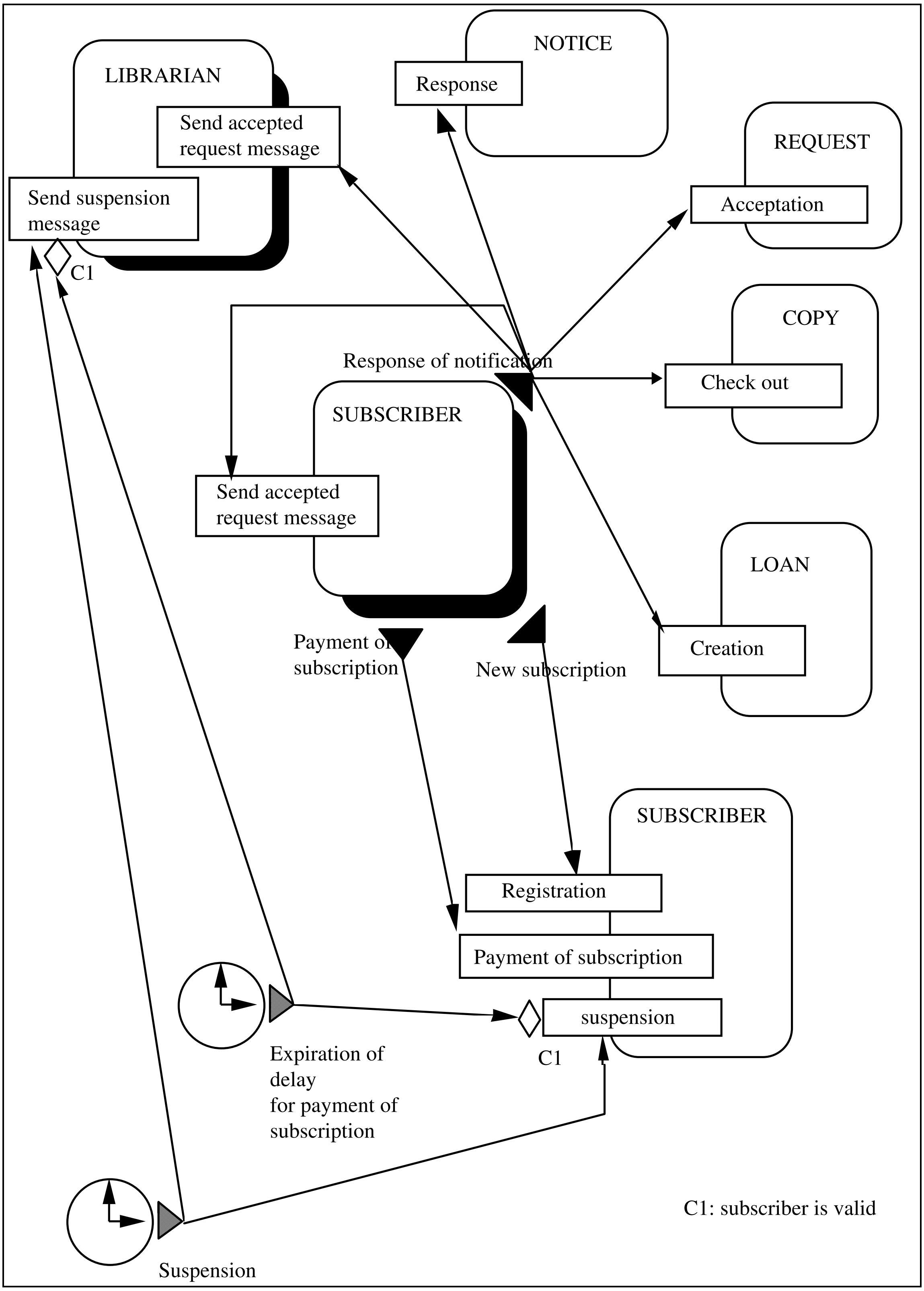


Figure 80 : Global behavior view (part 4)

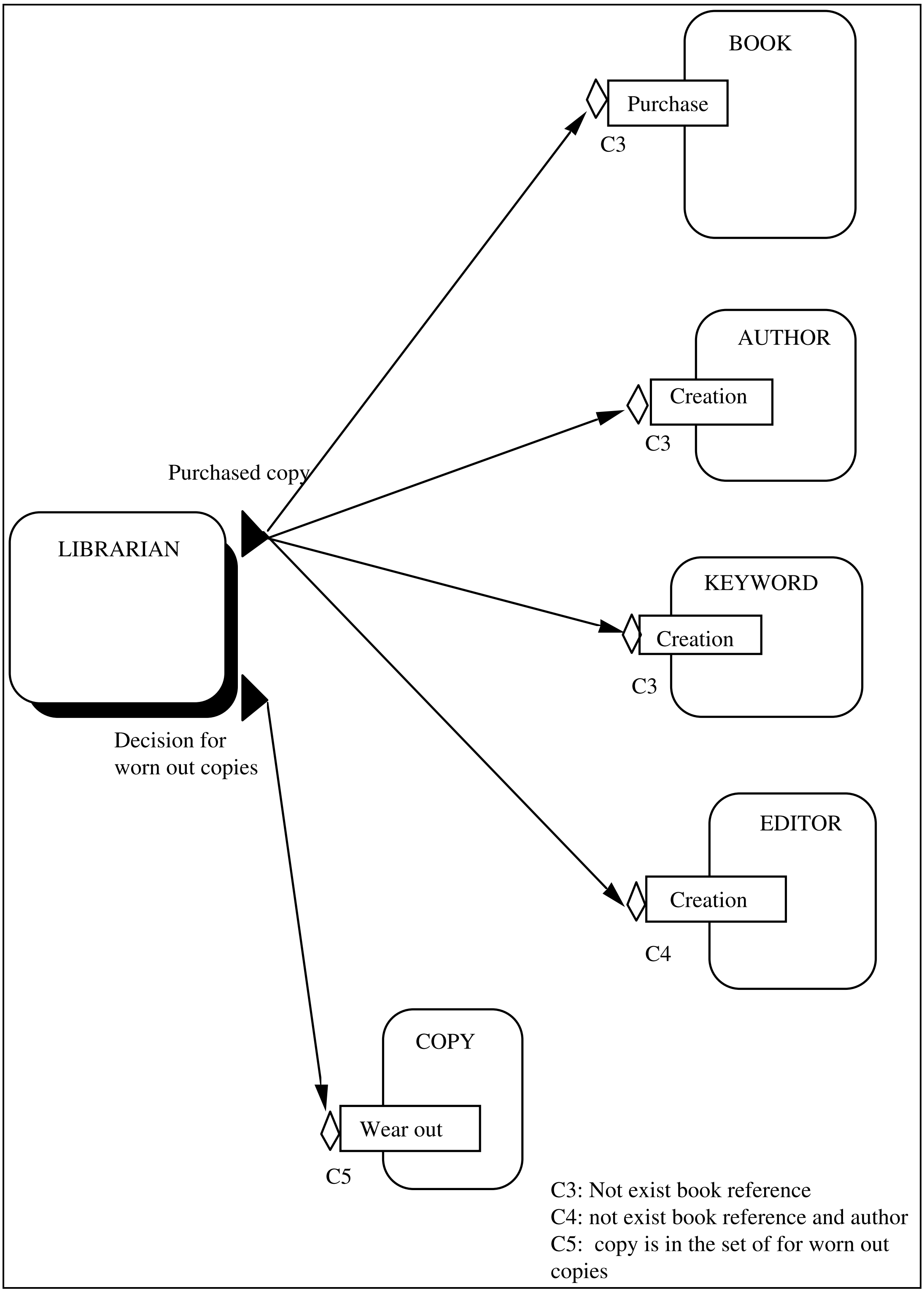


Figure 81 : Global behavior view (part 5)

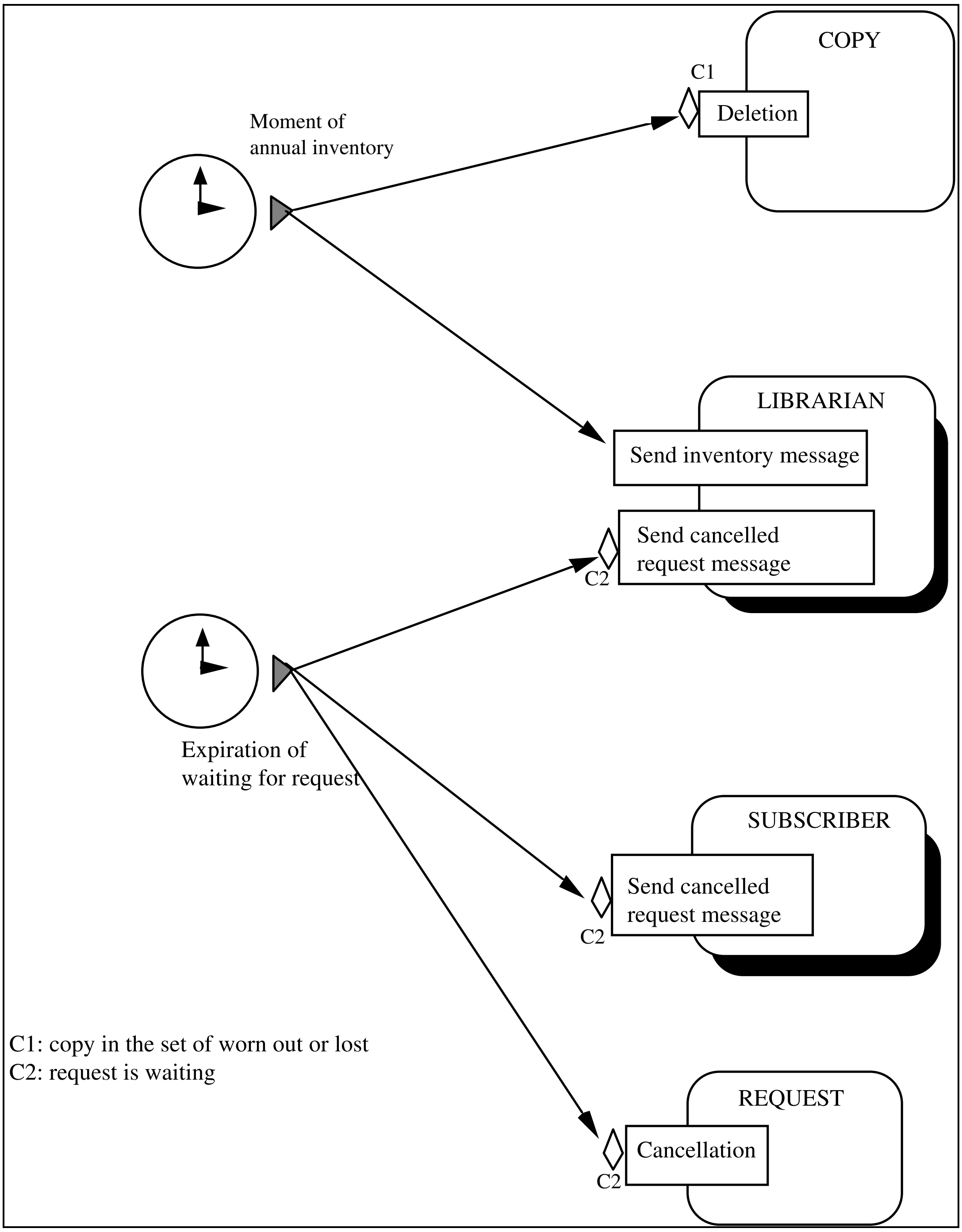


Figure 82 : Global behavior view (part 6)

3.7.5 Event Induction graph

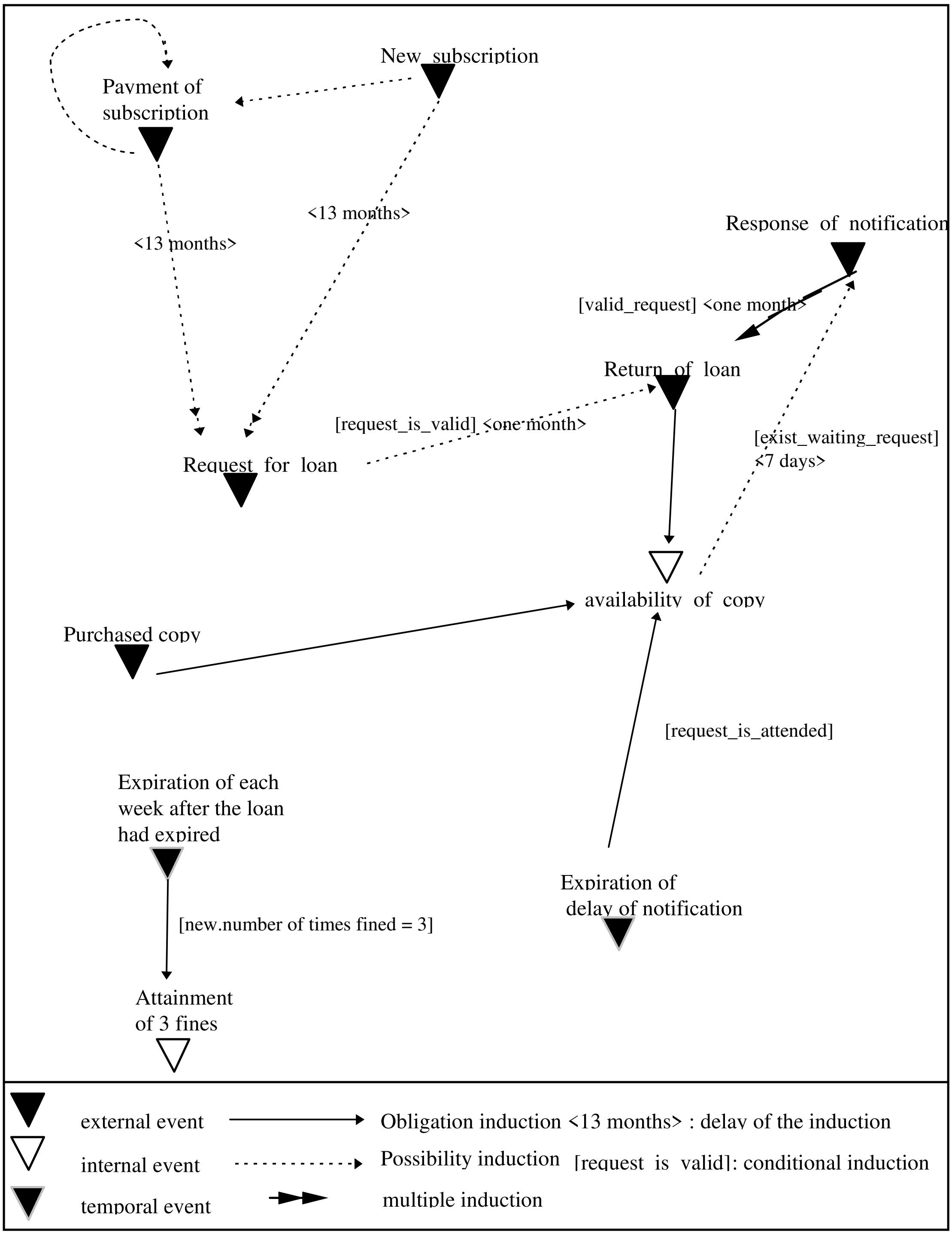


Figure 83: Event Induction graph

3.7.6 Event Concurrence graph

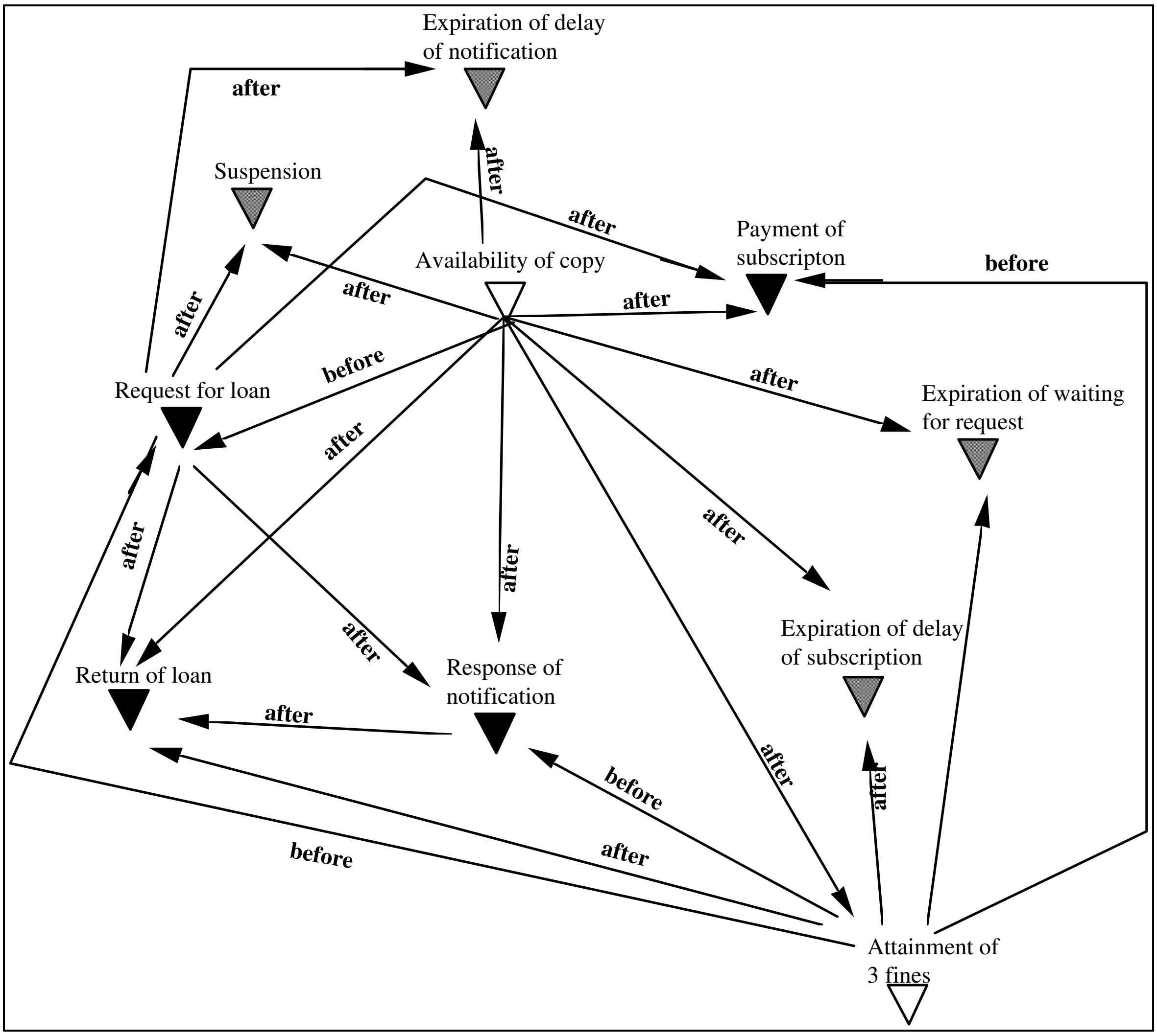


Figure 84 : Event concurrence graph (part 1)

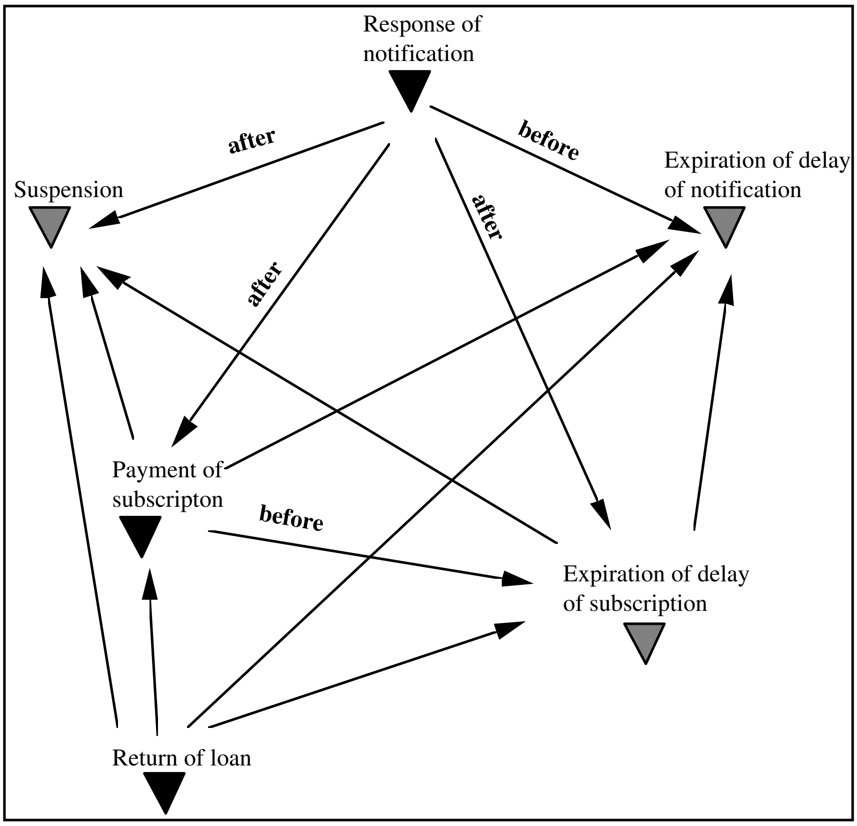


Figure 85 : Event concurrence graph (part 2)

3.7.7 Local class view

PERSON

properties
id of person : string
family name : string
first name : string
address: string
date of birth : date
profession : string

constraints
unique(Persons,family name, first name, date of birth)

operations
PERSON creation(p:structure(PERSON)) type create

Figure 86: Local class view of the object-class PERSON

EDITOR : PERSON

properties
books : inverse (zeromany, book.editor)

operations
EDITOR creation(e:structure(EDITOR)) type create :
Person::creation (e)

Figure 87: Local class view of the object-class EDITOR

AUTHOR : PERSON

properties
books : inverse (zeromany, book.author)

operations
AUTHOR creation(a:structure(AUTHOR)) type create :
Person::creation (e)

Figure 88: Local class view of the object-class AUTHOR

SUBSCRIBER : PERSON

properties

number of copies in-hand : integer
date of subscription : date
state : subscriber_state
requests : inverse (zeromany, REQUEST::subscriber)

operations

SUBSCRIBER registration(s:structure(SUBSCRIBER))
type create : Person::creation (e)
suspension() type update
exclusion() type update
payment of subscription() type update
add to copies in-hand() type update
reduce from copies in-hand() type update

queries

SUBSCRIBER find_subscriber(id:string) type class
service

Figure 89: Local class view of the object-class SUBSCRIBER

BOOK

properties

reference : string
title : string
edition date : date
last edition : string
price: integer
requests : inverse (zeromany, book.request)
copies : aggr (zeromany, COPY)
editor : asso (one, EDITOR, inverse of books)
authors : asso (many, AUTHOR, inverse of books)
keywords : asso (zeromany, KEYWORD, inverse of
books)

constraints

unique(Books.reference)

operations

BOOK purchase(p:PURCHASED COPY) type create :
agg(many, copy::purchase)
COPY add copy(p:PURCHASED COPY) type update

queries

BOOK $find book(p:PURCHASED COPY)

Figure 90: Local Class View of the object-class BOOK

COPY

properties
reference : string
date acquired : date
date put out for loan : date
state : copy_state
loans : inverse (zeromany, LOAN.copy)
notices : inverse (zeromany, NOTICE.copy)

constraints
unique(Copies,reference)

operations
COPY purchase(c:structure(COPY)) type create
reservation() type update
availability() type update
check out() type update
wear out() type update
loss() type update
deletion() type update

queries
boolean $exist_copy_available()
internal-events
availability of copy

Figure 91: Local Class View of the object-class COPY

KEYWORD

properties
content : string
creation date : string
books : inverse (zeromany, BOOK::keywords)

constraints
invariants
unique(Keywords,content)

operations
KEYWORD creation(k:structure(KEYWORD)) type create

Figure 92: Local Class View of the object-class KEYWORD

REQUEST

properties
date of request : date
date of cancellation : date
subscriber : asso (one, SUBSCRIBER, inverse of requests)
book : asso (one, BOOK, inverse of requests)
loan : inverse (zeroone, LOAN::request)
notice : inverse (zeroone, NOTICE::request)
state : request_state

operations
REQUEST acceptation(id of subscriber:string, reference of book:string) type create
REQUEST wait(id of subscriber:string, reference of book:string) type create
notification() type update
cancellation() type update

queries
REQUEST **$** find_earliest_request(book : BOOK)

Figure 93: Local Class View of the object-class REQUEST

LOAN

properties
date borrowed : date
return date : date
nunmber of times fined : integer
copy : asso (one, COPY, inverse of loans)
request : asso (one, REQUEST, inverse of loan)
reminders : inverse (zeromany, REMINDER::loan)
state : loan_state

operations
LOAN creation(r:REQUEST FOR LOAN) type create
expiration() type update
settlement() type update
update the number of times fined() type update

queries
LOAN **$** find_loan(m:RETURN_LOAN)

internal-events
attainment of 3 fines

Figure 94: Local Class View of the object-class LOAN

```
NOTICE

properties
  reference : string
  date of notification : date
  date of response : date
  request : asso (one, REQUEST, inverse of notice)
  copy : asso (one, COPY, inverse of notices)
  state : notice_state

constraints
  unique(Notices,reference)

operations
  NOTICE creation(c:COPY) type create
  response() type update
  cancellation() type update

queries
  NOTICE    $ find_notice(m:RESPONSE OF NOTIFICATION)
```

Figure 95: Local Class View of the object-class NOTICE

```
REMINDER

properties
  reference : string
  date of reminder : date
  date of settlement : date
  loan : asso (one, LOAN, inverse of
        reminders)
  state : reminder_state

constraints
  unique(Reminders,reference)

operations
  REMINDER creation(l:LOAN) type create
  settlement() type update
```

Figure 96: Local Class View of the object-class REMINDER

SUBSCRIBER (actor)

properties
id of subscriber : string

external operations
send_notice (n: NOTICE)
send_reminder (r: REMINDER)
send_waiting_request_message (r : REQUEST)
send_accepted_request_message (r : REQUEST, c: COPY)
send_cancelled_request_message (r: REQUEST)
send_exclusion_message (s: SUBSCRIBER)

external events
request for loan
response of notification
return of loan
new subscription
payment of subscription

Figure 97: Local Class View of the actor-class SUBSCRIBER

LIBRARIAN (actor)

properties
name of librarian : string

external operations
send_notice (n: NOTICE)
send_reminder (r: REMINDER)
send_waiting_request_message (r : REQUEST)
send_accepted_request_message (l:LOAN)
send_cancelled_request_message (r: REQUEST)
send_exclusion_message (s: SUBSCRIBER)
send_suspension_message (s: SUBSCRIBER)
send_lost_copy_message (c:COPY)
send_inventory_message (s : set (COPY))
send_no_copy_message (b:BOOK)

external events
purchased_copy
decision_for_worn_out_copies

Figure 98: Local Class View of the actor-class LIBRARIAN

CLOCK
temporal-events expiration of delay for payment of subscription expiration of waiting for request expiration of delay of notification moment of annual inventory expiration of delay for returning a book expiration of each week after the loan has expired suspension

Figure 99: Local Class View of the calendar-class CLOCK

3.7.8 Textual specification of the library system

application library system

definition

object-class PERSON **extents** Persons

properties

id of person : **string**

family name : **string**

first name : **string**

address: **string**

date of birth : **date**

profession : **string**

constraints

unique(Persons, family name, first name, date of birth**)**

operations

PERSON creation(p:structure(PERSON)) **type create**

endobject-class

object-class EDITOR **:** PERSON **extents** Editors

properties

books **: inverse (zeromany,** book.editor**)**

operations

EDITOR creation(e:structure(EDITOR)) **type create** : Person::creation (e)

endobject-class

object-class AUTHOR **:** PERSON **extents** Authors

properties

books **: inverse (zeromany,** book.author**)**

operations

AUTHOR creation(a:structure(AUTHOR)) **type create** : Person::creation (e)

endobject-class

object-class SUBSCRIBER **:** PERSON **extents** Subscribers

properties

number of copies in-hand : **integer**

date of subscription : **date**

state : **subscriber_state**

requests : **inverse (zeromany,** REQUEST::subscriber**)**

operations

SUBSCRIBER registration(s:structure(SUBSCRIBER)) **type create** : Person::creation (e)

suspension() **type update**

exclusion() **type update**

payment of subscription() **type update**

add to copies in-hand() **type update**

reduce from copies in-hand() **type update**

queries

SUBSCRIBER $ find_subscriber(id:string)

endobject-class

object-class BOOK **extents** Books

properties

reference : **string**

title : **string**

edition date : **date**

last edition : **string**

price: **integer**

requests **: inverse (zeromany,** book.request**)**

copies : **aggr(zeromany,** COPY**)**

editor : **asso (one,** EDITOR, **inverse of** books**)**

authors : **asso (many,** AUTHOR, **inverse of** books**)**

keywords : **asso (zeromany,** KEYWORD, **inverse of** books**)**

constraints

unique(Books,reference**)**

operations

BOOK purchase(p:PURCHASED_COPY) **type create : aggr(many,** copy::purchase**)**

/* copies will be created implicitly through this operation and the states of all copies will be iimplicitly initialized to "available" */

COPY add_copy(p:PURCHASED_COPY) **type update**

queries

BOOK $ find_book(p:PURCHASED_COPY)

endobject-class

object-class COPY **extents** Copies

properties

reference : **string**

date acquired : **date**

date put out for loan : **date**

state : **copy_state**

loans **: inverse (zeromany,** LOAN: :copy**)**

notices **: inverse (zeromany,** NOTICE : :copy**)**

constraints

unique(Copies,reference)

operations

COPY purchase(c:structure(COPY)) **type create**

reservation() **type update**

availability() **type update**

check out() **type update**

wear out() **type update**

loss() **type update**

deletion() **type update**

queries

boolean $ exist_copy_available()

internal-events

availability of copy

induction

may IF exist_waiting_request(**context**) **with delay** 7 **days**

context.request.subscriber.response of notification (**context**.notice,
context.request.subscriber,
context.request.book)

concurrent events

before request for loan // conflict access to subscriber and copy

after return of loan // conflict access to subscriber

after response of notification // conflict access to subscriber

after payment of subscription // conflict access to subscriber

after expiration of delay of notification // conflict access to subscriber

after expiration of delay for pyment of subscription // conflict access to subscriber

after suspension // conflict access to subscriber

after attainment of 3 fines // conflict access to subscriber

after expiration of waiting for request // conflict access to request

predicate(not(old.state = "available"**) and (new.state** = "available"**))**

trigger

if exist_waiting_request(**context**) **then context**.reservation()

if exist_waiting_request(**context**) **then**

Requests.find_earliest_request(**context**.**inverse of** BOOK::copies).notification()

if exist_waiting_request(**context**) **then** Notices.creation()

if exist_waiting_request(**context**) **then**

Requests.find_earliest_request(**context**.**inverse of** BOOK::copies).subscriber.add to copies in-hand()

if exist_waiting_request(**context**) **then**

Requests.find_earliest_request(**context**.**inverse of** BOOK::copies).subscriber. send_notice(**context.**notice)

if exist_waiting_request(**context**) **then**

Librarian.send_notice(**context.**notice)

endtrigger

endobject-class

object-class KEYWORD **extents** Keywords

properties

content : **string**

creation date : **string**

books **: inverse (zeromany,** BOOK::keywords**)**

constraints

invariants

unique(Keywords,content**)**

operations

KEYWORD creation(k:structure(KEYWORD)) **type create**

endobject-class

object-class REQUEST **extents** Requests

properties

date of request : **date**

date of cancellation : **date**

subscriber : **asso (one,** SUBSCRIBER, **inverse of** requests**)**

book : **asso (one,** BOOK, **inverse of** requests**)**

loan : **inverse (zeroone,** LOAN::request**)**

notice : **inverse (zeroone,** NOTICE::request**)**

state : **request_state**

operations

REQUEST acceptation(id of subscriber:string, reference of book:string) **type create**

REQUEST wait(id of subscriber:string, reference of book:string) **type create**

notification() **type update**

cancellation() **type update**

queries

REQUEST $ find_earliest_request(book : BOOK)

endobject-class

object-class LOAN **extents** Loans

properties

date borrowed : **date**

return date : **date**

number of times fined : **integer**

copy : **asso (one,** COPY, **inverse of** loans**)**

request : **asso (one,** REQUEST, **inverse of** loan**)**

reminders **: inverse (zeromany,** REMINDER::loan**)**

state : **loan_state**

operations

LOAN creation(r:REQUEST FOR LOAN) **type create**

expiration() **type update**

settlement() **type update**

update the number of times fined() **type update**

queries

LOAN $ find_loan(m:RETURN_LOAN)

internal-events

attainment of 3 fines

concurrent events

before availability of copy // conflict access to subscriber

before request for loan // conflict access to subscriber

after return of loan // conflict access to subscriber

before response of notification // conflict access to subscriber

before expiration of delay for payment of subscription // conflict access to subscriber

expiration of delay of notification // conflict access to subscriber

before payment of subscription // conflict access to subscriber

predicate(old.number of times fined < 3 **and new**.number of times fined = 3**)**

trigger

if subscriber_not_excluded(**context**.request.subscriber) **then**

context.request.subscriber.exclusion()

if subscriber_not_excluded(**context**.request.subscriber) **then**

context.copy.loss()

if subscriber_not_excluded(**context**.request.subscriber) **then**

context.request.subscriber.send_exclusion_message()

if subscriber_not_excluded(**context**.request.subscriber) **then**

Librarian.send_exclusion_message(**context**.request.subscriber)

if subscriber_not_excluded(**context**.request.subscriber) **then**

Librarian.send_lost_copy_message(**context**.copy)

endtrigger

endobject-class

object-class NOTICE **extents** Notices

properties

reference : **string**

date of notification : **date**

date of response : **date**

request : **asso (one,** REQUEST, **inverse of** notice**)**

copy : **asso (one,** COPY, **inverse of** notices**)**

state : **notice_state**

constraints

unique(Notices,reference**)**

operations

NOTICE creation(c:COPY) **type create**

response() **type update**

cancellation() **type update**

queries

NOTICE $ find_notice(m:RESPONSE OF NOTIFICATION)

endobject-class

object-class REMINDER **extents** Reminders

properties

reference : **string**

date of reminder : **date**

date of settlement : **date**

loan : **asso (one,** LOAN, **inverse of** reminders**)**

state : **reminder_state**

constraints

unique(Reminders,reference**)**

operations

REMINDER creation(l:LOAN) **type create**

settlement() **type update**

endobject-class

actor-class SUBSCRIBER

properties

id of subscriber : **string**

external operations

send_notice (n: NOTICE)

send_reminder (r: REMINDER)

send_waiting_request_message (r : REQUEST)

send_accepted_request_message (r : REQUEST, c: COPY)

send_cancelled_request_message (r: REQUEST)

send_exclusion_message (s: SUBSCRIBER)

external events

request for loan

message REQUEST_FOR_LOAN

reference of book : **string**

id of subscriber : **string**

induction

may IF valid_request (**context**) **with delay** 1 month

self.return of loan(copy_reference=**context**.copy)

concurrent events

after availability of copy // conflict access to subscriber and copy

after attainment of 3 fines // conflict access to subscriber

after suspension // conflict access to subscriber

after return of loan // conflict access to subscriber and copy

after payment of subscription // conflict access to subscriber

after expiration of delay of notification //conflict access to subscriber and copy

after response of notification // conflict access to subscriber

after expiration of delay for payment of subscription //conflict access to subscriber

predicate(the book must exist and the subscriber must exist in valid state)

trigger

if valid_request(**context**) **then**

Requests.acceptation(Subscribers.find_subscriber(**context.**id of subscriber),

context.reference of book)

if valid_request(**context**) **then** Loans.creation(**context**)

if valid_request(**context**) **then** Copies.find_copy_available(**context**).check out()

if valid_request(**context**) **then**

Subscribers.find_subscriber(**context.**id of subscriber).add to copies in-hand()

if copy_not_available(**context**) **then**

Requests.wait(Subscribers.find_subscriber(**context.**id of subscriber),

context.reference of book)

if copy_not_available(**context**) **then**

context.id of subcriber.send_waiting_request_message (**context.**request)

if copy_not_available(**context**) **then**

Librarian.send_waiting_request_message (**context.**request)

if valid_request(**context**) **then**

Librarian.send_accepted_request_message (**context.**loan)

if valid_request(**context**) **then**

context.id of subcriber.send_accepted_request_message (**context.**request**,**

context.copy)

endtrigger

response of notification

message RESPONSE_OF_NOTIFICATION

reference of notification : **string**

id of subscriber : **string**

reference of book : **string**

induction

may IF valid_request (**context**) **with delay** 1 month

self.return of loan(copy_reference=**context**.copy)

concurrent events

before availability of copy // conflict access to subscriber

after attainment of 3 fines // conflict access to subscriber

after suspension // conflict access to subscriber

after return of loan // conflict access to subscriber

after payment of subscription // conflict access to subscriber

before expiration of delay of notification //conflict access to subscriber

after expiration of delay for payment of subscription //conflict access to subscriber

before request for loan // conflict access to subscriber

predicate (the notification must exist and today < date of notification +7 days)

trigger

Notices.find_notice(**context**).response()

Notices.find_notice(**context**).request.acceptation(**context**.id of subscriber, **context**.reference of book)

Notices.find_notice(**context**).copy.check out()

Loans.creation(**today**,

Notices.find_notice(**context**).request,

Notices.find_notice(**context**).copy)

Librarian.send_accepted_request_message (**context.**loan)

context.id of subcriber.send_accepted_request_message (**context.**request**,**

context.copy)

endtrigger

Return of loan

message RETURN_LOAN

s_date_of_birth : **date**

s_family_name : **string**

s_name : **string**

copy_reference : **string**

induction

must context.copy.availability of copy()

concurrent events

before availability of copy // conflict access to subscriber

before attainment of 3 fines // conflict access to subscriber

suspension // conflict access to subscriber

before response of notification // conflict access to subscriber

payment of subscription // conflict access to subscriber

expiration of delay of notification // conflict access to subscriber

expiration of delay for payment of subscription // conflict access to subscriber

before request for loan // conflict access to subscriber

predicate(the subscriber must exist and the copy must exist)

trigger

if (return_loan_is_late(Loans.find_loan(**context**)) **then**

foreach r in set_of_reminders(Loans.find_loan(**context**)) **do** r.settlement()

Loans.find_loan(**context**).copy.availability()

Loans.find_loan(**context**).settlement()

Loans.find_loan(**context**).request.subscriber.reduce from copies in-hand()

endtrigger

new subscription

message NEW_SUBSCRIPTION

sub : **structure**(SUBSCRIBER)

induction

may IF subscriber_not_exist (**context**.sub) **with delay** 1 year **more**

self.request for loan()

may IF subscriber_not_exist (**context**.sub) **with delay** 1 year

self.payment of subscription()

predicate(**today** - sub.date of birth >= 8)

trigger

if subscriber_not_exist(**context.**sub) **then**

Subscribers.registration(**context.**sub)

endtrigger

payment of subscription

message PAYMENT_OF_SUBSCRIPTION

id of subscriber : **string**

induction

may IF subscriber_not_excluded (**context**.sub) **with delay** 1 year **more**

self.request for loan()

may IF subscriber_not_excluded (**context**.sub) **with delay** 1 year

self.payment of subscription()

concurrent events

before availability of copy // conflict access to subscriber

after attainment of 3 fines // conflict access to subscriber

suspension // conflict access to subscriber

before response of notification // conflict access to subscriber

expiration of delay of notification // conflict access to subscriber

before expiration of delay for payment of subscription //conflict access to subscriber

before request for loan // conflict access to subscriber

return of loan // conflict access to subscriber

predicate(the subscriber must exist)

trigger

```
        if subscriber_not_excluded(Subscribers.find_subscriber(context.id of subscriber)) then
            Subscribers.find_subscriber(context.id of subscriber).payment of subscription()
    endtrigger
endactor-class

  actor-class LIBRARIAN
  properties
    name of librarian : string

    external operations
      send_notice (n: NOTICE)
      send_reminder (r: REMINDER)
      send_waiting_request_message (r : REQUEST)
      send_accepted_request_message (l:LOAN)
      send_cancelled_request_message (r: REQUEST)
      send_exclusion_message (s: SUBSCRIBER)
      send_suspension_message (s: SUBSCRIBER)
      send_lost_copy_message (c:COPY)
      send_inventory_message ( s : set (COPY))

  external events

    purchased_copy
    message PURCHASED_COPY
        b : structure(BOOK)
        date put out for loan : date
    induction
        must context.copy.availability of copy()

    trigger
        if not_exist_book_reference(context) then Books.purchase(context)
        if not_exist_book_reference(context) then
            foreach a in set_of_authors do Authors.creation(a)
        if not_exist_book_reference(context) then
            foreach k in set_of_keywords do Keywords.creation(k)
        if not_exist_book_reference_and_editor(context) then Editors.creation(context)
        add_copy(context)
    endtrigger
```

```
decision_for_worn_out_copies
    message D_WORN_OUT_COPIES
        copy_refs : set(string)
    predicate ( the copies must exist)
    trigger
        foreach c in set_of_for_worn_out_copies(context) do c.wear out()
    endtrigger
endactor-class

calendar-class CLOCK
    temporal-events

    expiration of delay for payment of subscription
    concurrent events
            before availability of copy // conflict access to subscriber
            after attainment of 3 fines // conflict access to subscriber
            suspension // conflict access to subscriber
            before response of notification // conflict access to subscriber
            expiration of delay of notification // conflict access to subscriber and copy
            after payment of subscription // conflict access to subscriber
            before request for loan // conflict access to subscriber
            return of loan // conflict access to subscriber

        predicate( today = SUBSCRIBER.date of subscription + 13 months)
        trigger /* context is a subscriber */
            if subscriber_is_valid(context) then context.suspension()
            if subscriber_is_valid(context) then Librarian.send_suspension_message(context)
        endtrigger

    expiration of waiting for request
        concurrent events
            before availability of copy // conflict access to request

        predicate(today = REQUEST::date of request+3 months)
        trigger /* context is a request */
                if request_is_waiting(context) then context.cancellation()
                if request_is_waiting(context) then
                    context.subscriber.send_cancelled_request_message(context)
                if request_is_waiting(context) then
```

Librarian.send_cancelled_request_message(**context**)

endtrigger

expiration of delay of notification

induction

must IF request_is_attended(**context**) **context**.copy.availability of copy()

concurrent events

before availability of copy // conflict access to subscriber

attainment of 3 fines // conflict access to subscriber

suspension // conflict access to subscriber

after response of notification // conflict access to subscriber

expiration of delay for payment of subcription // conflict access to subscriber

payment of subscription // conflict access to subscriber

before request for loan // conflict access to subscriber

return of loan // conflict access to subscriber

predicate(today = NOTICE::date of notification + 7 **days)**

trigger /* context is a notice */

if request_is_attended(**context**) **then context**.cancellation()

if request_is_attended(**context**) **then context**.request.cancellation()

if request_is_attended(**context**) **then context**.copy.availability()

if request_is_attended(**context**) **then**

context.request.subscriber.reduce from copies in-hand()

if request_is_attended(**context**) **then**

context.request**.**subscriber.send_cancelled_request_message(**context.**request)

if request_is_attended(**context**) **then**

Librarian.send_cancelled_request_message(**context.**request)

endtrigger

moment of annual inventory

predicate(each end_of_year)

trigger

foreach c **in** set_of_worn_out_or_lost() **do** c.deletion()

Librarian.send_inventory_message (set_of_worn_out_or_lost())

endtrigger

expiration of delay for returning a book

predicate(today =LOAN::date borrowed+1 **month)**

trigger /* context is loan */

if loan_is_valid(**context**) **then context**.expiration()

endtrigger

expiration of each week after the loan has expired

predicate(today = (LOAN::date borrowed +1month)+n+1 **weeks)**

trigger /* context is loan */

if loan_is_expired(**context**) **then context**.update the number of times fined()

if loan_is_expired(**context**) **then** Reminders.creation(**context**)

if loan_is_expired(**context**) **then** context.request.subscriber.send_reminder(**context**)

if loan_is_expired(**context**) **then** Librarian.send_reminder(**context**)

endtrigger

suspension

concurrent events

before availability of copy // conflict access to subscriber

before attainment of 3 fines // conflict access to subscriber

expiration of delay of notification // conflict access to subscriber

before response of notification // conflict access to subscriber

expiration of delay for payment of subcription // conflict access to subscriber

payment of subscription // conflict access to subscriber

before request for loan // conflict access to subscriber

return of loan // conflict access to subscriber

predicate(today=LOAN::date borrowed+1 **month**+2 **weeks))**

trigger

context.request.subscriber.suspension()

Librarian.send_suspension_message(**context**.request.subscriber)

endtrigger

endclass-calendar

domain subscriber_state : { "valid", "suspended", "excluded" }

domain copy_state : { "available", "reserved", "check out", "worn out", "lost" } **enddomain**

domain request_state : { "accepted", "wait", "attended", "cancelled" } **enddomain**

domain loan_state : { "valid", "expired", "settled"} **enddomain**

domain notice_state : { "valid", "cancelled", "responded"} **enddomain**

domain reminder_state : { "fined", "paid"} **enddomain**

triggering-condition

boolean exist_waiting_request(c:COPY)

comment does there exist a request waiting for the book became available which can be satisfied ?

endtriggering-condition

triggering_condition

boolean valid_request(r: REQUEST FOR LOAN)

comment if a copy of the request book is available and the subscriber is valid and the number of copies in hand < 3

endtriggering-condition

triggering_condition

boolean copy_not_available(r: REQUEST FOR LOAN)

comment there is no available copy of the requested book

endtriggering-condition

triggering_condition

boolean not_exist_book_reference(p:PURCHASED_COPY)

comment the book does not exist

endtriggering-condition

triggering_condition

boolean not_exist_book_reference_and_editor(p:PURCHASED_COPY)

comment the editor does not exist

endtriggering-condition

triggering_condition

boolean subscriber_not_exist(n:NEW SUBCRIPTION)

comment the subscriber does not exist

endtriggering-condition

triggering_condition

boolean subscriber_not_excluded(s:SUBSCRIBER)

comment the subscriber are not excluded

endtriggering-condition

triggering_condition

boolean subscriber_is_valid(s:SUBSCRIBER)

comment the subscriber is valid

endtriggering-condition

triggering_condition

boolean request_is_waiting(r:REQUEST)

comment the request is in the waiting state

endtriggering-condition

triggering_condition

boolean request_is_attended(n:NOTICE)

comment the state of the request is "attended"

endtriggering-condition

triggering_condition

boolean loan_is_valid(l:LOAN)

comment the state of the loan is valid

endtriggering-condition

triggering_condition

boolean return_loan_is_late (l:LOAN)

comment today > date borrowed of the loan + 1 month

endtriggering-condition

triggering-factor

set(AUTHOR) set_of_authors(p:PURCHASED_COPY)

comment set of authors of the purchased book

endtriggering-factor

triggering-factor

set(KEYWORD) set_of_keywords(p:PURCHASED_COPY)

comment set of keywords associated to the purchased book

endtriggering-factor

triggering-factor

set(COPY) set_of_for_worn_out_copies(d : D_WORN_OUT_COPIES)

comment set of copies mentioned in the worn out copies message

endtriggering-factor

triggering-factor

set(COPY) set_of_worn_out_or_lost()

comment set of worn_out or lost copies

endtriggering-factor

triggering-factor

set(REMINDER) set_of_reminders(l:LOAN)

comment set of reminders of the loan

endtriggering-factor

4. THE TEMPORAL EXTENSION (TOOM)

Contents

4.1 Introduction

The Temporal Object Oriented Method (TOOM) is described in two fragments of methods. The first fragment deals with the basic OO method whereas the second fragment describes the temporal extension of the OO method. This chapter is the reference manual of the temporal extension. We recommend to read the reference manual of the basic OO method.

The pre-requisite for reading this document is to understand the concepts of object class, actor class, calendar class, event and operation explained in the reference manual of the basic OO method (Chapter 3), first.

The document reminds to the reader the basic knowledge relative to temporal management in temporal database applications (see Appendix 4.4.1page 244).

The document is structured in three sections and three appendices :

- introduction
- the temporal requirements handled by the methodology,
- the concepts of the temporal extension,
- Appendices :
    - the foundation of time management,
    - the syntax of the specification language,
    - Case study.

4.2 Temporal requirements

Time is everywhere in information systems. Even if time is required for every application, the use and the need relative to time are different. We have identified five categories of use :

Temporal database

This category corresponds to database applications including data and business rules strongly dependent on time. Time is linear and discrete.

Temporal reasoning

This category aims at reasoning on events and/or fuzzy events for finding their possible temporal ordering. Temporal reasoning is based on intervals (time is not anchored to the time line). A particular use of temporal reasoning is automatic planning.

Branching time

Classically, the time is considered linear, but some applications manage activities done in parallel with different time and/or several calendars (versioning in design database, events in parallel systems etc.).

<u>Real-time systems</u>

Real-time systems are interested in the duration, the priority and the synchronization of tasks.

<u>Time series management</u>

A time series are defined by a set of vectors for different time period. A vector is a set of values for a time period . In general, the time series are based on periodicity requiring a user-defined granularities or a user-defined calendar. This management requires special functionality on the data management and time management.

The conventional databases are designed to store and retrieve current data. During update operations, the data relative to evolving real situations are erased and replaced by their new values. It means that past data are no longer in the database, only current data are kept. Except if the designer of the application their historisation.

The relevance of time in the database area has been revealed from 1975. From this date, applications require more and more to manage past, present and future data. This justifies to add to DBMS, mechanisms dedicated to model, retrieve, store and modify time dependent data. This trend is confirmed by the introduction of SQL/T (temporal extension of SQL) in SQL3. This new generation of temporal DBMS integrates time into the data model in order to facilitate the management of temporal aspects at the application layer. Such DBMS have simplified the storage and the request of temporal data. But, they don't solve design problems such as : definition of the data schema, specification of the activities of the application, verification of the consistency of the temporal database and also how to administrate such database. The temporal extension of the OO method is focusing on these design problems relative to *temporal databases*.

The temporal requirements relative to the temporal database is refined in 7 sub-categories :

- *User-defined time management* means that the semantic of time is managed by the application. This management requires to have an extended set of temporal abstract data types.

- *Management of reports of activities on periods of time* requires to handle the valid time on information related to the relevant activities.

- *Retroaction and postaction* represent respectively activities performed with a certain delay or in advance. In these two cases, the consistency of the database through the time is assured by undo treatments and reply actions using the current state of the database.

- *Business rules dependent to time* requires functionality such as :
  - handling temporal triggers or alerters,
  - handling temporal constraints on event occurrences,

- handling statistical triggers or alerters.

- *Planning oriented applications* essentially require the use of relative time instead of absolute time.

- *Auditing-oriented applications* need to use the transaction time. It's the time during which the data was/is/will be present in the database. It permits to keep track of input mistakes, delay of registration in order to measure their impact on the activities of the enterprise.

- *Schema Versioning*. The TSQL2 terminology makes a distinction between schema modification, schema evolution and schema versioning. Schema modification does not guarantee to recover data whereas schema evolution has to guarantee it. And finally, in the schema versioning, the schema is considered as temporal data. It means that the evolution of the schema is kept in the database. Schema modification and schema evolution are not specific to temporal DBMS whereas schema versioning is. Schema versioning is not yet supported in current temporal DBMS.

The TOOM method supports the designer while analysis temporal applications meeting one or several of the six mentioned categories of temporal requirements.

4.3 Temporal extension

The temporal extension aims at introducing at the analysis level, the concepts required to manage the temporal aspects of a database application.

In addition information systems become more and more dataware houses which have to manage basic information of the enterprise but also derived information such as statistics, decisional supports , management reports etc. This new trend is taken into account at the analysis level, by providing in the OO method the notion of derived class.

The temporal extension of the OO method described in chapter 3 is structured in 8 points :

- definition of one or more calendars for an application,
- extension of basic domains to temporal domains,
- derived class : specialization of object class,
- extension of object classes to time dimensions,
- extension of static links between object classes to time dimensions,
- extension of constraints applied on object classes,
- extension of events to time dimensions,
- Database administration policy.

4.3.1 Definition of one or more calendars

A calendar is a metric system to apply on the time line (cf. section 4.4.1.2, page 248). The OO method proposes the concept of *calendar class* for defining temporal events of the application (cf. section 3.3.1.4.). In the basic OO method, the calendar is the Gregorian one. At the object level, the calendar class has only one instance : the clock of the *Gregorian* calendar.

We extend the definition of the *calendar class* for defining any calendar useful for the application.

A calendar is described by :

- a *name*,
- an *origin* (instant),

- a *basic unit (granule)*,
- a *set of granules* . Each granule is described by the upward and downward mappings. If the mapping is regular, the mapping is described with a coefficient whereas an irregular mapping is explained by a function name.
- three *operations* have to be redefined for the calendar :
    - addition of an interval to an instant,
    - addition of an instant to an interval,
    - extraction of an interval between two instants.
- a *set of external formats*. An external format is a temporal format used by the application which has to be mapped into the defined calendar. They are managed by the application and the mapping functions are explicitly invoked by the application.

The *Gregorian* calendar is described in 4.3.1[1]. The chronon of the system is a second. As we can see , months and minute are granules with irregular mapping (cf. section 4.4.1.2 page 248).

```
Calendar class    Gregorian
origin   "01/01/0001:00:00:00"  basic unit =chronon
granules
second :   {1 basic unit, basic unit=1second}
minute :  {minute_second(), second_minute()}
hour :  {60 minutes, minute=1/60hour}
day :  {24 hours, hour=1/24day}
month :  {month_day(),day_month()}
year :  {12 months, mois=1/12 year}
operations
instant<Gregorian, *> operator+ (instant<Gregorian, *>, interval<Gregorian, *>)
instant<Gregorian, *> operator+ (interval<Gregorian, *>, instant<Gregorian, *>)
interval<Gregorian, *> operator- (instant<Gregorian, *>, instant<Gregorian, *>)
external formats
TimeZone= "instant<Gregorian, hour>, TZ, DST"
              {instant<Gregorian, hour> input(TimeZone),
               TimeZone output(instant<Gregorian, hour>)}
EndCalendarClass
```

Figure 100 : The Gregorian Calendar Definition (defined by the system).

[1] The irregular mapping is denoted by a function : minute_second().
The three temporal types are instant (point on the time line), interval (unanchored quantity of time) and period (quantity of time between two instants).
Instant<Gregorian,hour> means an instant at the *hour* level of the *Gregorian* calendar.
*Instant <Gregorian, *>* means an instant with one of the granularity of the *Gregorian* calendar.
*Interval<Gregorian, *>* means an interval with one of the granularity of the *Gregorian* calendar.

An instant in the Gregorian calendar has the following format : "year-day-month hour: minute: second". This format is used to store physically an instant in the database. But, companies become more and more wide-world, it means that information systems have to deal with temporal data spread in different time zones. The Gregorian calendar includes local time as an external format. The external format *TimeZone* is composed of :

- the value of an instant,
- the value of the time zone (the property is called GMT and the value belongs to [-12, 12]),
- the value of the day saving time (the property is called DST and the value is true if the time is different in Winter and Summer otherwise it is false).

The input function allows us to map a local time to a "reference time" which is the time at GMT+0 without DST. The output function do the inverse mapping from a reference time to a local time.

The Gregorian calendar is provided by default as it is defined in 4.3.1.

How to proceed if the application requires extra granules such as week, quarter etc. ?

You have to analyze why the application needs such granules. Two cases are possible :

- the granularity is helpful for giving and retrieving information. No temporal manipulation is done with this granule. Thus, the granule can be viewed as user-defined external formats of the Gregorian calendar. For each granule, you have to define the input and the output functions. Figure 101 : Overloading of the Gregorian Calendar. shows how to add the external format *week* to the Gregorian Calendar.

Calendar class Gregorian

External formats

week="year/week" {instant<Gregorian, day> input(week),
 week output(instant<Gregorian, day>)}
endCalendarClass

Figure 101 : Overloading of the Gregorian Calendar.

- the granularity is used to measure and refer to instant on the time line. It means that time manipulation and physical storage require this granularity. Thus, a new calendar should be defined. Figure 102 explains how to define new calendars such as *MyWeek* and *MyHalfYear*.

```
Calendar class    MyWeek
origin ="01/0001" , basic unit = <Gregorian, day>
granules
week :  {day_week(), week_day() }
year :  {53 weeks, week=1/year},
operations :
instant<MyWeek,*> operator + (instant<MyWeek,*>, interval<MyWeek,*>)
instant<MyWeek,*> operator + ( interval<MyWeek,*>, instant<MyWeek,*>)
interval<MyWeek,*> operator - ( instant<MyWeek,*>, instant<MyWeek,*>)
End Class
```

```
Calendar class    MyHalfYear
origin ="01/0001" , basic unit=<Gregorian,month>
granules
half :  {6 months, month=1/6half}
year :  {2 halfs, half=1/2year},
operations :
instant<MyHalfYear,*> operator + (instant< MyHalfYear,*>,
interval< MyHalfYear,*>)
instant< MyHalfYear,*> operator + (interval < MyHalfYear,*>,
instant< MyHalfYear,*>)
interval < MyHalfYear,*> operator - ( instant< MyHalfYear,*>,
instant< MyHalfYear,*>)
End Class
```

Figure 102 : MyWeek and MyHalfYear calendars.

How to use several calendars in the same application ?

Assume that we have three object classes : *Employee*, *Project* and *Charge* (Figure 103). The class *Charge* represents the amount of time dedicated to a project by a person for a given week. The *birthday* of the employee is an instant at the *day* level of the *Gregorian* calendar whereas the *week* attribute in the *Charge* class is an instant at the *week* level of the *MyWeek* calendar. The *charge* attribute is a number of hours ; it means an interval at the *hour* level of the *Gregorian* calendar.

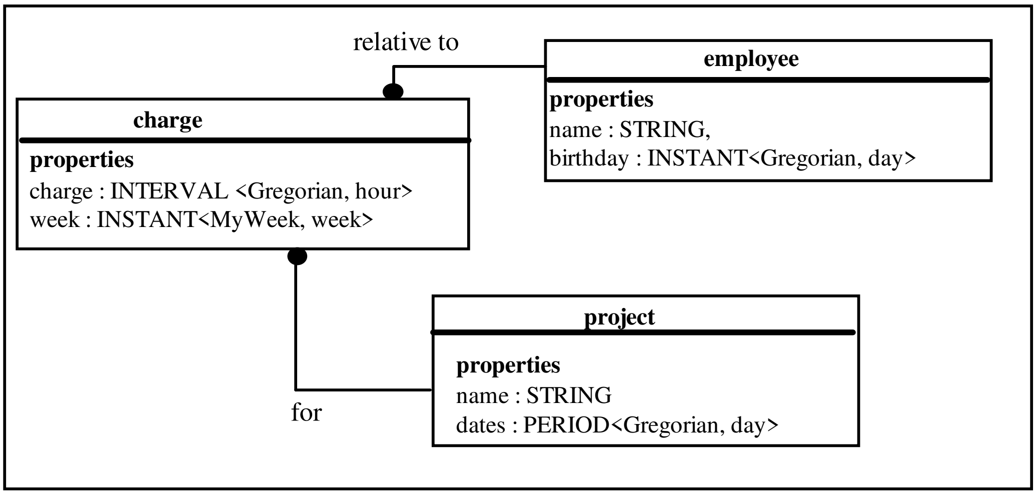


Figure 103 : examples of classes using several calendars

We will see in section 4.3.7.3 (page 202) how to integrate the temporal events of the application into the different calendars.

4.3.2 Extension of the basic domains to temporal domains

Three temporal domains are introduced in section 4.4.1.1 (page 244) : instant, period and interval. An instant represents a point on the time line (*1996-12-12*). A period represents the quantity of time between two instants (*[1996-12-12, 1996-12-15)*[2]). By contrast an interval is an unanchored quantity of time (*3months*).

These three types are expressed with one granularity of one calendar. Consequently, the granularity and the calendar are their parameters. For instance, an instant expressed at the *week* level of *MyWeek* calendar is described as INSTANT<MyWeek, week>. An interval expressed at the *hour* level of *Gregorian* calendar is defined as follows : INTERVAL<Gregorian, hour>. And finally, PERIOD<Gregorian, day> defines a period at a *day* level of *Gregorian* calendar. Figure 104 shows different temporal attributes.

Temporal attributes :
birthday : INSTANT<Gregorian,day>
charge : INTERVAL <Gregorian, hour>
week : INSTANT <MyWeek, week>
dates : PERIOD <Gregorian, day>

Figure 104 : examples of temporal attributes

These generic temporal domains *(INSTANT<calendar, granule>, PERIOD<calendar, granule> and INTERVAL<calendar, granule>)* replace the classical types : *date*, *time* and *datetime* found in the classical DBMS.

[2] [1996/12/12, 1996/12/15) expresses a period including the beginning of the period and excluding the end of the period.

The two domains *INSTANT<calendar, granule> and PERIOD<calendar, granule>* permit to define temporal attributes which are absolute or relative times (cf. section 4.4.1.3 page 213). These generic domains are specialized in two categories for absolute and relative times respectively.

The domains *INSTANT-A<calendar, granule>, PERIOD-A<calendar, granule>* are used to specify absolute time domains.

But applications require to manage and store relative time for planning reasons. For instance, the end of the project is equal to *"J0+30months"*; *J0* is a temporal mark. In addition, the *task1* of the project should be the relative period *["begin of the project+7days", "begin of the project +2months").* The relative time appears as a temporal derived attribute. The temporal expression of a relative time is limited to *<temporal mark>* + *<interval>*. Instead of storing the temporal value, the temporal expression used to derive its value is stored. The characteristics of a relative time is to have no value until its temporal mark is valued. The relative time becomes absolute time since the valuation of its temporal mark has occurred. The temporal mark of a relative time attribute is a temporal attribute which can be itself a relative time or an absolute time. In the previous examples, *J0 and begin of the project* must be temporal attributes of the application.

The relative time is introduced by two new temporal domains : *INSTANT-R<calendar, granule> and PERIOD-R<calendar, granule>*.

Figure 105 shows how to use relative time domains. The *begin* attribute in the *project* class is an absolute time expressed at a *day* level of the *Gregorian* calendar. *The end of the project* is defined as a relative time. The *date of the task* is defined as a relative period of time. For instance, *the end of the TOOBIS project is « begin of the project + 30 months* ». The temporal mark is the *begin* attribute of the same project which is an absolute time. This project is composed of two tasks *T1* and *T2*. The task *T1* is planned to be *during [« begin of the project TOOBIS+7days », « begin of the project TOOBIS + 2months »)* whereas the task *T2* should take place during *[« T1().date.end +7days », «T1().date.end + 2months »).* The temporal mark of the attribute *date* of *T2 is the end of the task T1 of the project TOOBIS* which is also a relative time. The function *T1()* is a OQL or TOQL query allowing to access to task T1 from the task T2.

Two predefined relative instants are defined *tomorrow* and *yesterday* (cf. section 4.4.1.3 page 253). The usage of relative time is restricted to the user-defined times. It means that the valid time cannot be a relative time, it is always an absolute time from the past, the present or the future.

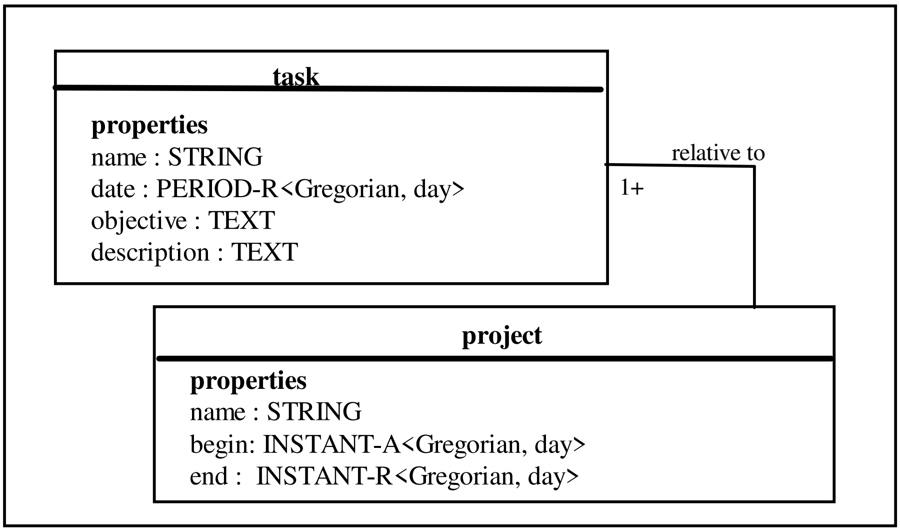


Figure 105 : examples of relative time attributes.

The generic temporal domains *INSTANT<calendar, granule>, PERIOD<calendar, granule>* are used when the attribute can be either absolute or relative time.

When temporal domains have to be used ?

These new temporal domains are used to define the domain value of temporal attributes. Three time dimensions are defined in the temporal DBMS : valid time, transaction time and user-defined time (cf. section 4.4.1.4, page 254). The valid and the transaction time are managed by the temporal DBMS (cf. section 4.4.1.4, page 254). The analyst defines explicitly in his specification the temporal attributes belonging to the user-defined times (semantically dependent on the application).

The transaction time is the time during which an information was/is present in the database ; its semantic is linked to the computerized information system instead of the application. For instance*, the time registration of an order* into the database is the transaction time of this order. In general, the problem is to have a clear distinction between the valid and the user-defined times.

What is the semantic associated to a temporal attribute ?

It is often difficult to differentiate the valid time and a user-defined attribute. The *birthday* of a *person* is dependent on the application. The attribute *week* of the *Charge* class (cf. Figure 103) is the valid time of the declared charge.

The *begin and the end* of the project (cf. Figure 105) should represent the valid time of the project if they were not declared as relative times. If based on relative time, *the begin and the end of the project* are user-defined temporal attributes. They represent the planning of the project, and therefore are not valid times.

The *dates* of a task (cf. Figure 105) should represent the planning of a task but not its valid time. Planning time or intended time is managed by the application.

4.3.3 Derived class Vs Basic class : specialization of object class

Information systems become more and more data warehouses or Executive Information Systems (EIS). Such systems may be projected in three distinct spaces [Toobis96a] : data, aggregation and variation spaces. The data space deals with basic information managed by the enterprise. The aggregation space results of aggregate computation on data elements, such as *the amount of sales per vendor and per week*. Finally, the variation space deals with the observation of changes of derived data belonging to the aggregation space.

Executive Information Systems (EIS) include basic information about the enterprise but also derived information such as statistics, decisional supports, management reports etc. The specification of such Information Systems requires to take into account the exploitation of basic information as well as derived information.

The persistent and relevant information of an application is designed in the OO method by the notion of *object class* (cf. section 3.3.1.2.).

The extension specializes this concept in two classes : *basic class and derived class*. A basic class is an object class describing basic objects of the enterprise (not derivable) or information belonging to the data space. A derived class is an object class derived from one or more object classes. It means that a derived object can be created or updated through its derivation function. A derived class is used to structure the aggregation space.

In the following figure, the *Order* class is a basic class whereas the *WeeklySale* class is a derived class representing the amount of sales for a vendor in one week. The objects of the class WeeklySale are created from the orders. In the global static diagram, derived and basic classes are graphically represented by a black box with the name of the class and its type : « basic class » or « derived class ».

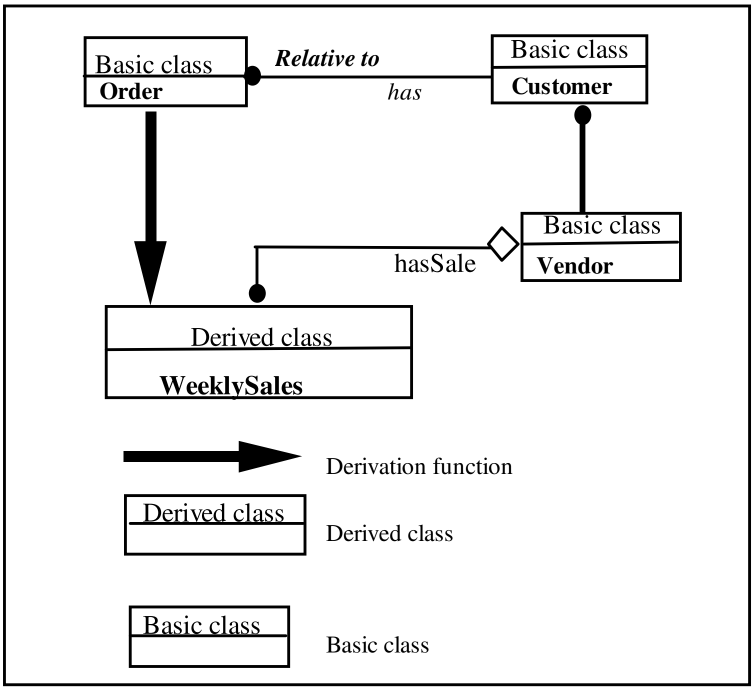


Figure 106 : Examples of basic & derived classes

A *basic class* is defined as an object class (cf. section 3.3.1.2. page 94). A basic class is described by :

- *properties* : explain the structure of objects of this class. Attributes, association and aggregation are considered as properties;
- *constraints* : allow to limit the valid values of attributes or objects;
- *operations* : represent the actions of creation, updating and deletion applied on objects of this class;
- *queries* : correspond to services applied at the class or object levels returning a value;
- *events* : describe the activities of the application depending on the state change of an object of this class.

Basic Class : Order
properties amount : REAL deliveryDate : INSTANT-A<Gregorian, day> relative to : asso (one, Customer)
operations CreationOrder (Customer, amount) type create DeletionOrder() type delete Delivery(INSTANT-A<Gregorian, day>) type delete

Figure 107 : Local description of the Order class.

A *derived class* is described by :

- *properties :* they define the structure of the derived object. A property is either a *parameter* or a *derived value*. A property is *derived* if its value is the result of a derivation function whereas a property is a *parameter* if it is an input parameter of the derivation function.
- *derivation functions :* they explain how objects are derived ,
- *queries : they* are methods for accessing to derived objects. A query can be applied on an object or on the extent of the class. For instance $averageSalesVendor and $averageSales are methods applied at the class level.
- *events* : describe the activities of the application depending on the state change of a derived object. For instance, if the sales decreases by two, a message is sent to the vendor. This is an internal event based on the WeeklySales derived class.

Derived Class : WeeklySales
properties sales : REAL **derived** relative to : inverse (one, Vendor. hasSale) **parameter**
derivation weeklysales_derivation (Vendor, INSTANT<MyWeek, week>)
queries REAL $averageSaleVendor(v : Vendor) REAL $averageSale (s:INSTANT<MyWeek, week>)
events decreasebytwo

Figure 108 : Local description of the derived class WeeklySales.

4.3.4 Extension of object classes to time dimensions

A temporal DBMS is able to manage different kinds of objects : snapshot, historical, rollback and temporal (cf. section 4.4.1.4, page 254). A snapshot object is managed in non-temporal DBMS (without time management). Historical, rollback and temporal objects are specific to temporal DBMS (with a specific time management). Valid time is managed in historical and temporal objects whereas transaction time is included in rollback and temporal objects.

This section presents how the valid and the transaction times are introduced in object classes either basic or derived. Three sub-classes are introduced *: snapshot class, temporal class and temporal dictionary class*.

- A ***snapshot class*** is a class which does not require time management. The properties of such classes are not related to time. For instance, The *Order class* is a snapshot class because the evolution of its properties is not relevant.

- A ***temporal class*** is a class with time management (cf. historical, rollback and temporal objects). It means the properties are associated to the valid or/and transaction times. The *salary of an employee* is a temporal class because its evolution has to be kept in the database.

- A ***temporal dictionary class*** is a specific class where data are accessed only with the valid time key. For instance, *the inflation-rate of a country* can be defined as a temporal dictionary if each stored value is indexed by its valid time period. The semantic of the value depends on its valid time period.

4.3.4.1 The snapshot *class*

The snapshot class is used to model objects which do not require time management. Object classes described in chapter 3 are always snapshot classes.

For instance, the *Order* class is a snapshot class because the evolution of its properties is not relevant to keep track of.

- <u>The snapshot class may be a basic or a derived class</u>.
  For instance, the *Order* class is an example of a basic and snapshot class. Changing a value of one of its property leads to replace it without keeping the previous values.

  The *WeeklySales* class is a derived class. It is considered as a snapshot class if each derivation implies to replace the old sales value by the new one.

- User-defined time in snapshot class.

  A snapshot class can integrate temporal attributes (attributes based on temporal domains). For instance, the attribute *deliveryDate* of the *Order* class is user-defined time (managed by the application).

- Properties, constraints, methods and events in snapshot class.

  A *property* of a snapshot class can be a relationship to another class or an attribute as it is defined in the object class (cf. section 3.3.1.5 page 68). The relationships between classes are aggregation, association and inheritance links.
  A *constraint* defined for a snapshot class can be an attribute constraint or an inheritance constraint as it is described in the object class (cf. section 3.3.1.7 page 82)
  Methods of a snapshot class can be queries or basic operations, and they can be applied at the object or class level. (cf. section 3.3.2.1.3 page 87). These features are defined on the object class concept and are available for the snapshot class. The types of basic operations are create, update, delete or erase. Because of the temporal consistency of the temporal database, the delete operation applied on a snapshot object is a logical deletion. The physical deletion is materialized by the erase operation.
  The derived class is defined by a set of derivation functions instead of basic operations.
  And finally, *internal events* can be expressed on snapshot classes (see internal events, section 3.3.3.1 page 91).

- Textual and graphical specification of a snapshot class.

A snapshot class may be specialized into basic or derived class.

  - Basic class : the textual specification of a basic class is similar to object class (introduced in section 3.3.1.2 page 66). For instance, the textual specification of the *copy* basic class is shown in Figure 109.

```
basic-class COPY extents Copies
    properties
        sales : real derived
        vendor: date
        date put out for loan : date
        state : copy_state
        loans : inverse (zeromany, LOAN.copy)
        notices : inverse (zeromany, NOTICE.copy)
    constraints
            unique(Copies,reference)
    operations
        COPY purchase(c:structure(COPY)) type create
        reservation() type update
        availability() type update
        check out() type update
        wear out() type update
        loss() type update
        deletion() type delete
    queries
        boolean $ exist_copy_available()
    internal-events
        availability of copy ...
endbasic-class
```

Figure 109 : Textual specification of a basic class.

The graphical notation is composed of two views : local class view (cf. section 3.3.4.5 page 108) and global structural view (cf. section 3.3.4.1 page 103). Figure 110 describes the local view of the copy basic class whereas Figure 111 illustrates the global structural view of the application.

Basic class : COPY

properties
reference : string
date acquired : date
date put out for loan : date
state : copy_state
loans : inverse (zeromany, LOAN.copy)
notices : inverse (zeromany, NOTICE.copy)

constraints
unique(Copies,reference)

operations
COPY purchase(c:structure(COPY)) type create
reservation() type update
availability() type update
check out() type update
wear out() type update
loss() type update
deletion() type update

queries
boolean $exist_copy_available()

internal-events
availability of copy

Figure 110 : Local class view of a basic class.

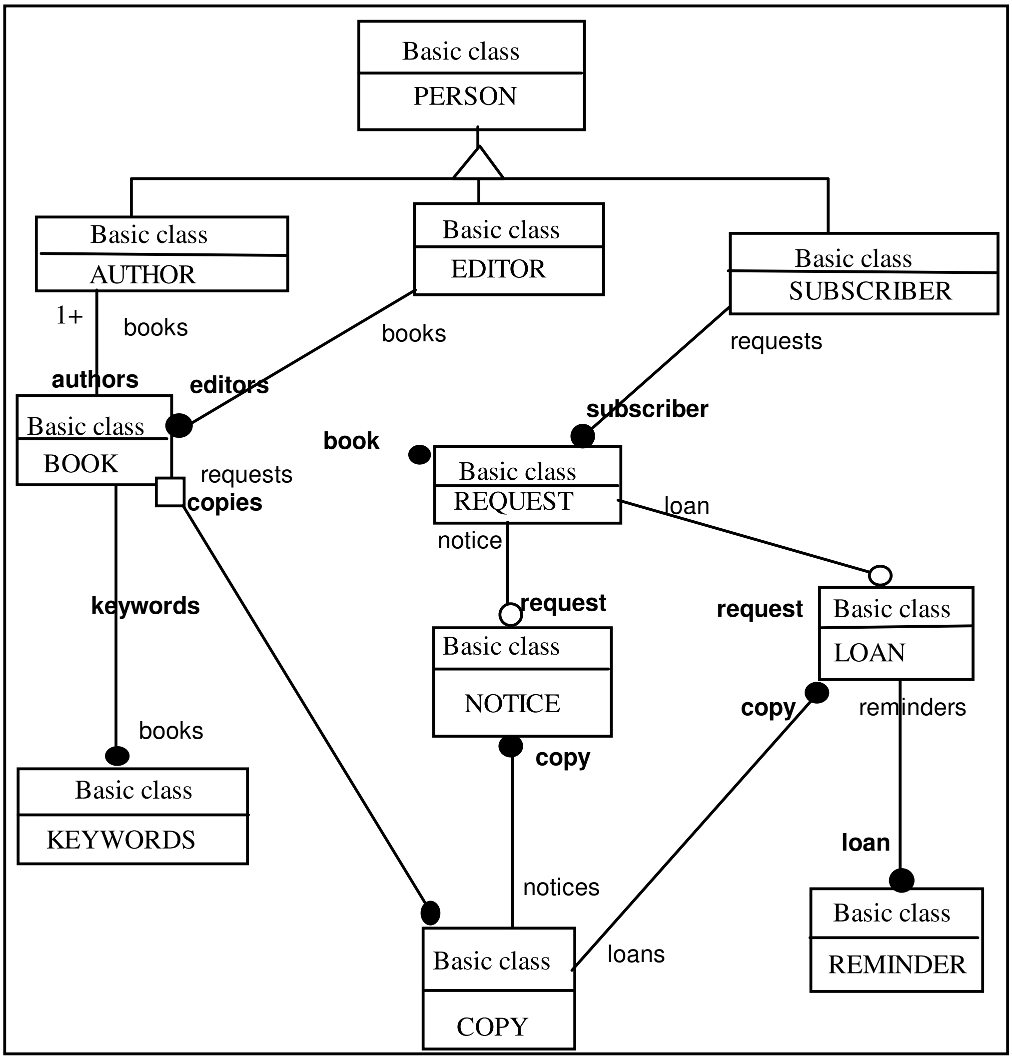


Figure 111 : Global structural view.

- Derived class : The textual specification of the *WeeklySales* derived class is illustrated in Figure 112. The local class view of the WeeklySales derived class is introduced in Figure 108 and a global structural view is given in Figure 106.

```
derived-class WeeklySales of vendor extents WSales
    properties
        sales : real derived
        relative to : inverse (one, Vendor.hasSale) parameter

    derivation
        weeklysales_derivation (Vendor, INSTANT<MyWeek, week>)
    queries
        real $ averageSaleVendor(v :Vendor)
        real $ averageSale (s :INSTANT<MyWeek, week>)
    events
        decreasebytwo ...
endderived-class
```

Figure 112 : Textual specification of a derived class.

4.3.4.2 The **temporal** *class*

The temporal class is a class with time management. It means that the properties are associated to the valid or/and transaction times. The *function of an employee* is a temporal class because its evolution has to be kept in the database.
The temporal class permits to integrate the valid or/and the transaction times management into the class definition. This concept represents the way to define historical, rollback and bi-temporal objects found in the temporal DBMS literature.

- State Vs History

The *state* of an object is composed of a set of values (a value for each property) at a given point of time. A state is associated to a timestamp (valid or/and transaction times).
A state is called *valid* when it is associated to the valid time whereas it is named *database state* when it is related to the transaction time. And finally, a *bi-temporal state* of an object integrates its valid and its transaction times.

The notion of *history* permits to keep all states of an object in order to have its evolution over time. The *valid history* of an object reflects its evolution from the real world point of view (valid time management). The *database history* of an object represents its evolution according to the database viewpoint (transaction time management). And finally, a *bi-temporal history* of an object combines the real world and database viewpoints.

The temporal class is characterized by the time dimension to manage. For each time dimension, two possibilities are offered : *last state* or *history*. The possible combinations from the *temporal class* concept are summarized in Table 4. Note that a class with no time dimension to manage is not handled by the *temporal class* concept but by *the snapshot class* concept.

		transaction time		
		none	state	history
valid time	none	no time management (snapshot class)	last database state of an object	database history of an object
	state	last state of an object with its valid time	last bi-temporal state of an object	database history of an object and its last valid state
	history	valid history of an object	valid history of an object and its last database state	bi-temporal history of an object

Table 4 : time management defined in a temporal class.

The *function of an employee* evolves over time, and the enterprise is interested by its evolution, it is modeled as a temporal class. The valid, database and bi-temporal states and valid, database and bi-temporal histories are exemplified in the following table.

Assuming that an employee *Durand* was *programmer* before becoming *project leader* (see the detailed example in section 4.4.1.4, page 254). The following table summarizes the different options for this example.

	state	history
valid	project leader, **VT**[95-12-20,forever)	programmer, **VT**[94-12-12,95-12-20*)* project leader, **VT**[95-12-20,*forever)*
database	project leader, **TT**[96-01-01 15:56:50,*'now')*	programmer, **TT**[95-01-01 9:20:15,96-01-01 15:56:50) project leader*,* **TT**[96-01-01 15:56:50,*'now')*
bi-temporal	project leader, **VT**[95-12-20,*forever),* **TT**[96-01-01 15:56:50,*'now')*	programmer, **VT**[94-12-12,*forever)*, **TT**[95-01-01 9:20:15,96-01-01 15:56:50) programmer, **VT**[94-12-12,95-12-20), **TT**[96-01-01 15:56:50,*'now')* project leader, **VT**[95-12-20,*forever),* **TT**[96-01-01 15:56:50,*'now')*

According to the information you want to retrieve, you have to define the characteristics of each time dimension you want to manage.

- The semantic of a temporal class

A temporal class is used to group properties which evolve at the same time and on which the time management required is the same. For instance, the *salary* and the *function* of an *employee* should be two temporal classes if the *salary* and the *function* are not evolving at the same time.

A temporal class describes one temporal aspect of objects identified in a snapshot class. For instance, *the function of an employee* is a temporal aspect of the *employee* class. In the OODBMS, we consider that it exists at least one property which does not change : the identity of the object. Thus, a temporal class is a temporal variation of one snapshot class. By contrast, a snapshot class can have several temporal variations. The relationship of a temporal class to its snapshot class is called « temporal aggregation » (cf. Figure 113). A snapshot class is graphically mentioned with a black box whereas a temporal class is a grey box.

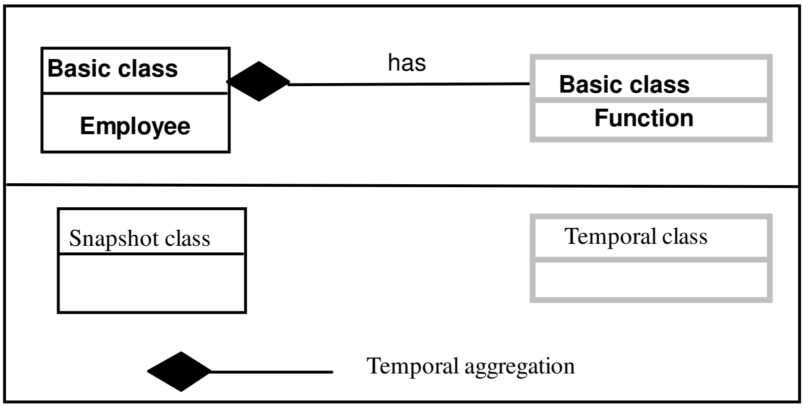


Figure 113 : Temporal class of the class Employee.

- <u>Definition of the valid time</u> (if it is required) :

  Several characteristics for the valid time have to be determined :

    - the nature of the valid time (instant or period),
    - the granularity and the calendar of the valid time,
    - the type of management (state or history),
    - the completeness of the history (complete or partial),
    - the extrapolation function if the history is partial.

  <u>The nature of the valid time</u> can be an instant or a period. The valid time is an <u>instant</u> when the application cannot control the evolution of the property but this property can be measured at certain points of time. For instance, the *weight* of a patient is measured at each visit of a *patient* to his doctor. As his weight can evolve between two visits, its validity is instantaneous.

  The valid time is a <u>period</u> when the application can control the evolution of a property or when the evolution of a property is known « a posteriori ». For instance, the *salary* and the *function* of an *employee* can be defined with a period as valid time because their evolution is controlled by the enterprise. The position of the « enemy » unit in the military system is updated « a posteriori ». We know that the *Panzer* unit was inside the *town hall* during the period *[1940/8/4 12:56:30, 1940/8/5 10 :25 :30).*

The granularity and the calendar of the valid time have to be determined according to the application. For instance, the *salary* and the *function* of an *employee* are associated to a valid time expressed at the *day* level in the *Gregorian* calendar. In addition the *WeeklySales* derived class can be historised with a valid time expressed at the *week* level of *MyWeek* calendar.

The type of management is defined by two options : state or history. The *state* option allows to manipulate the last valid state of an object whereas the *history* option permits to have all the valid states of an object. For instance, the *salary* of an *employee* should be an *history* whereas the last valid state of the *family status* of an employee is sufficient for his/her enterprise.

The completeness of the history is a characteristic applied only on an history managing the period of validity for each state. This characteristic expresses if the history is complete or partial.

- A complete history implies that the union of all periods of validity forms a period ; this period represents the life span of the object. It means that a state is known for each instant belonging to the life span of the object. For instance, the *salary* of an *employee* is a *complete history*.

- A partial history implies that the union of all periods of validity forms a set of periods (no adjacency). It means that we cannot determine a state for each instant belonging to the life span of the object. A valid history managing an instant as timestamp is considered to be a partial history. The *weight of a patient* is a partial history, because the weight between two visits is not known. A history with a period of validity can also partial, for instance the « Intelligence » department observes « enemy » units and can report on its observation by providing *position* of *unit* during a period of time.

The completeness of an history is managed as a constraint applied on the valid timestamp.

The extrapolation function can be defined if a partial history must be viewed as a complete one. The extrapolation function is used to determine the state of an object at a given instant which are not part of the history. The query operation called *get_value()* must be redefined as an extrapolation function. For instance, *the weight* of a *patient* is a partial history on which the extrapolation function must be defined.

- Definition of the transaction time (if it is required) :

  One characteristic for the transaction time has to be determined : the type of management (state or history).

  The type of management is defined by two options : state or history. The *state* option permits to manipulate the last database state of an object whereas the *history* option permits to have all the database states of an object.

- Textual and graphical specification of a temporal class.

  A temporal class is defined on a derived class or on a basic class. For instance, *Weight* of a *patient* is a *basic & temporal* class (cf. Figure 114) whereas *WeeklySales* of a *vendor* is a *derived & temporal* class (cf. Figure 115).

  The specific characteristics of a temporal class are the definition of the time dimensions to manage. The two temporal classes require the valid time management, but *Weight* is a partial history with extrapolation whereas

WeeklySales is a complete history. Note that the temporal aggregation link (between a temporal class to a snapshot class) in the textual specification is introduced by the keyword **of**.

The types of operations available in a basic & temporal class are : *create*, *evolve*, *correct*, (logical) *delete* and *erase*. Evolution and correction operations permit to distinguish an update due to an evolution of the object observed in the real world and an update due to an erroneous input in the database. Delete and erase operations allow us to differentiate logical and physical deletions. Delete operation corresponds to the logical deletion of an object (the end of valid or/and transaction periods are modified). The erase operation helps the database administrator to clean up the temporal database.

Temporal-class for basic class *Weight* **of** *patient* **extents** *Weights*
time dimensions VT INSTANT-A<Gregorian, day> **as history partial with extrapolation**
properties weight : REAL ;
constraints 0 < weight < 200 kilos ;
operations
create_weight (w : REAL, t : INSTANT<Gregorian, day>) **type create**
update_weight (w : REAL, t : INSTANT<Gregorian, day>) **type evolution**
correct_weight (w : REAL, t : INSTANT<Gregorian, day>) **type correction**
queries
REAL **get_value** (t : INSTANT<Gregorian, day>)
events ...
endtemporal-class

Figure 114 : Textual specification of a basic & temporal class.

Temporal-class for derived class *WeeklySales* **of** *vendor* **extents** *Wcollection*
time dimensions VT PERIOD-A<MyWeek, week> **as history complete**
properties
sales : REAL **derived**
derivation
weeklysales_derivation (Vendor, PERIOD-A<MyWeek, week>)
queries
real $averageSaleVendor(v :Vendor)
real $averageSale (s :PERIOD-A<MyWeek, week>)
events
decreasebytwo
endtemporal-class

Figure 115 : Textual specification of a derived & temporal class.

The temporal class concept may be used to manage only the last state of an object instead of an history. For instance, *FamilyStatus* of an *employee* can be managed as a state (cf. Figure 116).

Temporal-class for basic class *Family_status* **of** *Employee* **extents** *Fcollection*
time dimensions VT PERIOD-A <Gregorian, day> **as state**
properties status : FAMILYSTATUS ;
operations
...
events ...
endtemporal-class

Figure 116 : Example of Valid state management.

In addition, the temporal class concept can manage the valid and the transaction time. For instance, *Position* of an « *enemy* » *unit* may require a bi-temporal management with a valid history and the last database state.

Temporal-class for basic class *Position* **of** *EnemyUnit* **extents** *Pcollection*
time dimensions VT PERIOD-A <Gregorian, minute> **as partial history**
TT as state
properties position : POSITION ;
operations ...
endtemporal-class

Figure 117 : Example of bi-temporal management.

The bi-temporal management can be also applied on derived class. For instance, *WeeklySales* of a *Vendor* may require a bi-temporal management with the valid and the database histories (cf. Figure 118).

Temporal-class for derived class *WeeklySales* **of** *vendor* **extents** *Wcollection*
time dimensions VT PERIOD-A<MyWeek, week> **as history complete**
TT as history
properties
sales : REAL **derived**
derivation
weeklysales_derivation (Vendor, PERIOD-A<MyWeek, week>)
queries
real $averageSaleVendor(v :Vendor)
real $averageSale (s :PERIOD-A<MyWeek, week>)
events
decreasebytwo
endtemporal-class

Figure 118 : Example of bi-temporal management for a derived class.

The local class view assigned to the *WeeklySales* class defined in Figure 117 is similar to a local class view for a snapshot class except that the time dimensions are illustrated as well (cf. Figure 119).

Temporal & Derived Class : *WeeklySales* of *Vendor*

Time dimensions
VT PERIOD-A<MyWeek, week> as history complete
TT as history

properties
sales : REAL **derived**

derivation
weeklysales_derivation (Vendor, INSTANT<MyWeek, week>)

queries
REAL $averageSaleVendor(v : Vendor)
REAL $averageSale (s:INSTANT<MyWeek, week>)

events
decreasebytwo

Figure 119 : Local class view of WeeklySales class.

The global structural view permits to show the relationships between classes which are snapshot or temporal classes. The characteristics of time dimensions for a temporal class (cf. Figure 119) are mentioned graphically in the grey box :

VT or TT and the type of management (state or history) as well.

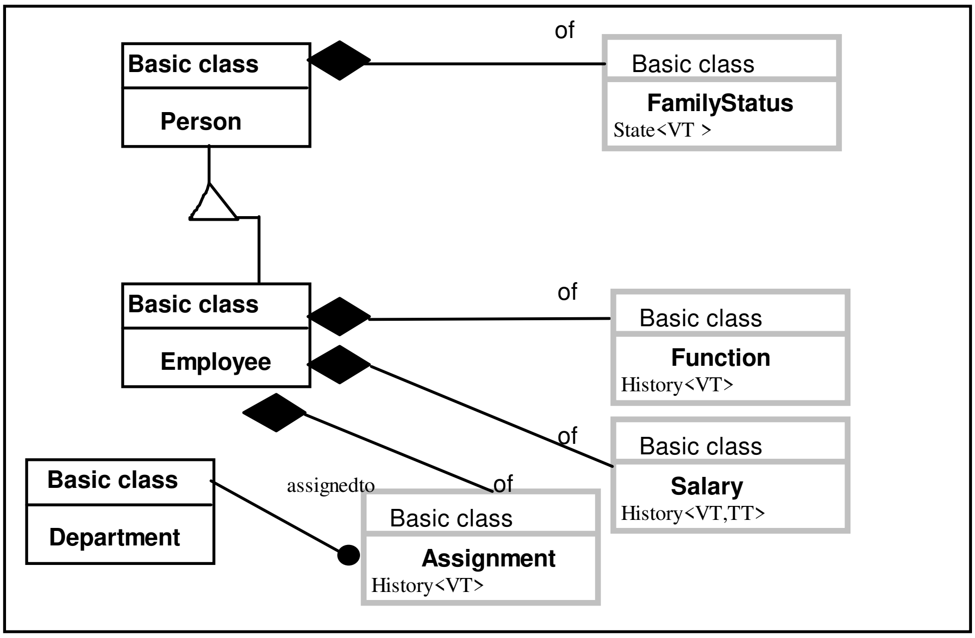


Figure 120 : Global structural view with temporal classes.

4.3.4.3 The temporal dictionary class

The dictionary is a particular collection class in ODMG-97 allowing to access to data via a key. A dictionary class is a collection of pairs (key, data).

For example, The storage unit should be designed as a dictionary where each file (data) is accessed via its address (key).

The TOOM method introduces a temporal form of the ODMG-97 *dictionary* collection called *temporal dictionary class*, where the key is the valid time period of the corresponding data. For example, the *inflation rate* of a *country* can be a *temporal dictionary class* of the snapshot class *country*, because the semantic of an *inflation rate* value depends of its valid time period.

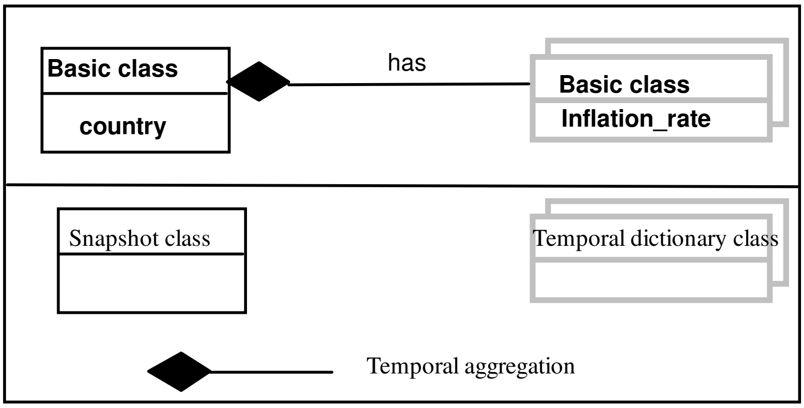


Figure 121 : Example of temporal dictionary class.

Consider a class *country*, that has a temporal variation called *Inflation_rate* class which is a temporal dictionary class. *Inflation_rate* may have entries with overlapping valid time periods as long as the exact period cannot be used with two different values. This means that for a specified country, say Greece, we may have different inflation rate values according to their valid time period (cf. Table 5 : Inflation rate of Greece as a temporal dictionary class.).

Inflation_rate	**key :** **Valid Time Period**
7.8%	[1995-01, 1996-01)
8.8%	[1995-01, 1995-07)
6.8%	[1995-07, 1996-01)
9.5%	[1996-01, 1997-01)

Table 5 : Inflation rate of Greece as a temporal dictionary class.

Note that the periods that refer to the inflation rates are overlapping. This is totally acceptable, since the inflation rate depends on the time period on which it is measured as a whole, and has no meaning for subperiods. However, a value for the inflation rate equal to, say, 9.9% and valid time period equal to [1995-01, 1995-07) would be unacceptable, since a value for this time period is already defined.

- <u>Difference between *temporal dictionary class* and *temporal class*</u>

In the temporal dictionary, the valid time periods can overlap because the valid time is only used to access the value.
The *inflation rate* of a *country* was not considered as a temporal class because the different values can have different semantics depending of their valid time and all the values do not build the evolution of the *inflation rate*.

If *inflation_rate* was designed as a temporal class with valid time and history management, the valid time periods will not overlap and the values taken by the attribute *inflation_rate* would have the same semantic. Table 6 describes the temporal class *inflation_rate* for the country *Greece*. The inflation rate attribute provides the annual rate (periodicity is fixed). With the *inflation_rate* temporal class, one can examine the evolution of the inflation rate and might define an extrapolation function if required.

Inflation_rate	**Valid Time**
7.8%	[1995, 1996)
9.5%	[1996, 1997)

Table 6 : Inflation rate of Greece as a temporal class.

If the application requires the monthly and yearly rates for each country, two temporal classes would be necessary : *Monthly_Inflation_rate* and *Yearly_Inflation_rate*.

- <u>Time dimensions definition</u>

- Definition of the valid time: the key of the temporal dictionary is the valid time period. The valid timestamp is a period, several values can be managed but are not considered as an history. The granularity and the calendar of the valid time has to be determined. The *inflation_rate* temporal dictionary class exemplified in Table 6 has a valid time expressed at the *month* level of the *Gregorian* calendar.

- Definition of the transaction time (if it is required) :

  The transaction time can be managed if the database history or the last database state for each key is required. Table 7 illustrates the transaction management inside the *inflation_rate* temporal dictionary class. A database history is managed for each key. It permits to know the successive values stored in the database with their transaction time. For instance, at the instant « 1996-03-25 10 :15 :45 », the inflation rate for the period [1995-01, 1996-01) was known as 7,6% was updated a posteriori at the instant « 1996-03-26 11 :40 :12 » to 7,8%.

Key : Valid time	database history	
	transaction time	inflation_rate
[1995-01, 1996-01)	[1996-03-25 10 :15 :45 , 1996-03-26 11 :40 :12)	7,6%
	[1996-03-26 11 :40 :12 , 'now')	7,8%
[1995-01, 1995-07)	[1996-08-25 12 :15 :50 , 1996-09-26 11 :40 :55)	8.5%
	[1996-09-26 11 :40 :55, 'now')	8,8%
[1995-07, 1996-01)	[1996-02-25 10 :15 :45 , 1996-03-26 11 :40 :12)	6.7%
	[1996-03-26 11 :40 :12 , 'now')	6,8%
[1996-01, 1997-01)	[1997-02-10 10 :15 :45, 'now')	9.5%

Table 7 : Example of temporal dictionary class with database history management.

One characteristic for the transaction time has to be determined :

- the type of management (state or history),

The type of management is defined by two options : state or history. The *state* option allows to manipulate the last database state of a data whereas the *history* option permits to have all the database states of a data. The history management is shown in Table 7 whereas the state management is illustrated in Table 8.

Key : Valid time	database state	
	transaction time	inflation_rate
[1995-01, 1996-01)	[1996-03-26 11 :40 :12 , 'now')	7,8%
[1995-01, 1995-07)	[1996-09-26 11 :40 :55, 'now')	8,8%
[1995-07, 1996-01)	[1996-03-26 11 :40 :12 , 'now')	6,8%
[1996-01, 1997-01)	[1997-02-10 10 :15 :45, 'now')	9.5%

Table 8 : Example of temporal dictionary class with database state management.

- Temporal dictionary derived class

The *temporal dictionary class* is useful for derived classes. For instance, *Sales* is a derived class evaluating the sales relative to a vendor. The sales are evaluated weekly, monthly and quarterly. Thus, the semantic of the attribute *sales* is dependent of its period of time. If the period has a month duration it means that the *sales* represents the sales performed during one month. Conceptually, the analyst should define as many derived classes as temporal granularities required by these statistics. If the sales per vendor are evaluated each week, each month, each quarter and each year, 4 derived classes *: WeeklySales, MonthlySales, QuarterlySales, and YearlySales* should be defined. Practically, it can appear difficult to determine in advance the temporal periodicity of the reporting. The periodicity of the reporting can evolve during the life of the application. The « temporal dictionary » class offers a flexible way to describe this situation (cf. Figure 122 : Example of Derived & Temporal dictionary class.).

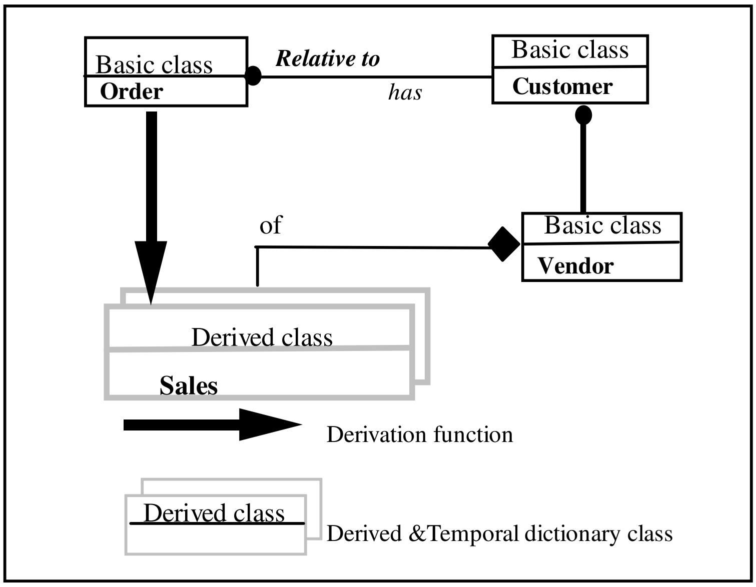


Figure 122 : Example of Derived & Temporal dictionary class.

- Textual and graphical specification of a temporal class.

A temporal dictionary class is associated to a basic or a derived class. Its definition is close to the definition of a temporal class. For instance, Figure 123 illustrates the textual definition of the *inflation_rate* class of a *country*. The valid time is a key of the dictionary and the transaction time is defined as an implicit time management.

Temporal-dictionary class for basic class *Inflation_rate* **of** *country* **extents** *Infl_set*
key VT PERIOD-A<Gregorian, day>
time dimension TT as history
properties inflation_rate : REAL ;
operations
...
events ...
endtemporaldictionary-class

Figure 123 : Textual specification of a basic & temporal dictionary class.

In the same way, we can define *Sales* of a *vendor* as a derived & temporal dictionary class.

Temporal-dictionary class for derived class *Sales* **of** *Vendor* **extents** *Sales_set*
key VT PERIOD-A<Gregorian, day>
time dimension TT as history
properties sales : REAL **derived**
derivation...
endtemporaldictionary-class

Figure 124 : Textual specification of a derived & temporal dictionary class.

The local class view is similar to a temporal class, except that the valid time is mentioned as a key (cf. Figure 125). The global structural view with a temporal dictionary class is shown in Figure 121 and Figure 122.

Temporal dictionary & Derived Class : ***Sales*** **of** ***Vendor***
Key : VT PERIOD-A<Gregorian, month> **Time dimension : TT as history**
properties sales : REAL **derived**
derivation **...**
queries **...**
events **...**

Figure 125 : Local class view of a temporal dictionary class.

4.3.4.4 Temporal analysis process

The introduction of time in the structural definition of classes leads to the process of decomposition of an object class to its temporal variations. The three following questions are examined :

- What properties are not modified during the life span of objects ?
- What properties evolve during the life span of objects ?

- What properties are relevant as timestamp ?

A *salary* of an *employee* is a property which evolves during the life span of the employee. By contrast, the *social security number* of a *person* does not change. A *telephone number* of a *person* can change during his life span but its evolution is not necessarily relevant for the application.

A property is called *permanent* when the property does not change or its evolution has no interest. For instance, *social security number* and *telephone number* are permanent properties of a person.

A property is named *variable* when the property changes during the life span of the object and when its change should be related to time. For instance, the *salary* of an *employee* is a variable property because the enterprise has to keep its different values and to relate each value to its valid time.

The permanent properties of an object are grouped into a *snapshot class*. A *temporal class* allows to group variable properties changing at the same time and related to the same timestamp.

A temporal class is a temporal variation of a snapshot class. A snapshot class can have several variable classes. A variable class is always related to a snapshot class because the identity of an object is a property which does not change during the life of an object.

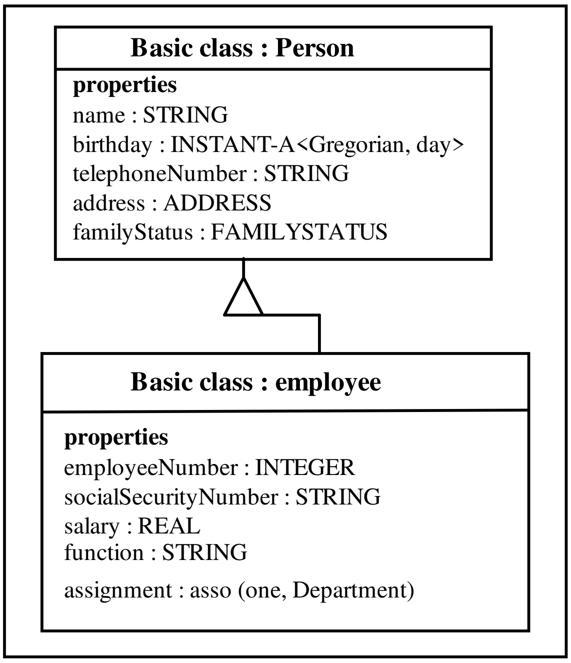


Figure 126 : Examples of basic classes.

Figure 126 presents two basic classes *employee* and *person* related by an inheritance link. The *employeeNumber*, *socialSecurityNumber* properties are considered permanent whereas the attributes *function*, *salary* and *assignment* are variable properties. Three temporal classes are defined (Figure 127) because these three properties do not change at the same time. The *FamilyStatus* attribute can change for a person, but the application requires only to timestamp the last status of a person ; a temporal class is introduced.

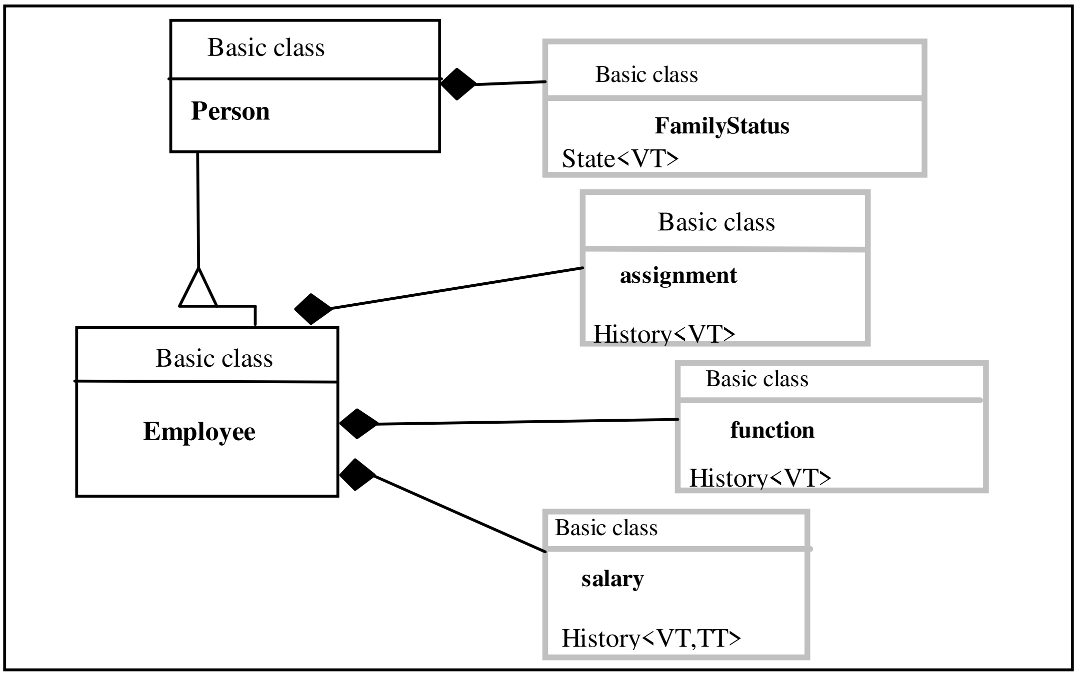


Figure 127 : Examples of snapshot & temporal classes.

This process of decomposition of an object into its temporal variations can also be applied on derived classes. The *WeeklySales* class allows to determine and store the *amount of sales* realized by a *vendor* for a given week. The *weekly amount of sales* realized by a *vendor* evolves during the life of a vendor. The manager of the Sales department is interested by the evolution of the weekly amount of sales per vendor. Thus, *WeeklySales* is a temporal class as it is shown in Figure 128.

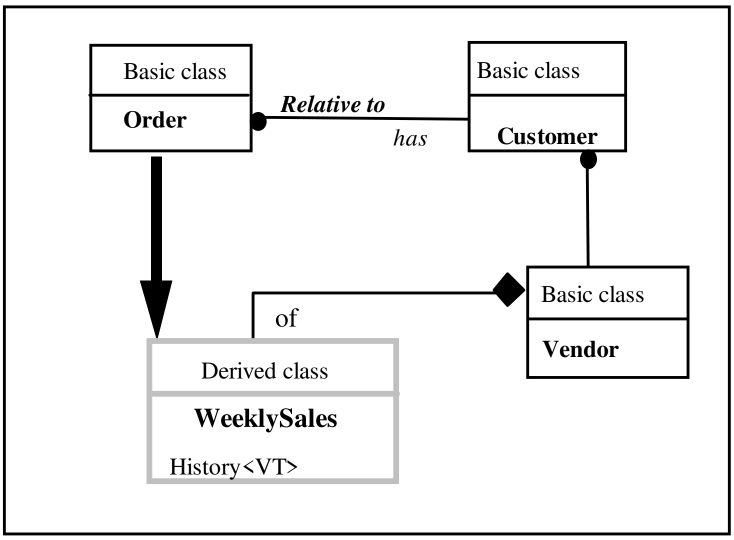


Figure 128 : Example of derived variable class.

4.3.5 Extension of static links between classes to time dimensions

The static links between object classes are aggregation, association and inheritance (cf. section 3.3.1.5 page 68).

- Aggregation

  The aggregation link existing in the OO method is used to express the composition of objects of snapshot classes. For instance, the *order* class is composed of *order lines*. This aggregation is time independent.

  The *temporal aggregation* is a kind of aggregation between a snapshot class and a temporal class or a temporal dictionary class. For instance, an *employee* has a *salary* and a *function* which are evolving during the career of the employee. The temporal variation is denoted graphically with a black diamond (cf. Figure 129).

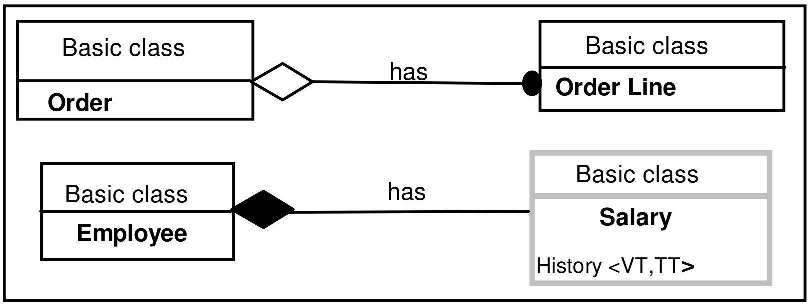


Figure 129 : Examples of aggregation.

- Association

  The association is defined in the OO method. It is applied on snapshot or temporal or temporal dictionary classes. Two kinds of relationship exist : an association referring to an object and association referring to a state (state of an object at a given instant).

  A person owns a car (a) and *an order is related to a customer* (b) are examples of association on objects. But, *a client has a version of a software* (c) is an association referring to a state of an object, such associations are *called state associations* (cf. Figure 130).

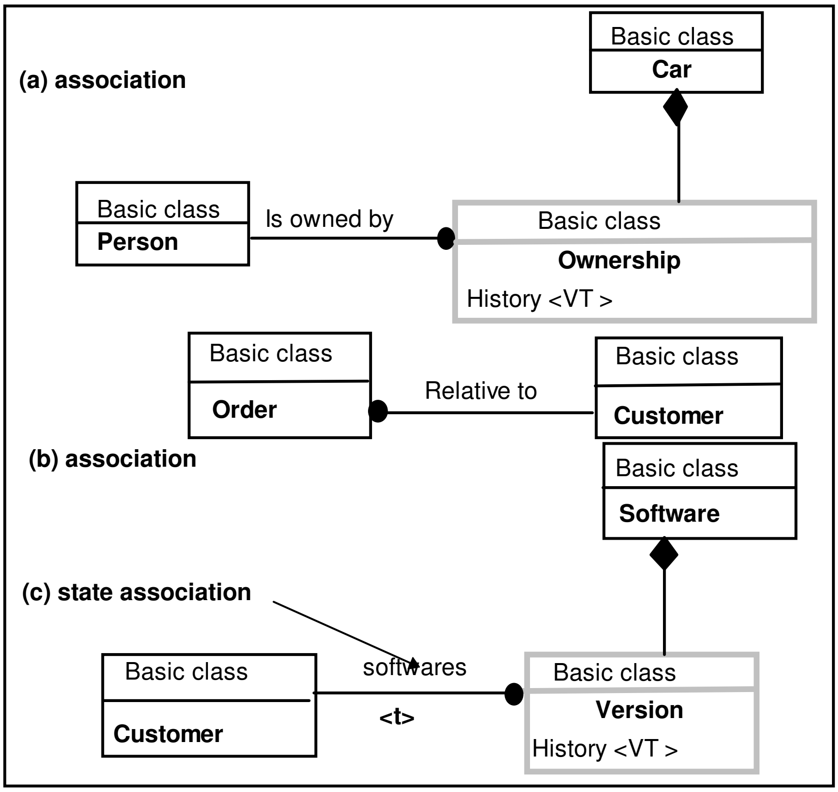


Figure 130 : Examples of association.

An <u>association</u> between a snapshot class and a temporal class expresses a link between objects. As the target objects are time dependent, the link provides the whole history of the target. For instance, the *association between person and ownership classes* expresses that a *person owns a car*, and *a car stored all its owners over time in the temporal class ownership*.

A <u>state association</u> expresses the fact that an object is linked to a specific state of an other object. It means that the time of the state to refer to is mentioned in the association state. Consequently, the identity of the referred state of an object is the OID of the object and the time of the required state (temporal OID). For instance, *a customer has a particular version of a software* ; it means a state of a software object. Figure 130 illustrates the graphical notation of a state association (case c).

The textual specification of a state association is the following :

```
basic-class Customer extents Customers
        properties
                name : string
                addressr: address
                sofwares : state asso (zeromany, Version) // state association
        ...endbasic-class
```

- <u>Inheritance</u>

    The inheritance mechanism can be applied only between snapshot classes. For instance, the *employee* class *inherits from* the *person* class. It means that an employee inherits from the person class all the structural properties as well as the behavioral properties ; in particular all the temporal variations, e.g. the *family_status* class. The inheritance is prohibited with temporal and temporal dictionary classes.

4.3.6 Extension of constraints applied on object classes

In the definition of an object class, we can define constraints that should be satisfied by its instances. The kinds of constraints available in the OO method are :

- ***uniqueness constraint*** : it allows to define that a value of an attribute or a set of attributes is unique in a specific collection. For instance, *the socialSecurityNumber of a person is unique in the set of persons (Persons)* is described as follows : *unique (socialSecurityNumber, Persons).*

- ***referential constraint*** : it is implicitly defined when an association exists between two classes. It means that a property of an object must refer an object belonging to another object class. For instance, *an order is relative to a customer* leads to the property *relative to* in the **Order** class which <u>must refer</u> to an object of the *Customer* class.
- ***cardinality constraint*** : it is embedded in the definition of an association or an aggregation. It expresses how many objects participate in the relationship. For instance, *an order is relative to <u>only one</u> customer*, *but a customer can have <u>many</u> orders*. The cardinality expresses that the property *relative to* in the *Order* class refers to only one customer whereas the inverse link *has* in the *Customer* class is a set of order references.
  The cardinality is considered instantaneous. It means that at a given instant, how many customers are associated to an order and at a given instant, how many orders are associated to a customer. The permanency of the association is not treated in the cardinality (i.e. the fact that an association between an order and a customer does not change during the life span of an order). By contrast with the following example : a car is owned by only one person at a time ; it means that the car can change its owner during its life span.

- ***attribute constraint*** : it is an assertion applied on properties of an object (cf. Part II). *The weight of a patient can not exceed 200 kilos or the birthday of a patient can not be posterior to 1800* are examples of attribute constraints.

The uniqueness and attribute constraints are explicitly described by the analyst whereas the referential and cardinality constraints are embedded in the definition of associations and aggregations. The inheritance constraint is not considered in the temporal extension because the inheritance link is not applied on temporal and temporal dictionary classes.

4.3.6.1 Classification applied on constraints

TOOBIS combines the features of an OO DBMS and of a temporal DBMS.

- In the active OO DBMS such as ODE[3], constraints are classified in two groups ***: intra-object*** and ***inter-object*** constraints. This distinction allows us to determine when they have to be verified. An intra-object constraint is defined locally to an object and can be checked at the end of each method execution. But an inter-object constraint uses several objects, thus it has to be checked at the end of the database transaction. A *referential constraint* is an example of inter-object constraint whereas *the weight of a patient can not exceed 200 kilos* is an example of intra-object constraint.

Constraint	intra-object	inter-object
uniqueness		X
referential		X
cardinality		X
inheritance		X
attribute	X[4]	X[5]

Table 9 : Classification of the constraints.

The second classification comes from the temporal database field[6]. They group the constraints in two main categories : ***intra-time*** and ***inter-time*** constraints.

An intra-time constraint is only dependent of an instant t. Each instant t should verify the constraint. Three semantics of constraints are derived from the time dimensions to consider : intra-VT, intra-TT and intra-Bi constraints.

- An intra-VT constraint is related to the snapshot view of the same real world excerpt at a given instant t. For instance, *for each week declared in the Weekly effort, an assignment of this employee for this project must exist*.
- An intra-TT constraint relates to the snapshot view of the database at a given instant t. For instance, *the name of a company must be unique in each database state*.
- An intra-Bi constraint relates to the snapshot of the same real world part at an instant vt stored in the database snapshot at an instant tt. For instance, if the weight of patient is stored in the temporal class with history<VT, TT>, *the valid time of a weight value cannot be superior to its transaction time*.

[3] Gehani N., Jagadish H.V., *Ode as an active database : constraints and triggers*, Proc. of the 17th VLDB, Barcelona, Spain, 91, pp. 327-336

[4] attribute contraints without using another object in its definition.. Example, *The weight of a patient can not exceed 200 kilos*.

[5] attribute contraints using other objects in its definition. Example, The temporal distance between two visits can not be superior to one month.

[6] Michael H. Böhlen, *Valid time integrity constraints*, 1994.

An inter-time constraint is defined by using information available or valid at different instants. Each instant t should verify the constraint. Three semantics of constraints are derived from the time dimensions to consider : inter-VT, inter-TT and inter-Bi constraints.

- An inter-VT constraint uses in its definition several snapshots of the real world. For instance, *the valid timestamp must be unique in the temporal class with history<VT>.*
- An inter-TT constraint uses in its definition several snapshots of the database. For instance, if the name of a person is in the temporal class with history<TT>, *the name of a person cannot change more than three times.*
- An inter-Bi constraint uses several snapshots of the real world stored in different database snapshots. For instance, if the weight of patient is stored in the temporal class with history<VT, TT>*, the weight of a patient for a specific valid time cannot be corrected more than three times.*

The application can change the past, future and present data about the real world but it can only change the current state of the database.

The intra-time constraints are easy to verify, they should be applied on the snapshot of the real world and the snapshot of the database modified by each database operation. But, the inter-time constraints are more expensive to check because after each database operation each snapshot (VT and TT) of the temporal database should be checked otherwise the database could be inconsistent.

4.3.6.2 Extension of constraints to time dimensions

- The ***cardinality constraint*** is included in the definition of an aggregation or an association. The cardinality by convention is instantaneous, thus the significance does not change if the classes participating to the association are temporal or temporal dictionary classes.

The following examples illustrate the point.

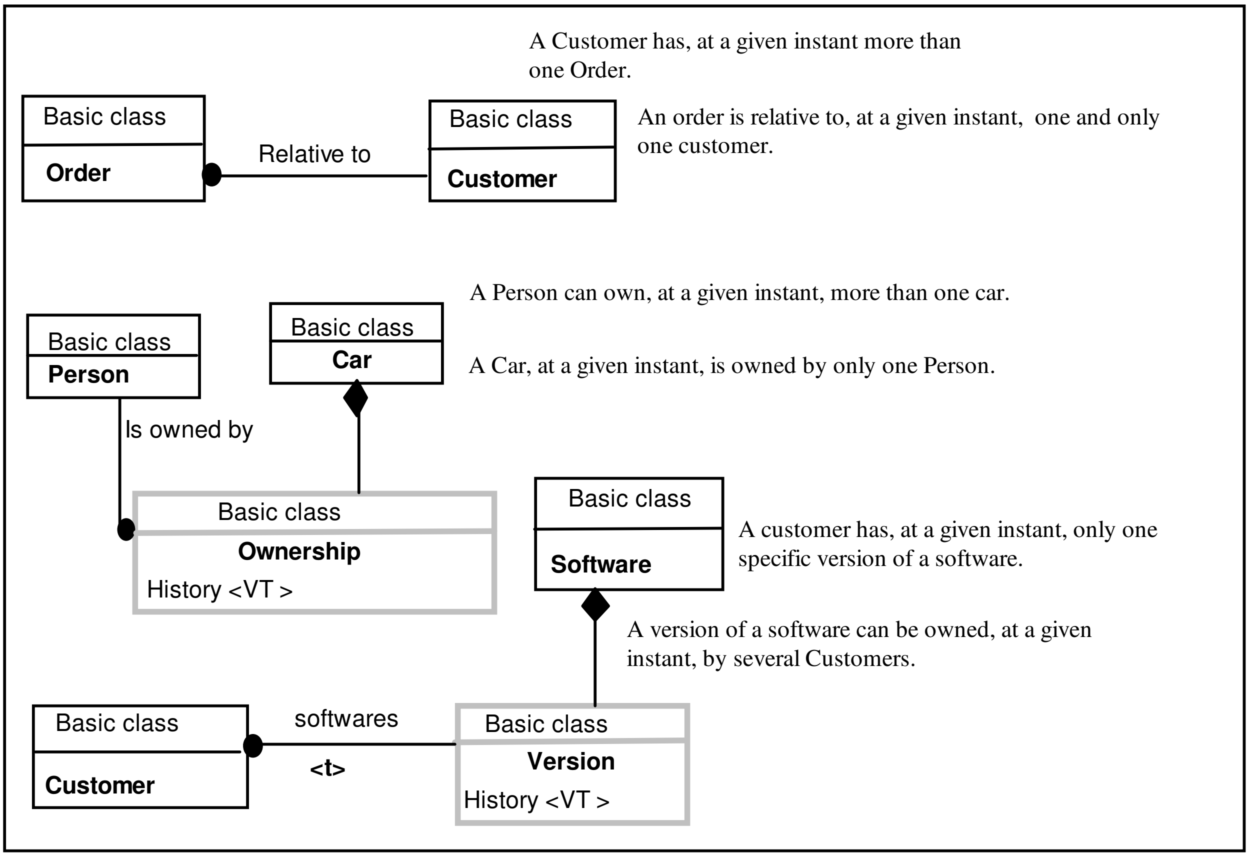


Figure 131 : Examples of cardinality constraints.

- The ***referential or temporal existence constraint*** is embedded in the aggregation and association definitions. But, three different referential constraints are introduced according to the time to consider.
  - The ***non temporal referential*** constraint is applied on snapshot classes. For instance, the attribute *relative to* in the *Order* class refers to an object of *Customer* class.
  - The ***temporal existence*** constraint is applied on temporal or temporal dictionary classes. For instance, the attribute *is owned by* in the *Ownership* temporal class with History<VT> refers to an object of the *Person* class which should exist at the valid time.
  - The ***temporal referential*** constraint is defined on state-association which refers to the state of an object at a given instant. For instance, the *customer* refers to a *version of a software at a given instant*.

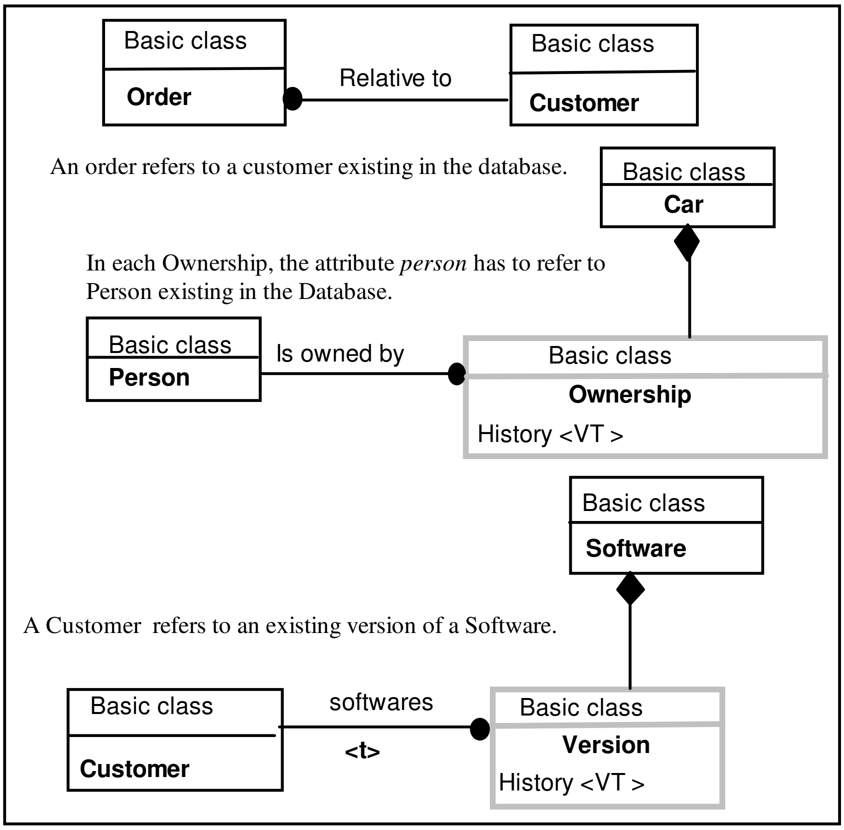


Figure 132 : Examples of referential constraints.

- The ***uniqueness constraint*** is also defined for temporal class and temporal dictionary class. According to the time integration in the collection to consider, 3 different uniqueness constraints are introduced :
    - The ***uniqueness*** constraint defines on the non-temporal collection. For instance, the attribute *SocialSecurityNumber of a person* must be unique in the set of *Persons* (collection associated to the class *Person*).
    - The ***local temporal uniqueness*** constraint is applied inside an object. For instance, *each instant belonging to the VT period* must *be unique* inside the *ownership* (temporal class with history<VT>) of a *car*.
    - The ***global temporal uniqueness*** constraint is applied on a temporal collection relative to several objects. For instance, the *immatriculationNumber of a car at a given instant* must be unique *in the set of cars at every instant*.

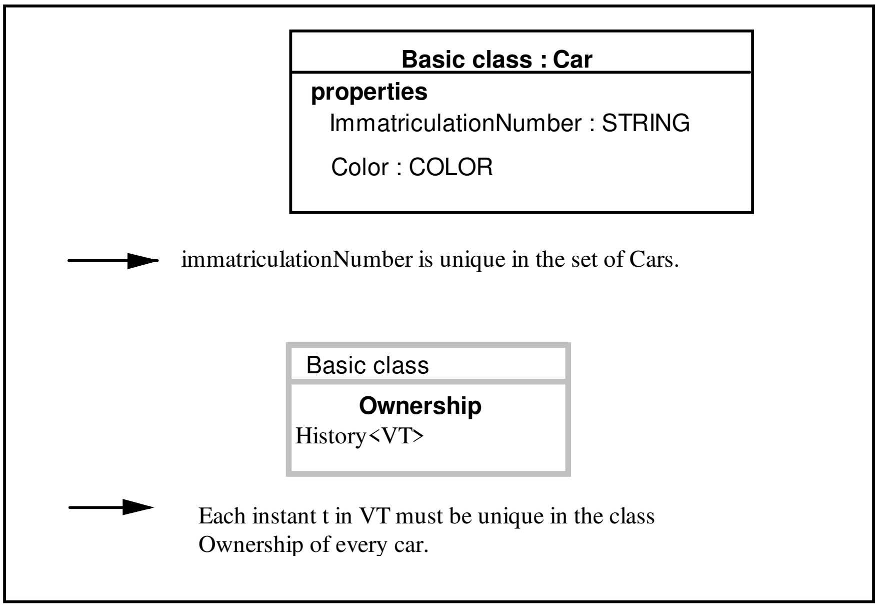


Figure 133 : Examples of Uniqueness constraints.

- The ***attribute constraint*** is defined for any kind of classes. They can refer to one or more objects and they can use one or more times. Below, is an example for each possible category (cf. Table 10).

	intra-object	inter-object
without time	age of a person is less than 200 years	date of visit2 < date of visit1 + 1,5 months
intra-vt	for each snapshot of the real world, effort declared by an employee for a project and for a week (temporal class with history<VT>) < 60 hours	effort declared by an employee for a week = sum of efforts per project declared by this employee for this week.
intra-tt	for each database snapshot, weight of a patient (temporal class with History<TT>) is less than 200 kilos	for each database snapshot, the assignment of an employee to a project is allowed if they belong to the same department
intra-Bi	for each snapshot of the real world stored in a database snapshot, effort declared by an employee for a project and for a week (temporal class with history<VT, TT>) < 60 hours	for each snapshot of the real world stored in a database snapshot, effort (temporal class with History<VT, TT>) declared by an employee for a week = sum of efforts per project declared by this employee for this week.
inter-vt	the salary of an employee (temporal class with History<VT>) can not be updated by increasing it by two	the salary of an employee (History<VT>) can not be increased by a percentage superior to the limit defined by department for the same period
inter-tt	the name of a person (temporal class with History<TT>) can not change more than three times	a person can not be employed two times by the same company with a lower salary (temporal class with History<TT>)
inter-Bi	the weight of a patient (temporal class with History<VT, TT>) for a specific valid can not be corrected more than three times	. a reviewer assigned to a project (temporal class with History<VT, TT>) should not belong to a company of the consortium at any time in the past or present time.

Table 10 : Categories of attribute constraints.

The classification of the constraints according to the two frameworks intra/inter object/time are helpful during the design phase for the mapping of the conceptual specification into the TOODBMS environment (TODL, TOQL, TODM and C++ binding of O2). The TOOBIS environment is not an active DBMS managing constraints and triggers. Consequently the concept of constraints and events (concept close to the trigger concept of the active

DBMS) introduced in the TOOM method have to be mapped into methods and database transactions. The classification of constraints is defined at the analysis phase but it is used and exploited at the design phase for mapping the conceptual to the logical specification.

The summary of the possible cases is given in the following table :

	intra-object	**inter-object**	**without time**	**intra-VT**	**intra-TT**	**intra-Bi**	**inter-VT**	**inter-TT**	**inter-Bi**
cardinality		**x**	**x**	**x**[7]	**x**[8]	**x**[9]			
referential		**x**	**x**						
temporal existence		**x**		**x**[10]	**x**[11]	**x**[12]			
temporal referential		**x**					**x**[13]	**x**[14]	**x**[15]
uniqueness		**x**	**x**						
local temp. uniqueness	**x**						**x**[16]	**x**[17]	**x**[18]
global temp. uniqueness		**x**					**x**[19]	**x**[20]	**x**[21]
attribute	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**

Table 11 : classification of constraints with the two frameworks.

The constraints are specified inside the class definition (snapshot, temporal and temporal dictionary classes). Table 12 defines how the kinds of constraints are specified in the specification language. In addition, for each constraint, its category has to be determined according to the two frameworks :

- intra-object or inter-object (OODBMS framework),
- intra-VT, intra-TT, intra-Bi, inter-VT, inter-TT, inter-Bi (temporal DBMS framework).

---

[7] *for each snapshot of the real world, a car is owned by only one person. (ownership is History<VT>)*

[8] *for each database snapshot, a car is owned by only one person (ownership is History<TT>)*

[9] *for each snapshot of the real world stored in a snapshot database, a car is owned by only one person (ownership is History<Bi>)*

[10] *the attribute owned by in the Ownership History<VT> class refers to an object of Person class which should exist at the valid time.*

[11] *the attribute owned by in the Ownership History<TT> class refers to an object of Person class which should exist at the transaction time.*

[12] *the attribute owned by in the Ownership History<Bi> class refers to an object of Person class which should exist at the valid and the transaction times*

[13] *the customer refers to a software at a given VT instant (version is an History<VT>).*

[14] *the customer refers to a software at a given TT instant (version is an History<TT>).*

[15] *the customer refers to a software at a given VT and TT instant (version is an History<Bi>).*

[16] *each instant belonging to the VT period must be unique inside the ownership history<VT> of a car*

[17] *each instant belonging to the TT period must be unique inside the ownership history<TT> of a car*

[18] *? ?*

[19] *the ImmatriculationNumber of a car (History<VT>) must be unique in the set of cars at every instant*

[20] *the ImmatriculationNumber of a car (History<TT>) must be unique in the set of cars at every instant*

[21] *the ImmatriculationNumber of a car (History<Bi>) must be unique in the set of cars at every instant*

The determination of the category of each constraint should be used when the specification is mapped onto the TODL/TOQL environment.

	How to specify it
cardinality	embedded in aggregation, association and state association
referential	embedded in aggregation, association and state association
temporal existence	specified as a TOQL statement
temporal referential	embedded in state association
uniqueness	defined as following : **unique** (<attribute-list>, <collection>)
local temp. uniqueness	defined as following : **temporal_unique** (**local**, <attribute-list>, <collection>)
global temp. uniqueness	defined as following : **temporal_unique** (**global**, <attribute-list>, <collection>)
attribute	described as a TOQL statement or a boolean function

Table 12 : How to specify the different kinds of constraints.

4.3.7 Extension of events to time dimensions

Event is the concept used in the OO method to define the dynamic of the application (cf. chapter 2). Three kinds of events are defined :

- an external event materializes the emission of a message by an actor,
- an internal event acknowledges a noteworthy state change of an object,
- a temporal event is a particular state of the clock.

The introduction of the time dimensions in the database leads to take into account the valid and the transaction times into the event concept.
The occurrence of an event happens at a given instant. This instant is the transaction time of this event. The valid time of an event depends on the kind of event.

4.3.7.1 External event

The emission of a message by an actor is an external event. The trigger part of this event represents the treatment to perform when such messages arrive to the system. This treatment is the answer of the IS to this message. The valid time of an external event is the valid time associated to the message. *An event in time* is an event whose valid time is the present time. *An « a priori » event* has a valid time in the future whereas an *« a posteriori » event* has a valid time in the past.

The events described with the basic OO method (cf. section 3.4.4 page 118) are event in time. The « *a priori* » and « *a posteriori* » events are exceptional cases, which have to be taken into account in temporal databases. These two kinds of events are managed in the TOOM method as follows :

- An « a priori » event allows to trigger treatment in advance. In order to avoid problems relative to the management of post-actions, the acquisition of the message is done in advance but the treatment associated to the event is executed in time.

- An « a posteriori » event is limited to exceptional cases aiming at correcting the modeled real world stored in the database. For instance*, the correction of the customer's account related to a registered cheque in a bank* is an example of « a posteriori » event. The treatment associated to this event has to correct the situation by erasing the treatment made with the « wrong » account and apply the treatment with the « right » one. In the same manner, the event *« input the salary of an employee with three months delay* » requires to redo the payment treatment for these three months. The « a posteriori » events are exceptional situations of the application requiring exceptional business rules to apply. The corrective treatment is in charge of the application layer. The temporal DBMS does not provide retro-action functionalities.

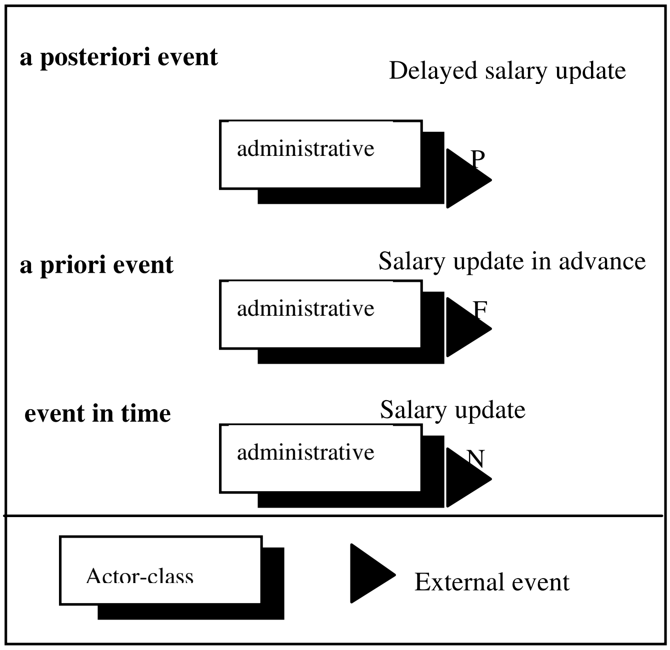


Figure 134 : Examples of external events.

- Textual specification of the three kinds of external events

  The textual specification of an external event is embedded in the definition of an actor class. The external event is composed of a message, a predicate and a trigger part and optionally the induction links and concurrent events can be described. The validity time of a message is mentioned by a keyword **past, future or in time**.

DelayedSalaryUpdate
past message DELAYEDSALARY
attributes
employee_id : INTEGER ;
salary : REAL ;
VT : PERIOD<Gregorian,month>
predicate « the employee must exist at this VT period»
trigger ...

4.3.7.2 Internal event

The valid time of an internal event is restricted in the TOOM method to the « event in time ».

Internal events based on snapshot classes can only be ascertained on the change of the current state of an object. Thus, the event is considered in time.

For the temporal classes with history, the « a priori » and « a posteriori » events are forbidden. Only change of the current state can trigger internal events. This limitation avoids the problem of automatically triggering retro-actions and post-actions raising the problem of consistency assurance.
Classically the predicate of an internal event expresses the state change of an object ; for instance, *if the stock of a product becomes less than the replenishement level*, a message is sent to the stock manager. This event is called *event on object* because it is based on the state change of the stock object. The extension of object classes to temporal classes with history implies a new class of events based on the state of the history instead of the state of an object. For instance*, If an employee has increased his salary four times during the last three years*, the staff department is alerted. The predicate of this event is based on the historical state changes. When the fourth update representing an increase of the salary is performed, the event occurs and its treatment is triggered. This kind of event is called *event on history*.

The *event on object* are applicable on all kinds of classes : *snapshot*, *temporal class*, and *temporal dictionary*. But the *event on history* is defined only on temporal classes with history or temporal dictionary with history<TT>.

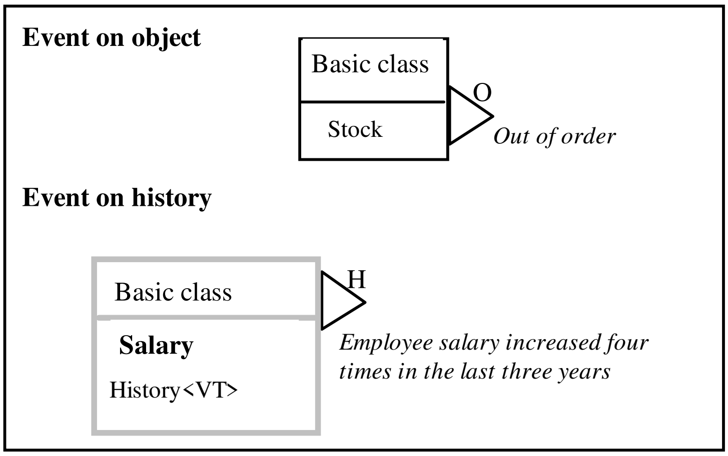


Figure 135 : Examples of internal events.

- <u>Textual specification of the three kinds of internal events</u>
  The textual specification of an internal event is embedded in the definition of a snapshot class, temporal class or temporal dictionary class. The internal event is composed of a predicate and a trigger part and optionally the induction links and concurrent events can be described. The type of internal event (on object or on history) is mentioned by the keywork **on object** or **on history**.

Out of stock **on object**
predicate « stock quantity is less than the limit fixed for the product»
trigger ...

4.3.7.3 Temporal event

A temporal event is a temporal expression based one of the calendars of the application. If the application requires several calendars, temporal events are local to a calendar. Each calendar class is characterized by the definition of the calendar explained in section 4.4.1.3 (page 213) and the set of temporal events based on the clock of this calendar.
Each calendar class is realized at the instance level by a clock. By default, a DBMS provides the clock of the *Gregorian* calendar. Each clock of the application is synchronized on the system clock.
Three types of temporal events are defined : absolute, relative and periodic events. For instance, the *«25th of December 1996»* is an absolute event, *« each 25th of December»* is a periodic event and *«1 month after the expiration of the loan»* is a relative event.
The transaction time of a temporal event is the instant when the event is acknowledged. The valid time of the temporal event is a period during which its predicate is true. In a temporal database, a time occurrence of a temporal event belongs to its valid. If the period of validity attached to temporal events has a duration less than *the minute granularity*, the application is considered as a characteristic of real time system application.

In the temporal extension TOOM, the relative event is refined according to the relative time domains defined in section 4.4.1.3 (page 213).
The relative event has a relative time expression as predicate. Two kinds of temporal marks are possible : *temporal attribute* or *time occurrence of event*.

- *«1 month after the expiration of the loan»* is an ***event relative to an object*** of the loan class and the temporal mark is the attribute *end of loan*. The valid time of the event is derived from the attribute used as temporal mark.
- *«15 days after the arrival of an order »* is an ***event relative to the event*** *« arrival of an order »*. The valid time of this event is derived when the event *« arrival of an order »* occurs. The temporal mark used in this case is the time occurrence of *the « arrival of an order »* (transaction time).

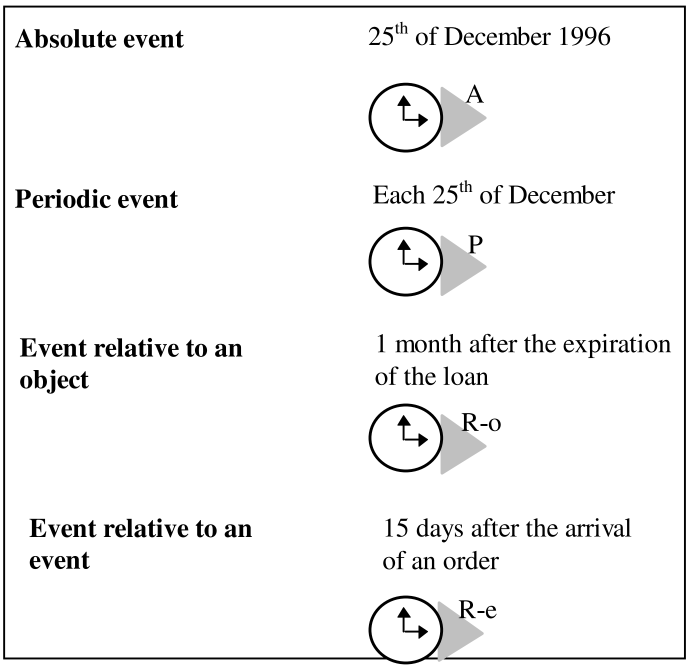


Figure 136 : Examples of temporal events.

- Textual specification of the three kinds of temporal events

The textual specification of an temporal event is embedded in the definition of a calendar class. The temporal event is composed of a temporal predicate and a trigger part. Optionally induction links and concurrent events can be described. The type of temporal event (absloute, relative to an object, relative to an event or periodic) is mentioned by the keywork **periodic time**, **absolute time, time relative to object**, **time relative to event.**

AfterLoanExpiration **with relative time to event**
predicate « one month after loan.expirationdate»
trigger ...

4.3.8 Database administration policy

Temporal databases aim at keeping all data about the real world which represent present, past or future knowledge. Such histories increase the volume of the database and also decrease its performance in terms of queries. These two aspects are essential for preserving the optimal usage of the database. The analyst has to model the application but has also to take into account the policy to apply for cleaning the database by physically deleting all past information about the real world which is no longer used. But, these vacuuming activities must keep the data consistent. For instance, an object or a state of an object at a given instant should not be deleted physically if an association refers to it.

The coherence of the temporal database is difficult to preserve if the policy is always local to a class. We advise the analyst to consider the administration of the temporal data as a global problem, and to design the set of vacuuming activities as a particular case of the database behavior.

The vacuuming rules are described through a dynamic schema in terms of events and operations on objects. Thanks to the event concept vacuuming rules can be classified as follows :

	how to specify it	example
Number maximum of states	internal event on history	The *salary* temporal class with history<VT> is limited to 10 states
Fixed interval	temporal event relative on object	The *salary* temporal class with history<VT> is limited to the interval *2 years*
Relative time limit	temporal event relative on object	The inferior limit of the *salary* temporal class with history<VT> is : « now » - 2 years (now-relative time)
Periodic triggering	temporal event with periodic time	Each year
Manual triggering	external event associated to the *database administrator* actor	Database transaction called « Salary vacuuming »

Table 13 : Kinds of vacuuming rules

- Examples of vacuuming rules

Assuming that the *salary* history should be limited to 2 years, the analyst can have 3 ways of modeling it :

- temporal event with relative time on the begin of the history (a)
- each two years (b)
- alerting automatically the database administrator who responds by triggering the vacuuming activity (c).

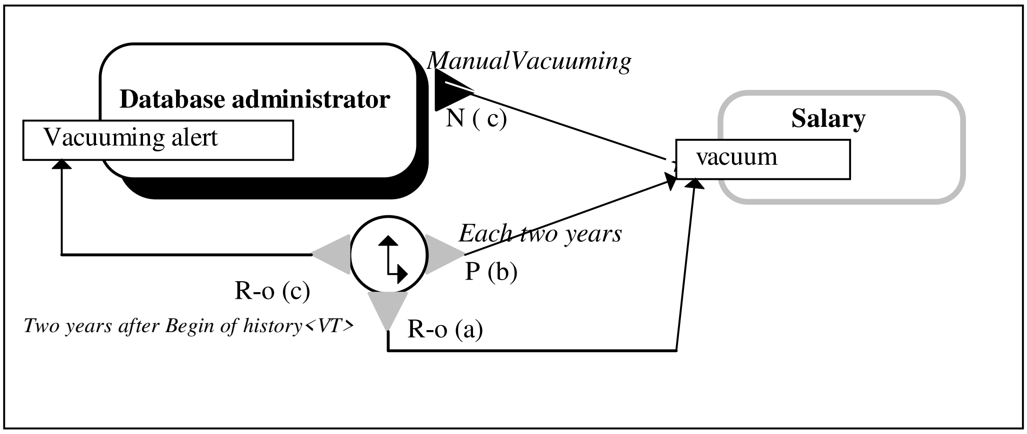


Figure 137 : Fixed interval of the *salary* history.

Assuming that the *salary* history should be limited to 10 states, the analyst can model it in two ways :

- an internal event on the history *« number of states in the history becomes greater than 10 »* (a) which automatically triggers the vacuuming activity
- Triggering an automatic alert to the database administrator with the internal event. The administrator responds by triggering the vacuuming activity(b).

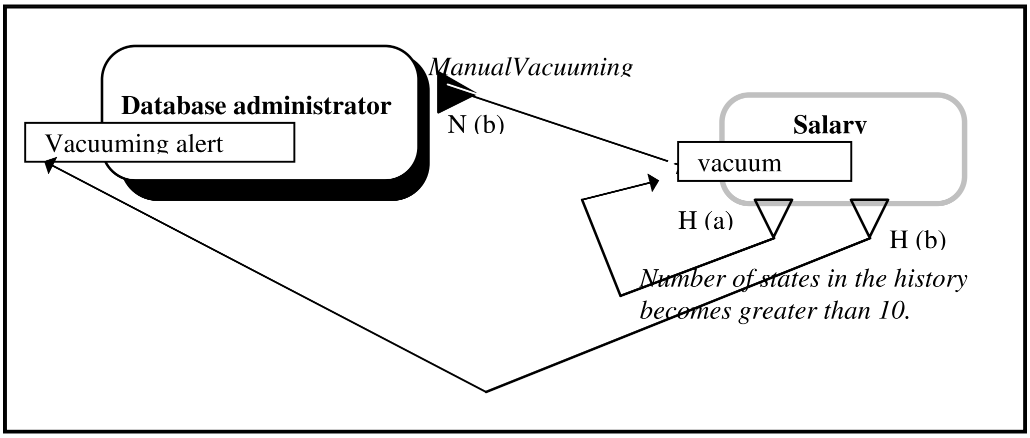


Figure 138 : Number maximum of states in the *salary* history.

- Consistency rules to follow

The vacuuming activities are defined by a set of database operations to execute in order to erase database objects in a consistent way. The trigger part of events is used to describe the vacuuming activity.
The analyst has to follow some guidelines in order to keep the database consistent.

R1 : The vacuuming operation should be performed on logically deleted objects,

R2 : A snapshot object can be physically deleted if :

- it has no association with a temporal object and
- it is not used in a temporal existence constraint.

R3 : The constraints have to be satisfied with the cleaned temporal database.

4.4 APPENDICES

4.4.1 Foundations of the time management

Time management is complex because time is complex. This appendix gives the general foundations of time modeling and time management in temporal databases. It comprises in four sections :

- **model of time** : basic knowledge about the temporal structure,
- **calendar & Granularities** : basic knowledge about time abstraction,
- **absolute, relative and periodic time** : different kinds of time expression,
- **time dimensions** : different semantics of time managed in temporal databases.

4.4.1.1 The model of time

• Temporal structure

In the temporal database community, one can find three time models which have been used as basis for all the researches on time : the continuous model, the dense model, and the discrete model.

• In the continuous model, the time is isomorphic to the real numbers: each real number is a point on the time line.

• In the dense model, the time is isomorphic to the rational numbers: each rational number is a point on the time line.

• In the discrete model, the time is isomorphic to the integer numbers: each integer number is a point on the time line.

Those three different models share a same hypothesis: the time line is an axis.

A temporal structure is a set of points with an asymmetric and transitive relation of precedence. The relation *less than* ('<') in the integer set **Z** is a temporal structure; the same holds for this relation in the rational set **Q**, and in the real set **R**. Then the time space is projected on a time line, with the relation of precedence using the axis direction.

A temporal structure is called **linear** if when one choose any two points in the temporal structure, then these two points satisfy the relation of precedence. Therefore the model of time is linear only if it supports one branch of time.

Z, Q, or **R** with '<' relation form a linear temporal structure (total order).

By opposition to the linear temporal structure, there exists branching or arborescent time. With branching time one may have several different pasts, called postlinear or right linear; several different futures, called antelinear or left linear. The '<' relation can only determine partial order, since it is only well defined over one branch of time.

In the temporal database field, the aim is not to understand the real world but to manage representation of it. Consequently, the linear structure of time is sufficient. However, if the database is used as a knowledge base integrated with a reasoning or/and planning systems, the arborescent time is required.

• Boundaries of Time

On a conceptual hand, the time is infinite, in the past as well as in the future. On a more practical hand, the time will be stored into some data structures, which are limited in size. Therefore the infinity of time will be impractical at the storage level.

Moreover, in the literature the *Big Bang*, 144 billion years ago, and the *Big Crunch*, sometime in the future, are known as the boundaries of Time.

Two special points enable to escape to these boundaries: **beginning** and **forever**. Those two points are located at the exterior of the boundaries: beginning will be smaller than any point, and forever will be greater than any point.

• Instant, Period and Interval

An isolated point on the time line is called an ***instant***. The time between two instants is called a ***period***. A period is precisely and only described by a couple of instants: its beginning boundary, and its ending boundary. All the instants contained between these two boundaries belong to the period. An instant can be seen as a period of zero duration, where its beginning and ending boundaries are equal.

An unanchored quantity of time is called an ***interval***. The difference between an interval and a period is that a period is well-located on the time axis, whereas an interval is only a time-window that can be moved over the time axis. An interval is not defined by the quantity of time between two boundaries but only by the a quantity of time.

The abstract domain used to represent all the temporal domains is called "temporal element". Its principal characteristic is to express the time with a certain granularity belonging to a particular calendar.

The choice among these three pre-mentioned models, namely continuous, dense and discrete, is not so important when considering the temporal database ontology. The definitions of instants and periods are the same, on continuous, dense or discrete time arrow. On the other hand, since the time is modeled on a discrete computing device, the view of time has to be ***discrete***. So the user will manipulate a discrete view of the time arrow which can be continuous, dense or discrete.

• A discrete view of time

Because of its boundaries, The time line is a time segment. Let us call T this time segment. We can divide T into a finite number of smaller segments called ***granules***. Each set of granules is well ordered, since each granule is associated to a segment of T and the ordered relation on T implies an ordered relation on the set of granules. This partitioning is called **scaling**. It gives a discrete view of the time segment. This view is dynamic since it only appears at the execution time of the requests. For instance, a day-sized scaling partitions the time segment into day-sized granules.

The smallest scale is composed of chronons. The ***chronon*** is the smallest granule. The size of the chronon is specific to the implementation.

When he chooses an appropriate scaling, the user can see the time segment as a discrete set of seconds, or of days, or of years ...

The discrete view of the time is created at the execution time of requests. But, if you look at an instant with different granularities, the view you will get of the information can be different. For instance, the instant « 1970-01-01 » seen through a day-sized scaling, becomes the period « 1970-01-01 00:00:00 , 1970-01-01 23:59:59 » through a second-sized scaling.

Let us introduce two terms : **temporal domains** and **temporal universes**.

A temporal universe, TU, is a finite succession of temporal domains, $<T_0, T_1, ... T_n>$. For two temporal domains, T_i and T_j, where i,j are disjoints. Each temporal domain T_i is a partitioning in segments of the temporal domain T_{i+1}. Therefore the segments of the temporal domain T_0 are the chronons.

For instance <seconds, minutes, hours, days, months, years> is a temporal universe. In this universe it is impossible to insert the weeks, since we cannot represent a week which overlaps two months or two years.

The discrete view, manipulated by users, is a discrete linear temporal structure, as (Z,'<') is one.

• Modeling Instants

The smallest quantity of time that can be represented in an instant timestamp is a chronon. But a chronon is a segment of time, whereas an instant is an isolated point on the time segment.

In fact, since we did not choose an explicit model of time, we can assume a continuous model. Then the chronons are bigger than the instants, since the continuous model implies that there are an infinite number of instants into each chronon.

NB: If we explicitly choose the discrete model, the size of an instant has to be equal to the size of a chronon.

Then an instant timestamp stores the information that this event has occurred sometime during a particular chronon or a particular granule. We can assume that the instant timestamp represents any instant during the granule in which it is located. Then the exact instant modeled by an instant timestamp, is never well-known. At the best, only the granule where it is located, is known.

However, two timestamps expressed with the same granule, may model two different instants. For instance, two instants located in the same hour-sized granule, can represent two different instants far away by fifty nine minutes. The two instants are represented by the same instant timestamp, with the granularity of an hour.

There exist some other hypothesis about the localization of the instants regarding to chronons and granules. Here are the two most common of those hypotheses, but they are far from reality.

• We may assume that the instant timestamp represents the first instant of the granule where it is located. Unfortunately, when an instant is represented by « 20h30min », this only means that we do not know more than it is located sometime during the minute described by « 20h30min ». It does not mean precisely the first instant of the 30th minutes after 20h.

• We may also assume that the instant timestamp represents the whole chronon or the whole granule where it is located. But an instant has a zero duration, whereas a granule has a non-zero duration. For instance, the instant timestamp « 20h30min » does not mean that it begins at the 30th minute after 20h, and that it ends just before the 31st minute.

So, an instant is modeled by an instant timestamp, with an associated time granularity. The instant represented by this timestamp is an instant from a granule, and this granule is determined by the scaling.

For example, the instant timestamp « 1996-01-01 », with a day-sized granularity, represents one instant during the first day of the January month of year 1996.

• Modeling Periods

The timestamp of a period owns an associated granularity. This timestamp is composed of two instant timestamps, and a constraint. This constraint forces the instant timestamp of the beginning boundary of the period, to precede or to be equal to the instant timestamp of the ending boundary. The two instant timestamps have the same granularity.

For instance, [1996-01-01, 1996-03-31) is a period timestamp, and represents a period from the first day of January of 1996 to the last day of March of 1996.

Let us have a look at another example: the period timestamp [1996-01-01 , 1996-01-01] represents a period somewhere during the day of January 1st 1996. But with this particular example, we cannot be more precise about the instant where it starts and where it ends, because the two instant timestamps are equal at the day granularity.

4.4.1.2 Calendars and Granularities

- Calendars

The basic international unit of time is the second. A lot of temporal units have been defined in order to rythm and measure temporal events. Then calendars have been introduced. A calendar is a human abstraction of the physical time line. A calendar is well-defined by its name, its origin, a collection of granules, and a corresponding system between the different units.
The default calendar is the Gregorian one. Its default granularities are year, month, day, hour, minute, and second.
There exist two ways of defining a calendar:

- the descending method, where one define a smaller unit thanks to the division of the basic unit. For instance, one minute is the division of one hour in sixty segments.

- the ascending method, where a greater unit is defined by the aggregation of several smaller units. For instance, one day is the aggregation of twenty four hours. This is the way temporal domains and temporal universes are built.

We follow the second method to define granules in a calendar. The smallest granule of each calendar has to be expressed with regards to the chronon which is implementation specific.

- **Conversion to finer or coarser granularities**

Each calendar has to provide the way to move from one of its granularities to another, i.e. a quantity of time expressed with a particular granule could be converted to a smaller, finer, or greater, coarser, granularity.
On one hand, converting an instant to a finer granularity is indeterministic since that transforms the instant into an indeterminate timestamp where we do not know precisely when the instant occurred. For instance, « January 1st 1996 » which represents the first day of 1996 at the day-sized granularity, is converted into the « 1996-01-01 00h ~ 1996-01-01 23h » indeterminate instant at the hour-sized granularity.
On the other hand, converting an instant to a coarser granularity results in a discard of precision about when the instant occurred. For instance, « January 1st 1996 » which represents the first day of 1996 at the day-sized granularity, is converted into the « January 1996 » instant at the month-sized granularity.
In TSQL2, two operations are provided to convert timestamps into finer or coarser granularity timestamps. First, *scale* allows to move a timestamp up or down the granularity system. Moving a timestamp up for the granules system produces a determinate timestamp, but moving a timestamp down for the granules system produces an indeterminate timestamp. Second, *cast* is similar to scale except that it only delivers determinate timestamps. When converting a timestamp into a finer granularity one, cast returns the first possible indeterminate timestamp that scale would return.
Examples of scale & cast operations from TSQL2 :

SCALE(DATE '06-01-1994' AS YEAR) = '1994'

SCALE(DATE '06-01-1994' AS MONTH) = ‘1994-06'
SCALE(DATE '06-01-1994' AS DAY) = ‘1994-06-01’
SCALE(DATE '06-01-1994' AS HOUR) = ‘1994-06-01 00 ~ 1994-06-01 23'
SCALE(DATE '06-01-1994' AS MINUTE) = ‘1994-06-01 00:00 ~ 1994-06-01 23:59'
SCALE(DATE '06-01-1994' AS SECOND) = ‘1994-06-01 00:00:00 ~1994-06-01 23:59:59'
CAST(DATE '06-01-1994' AS YEAR) = '1994'
CAST(DATE '06-01-1994' AS MONTH) =‘1994-06'
CAST(DATE '06-01-1994' AS DAY) =‘1994-06-01’
CAST(DATE '06-01-1994' AS HOUR) =‘1994-06-01 00’
CAST(DATE '06-01-1994' AS MINUTE) = ‘1994-06-01 00:00'
CAST(DATE '06-01-1994' AS SECOND) = ‘1994-06-01 00:00:00'

An indeterminate instant is described by a lower support, an upper support, and a probability mass function (p.m.f.). The manipulation of such timestamps is described in a special chapter of TSQL2 as a possible extension. About SQL/Temporal, the actual drafts do not deal with temporal indeterminacy. And in the road map of additions to SQL/Temporal, temporal indeterminacy appear as described in TSQL2.
In the Temporal Model (TM), we will not support temporal indeterminacy. TM adopts the semantic of the cast operation: for the conversion of a timestamp into a finer granularity one, the result will be the first possibility of the theoretical indeterminate timestamp.

- Gregorian Calendar

The most familiar and world-wide used calendar is the *Gregorian* one. The *Gregorian* calendar is based on the rotation of the earth on its axis and its revolution around the sun.
The origin used in SQL-92 is 'January 1st 1 AD, midnight'.
The default granularities are second, minute, hour, day, month, year.
The relationships between the granularities are classified in *regular* and *irregular mappings*.
• Regular mappings
- one Gregorian year includes 12 Gregorian months.
- one Gregorian day includes 24 Gregorian hours.
- one Gregorian hour includes 60 Gregorian minutes.
• Irregular mappings
- one Gregorian month includes 28 to 31 Gregorian days, depending on the month name and the month is located in a leap year or not (c.f. Table 14)
- one Gregorian minute includes 58 to 62 Gregorian seconds, depending on weather -2, -1, 0, 1, 2 leap seconds have been added.

Month	Duration in days
January	31
February	• 29, if the year is a leap year • 28, else
March	31

April	30
May	31
June	30
July	31
August	31
September	30
October	31
November	30
December	31

Table 14 : The Gregorian months

Leap seconds are due to the variation in the earth's rotation: the earth runs slow at roughly two milliseconds per day. Then after about five hundred days, the accumulation reaches near one second. Leap seconds are added just before midnight on dates selected in advance by the International Earth Rotation Service (IERS), in order to keep the Co-ordinated Universal Time within one second of the Greenwich Mean Time. For instance, the latest leap second occurred at the end of 1995 year: 1 leap second has been added on December 31st 1995 at 23h5min59sec, enabling the earth to catch up with its delay. Then the last minute of the 1995 year contained 61 seconds.

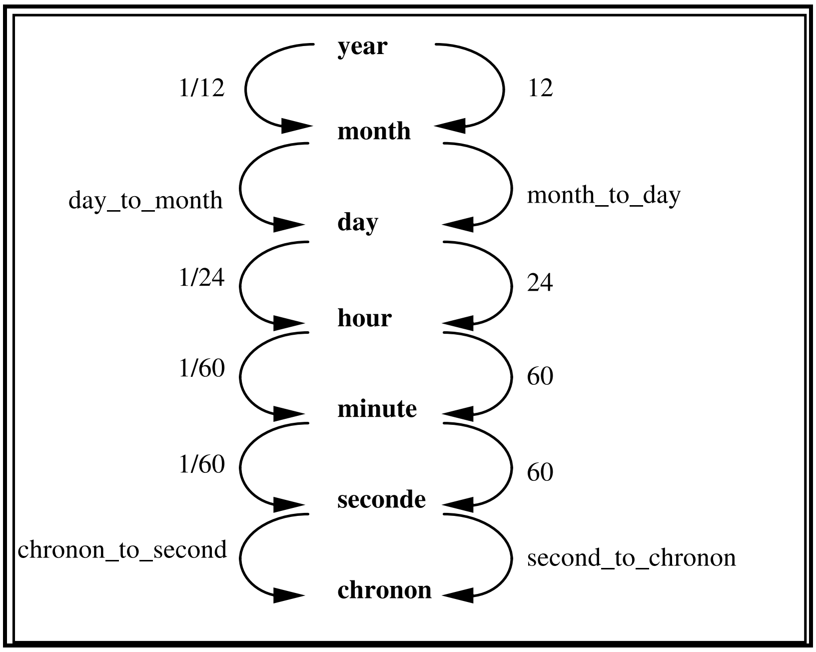


Figure 139 : The Gregorian conversion system

With the mapping between the different granules, we can define the system of Gregorian granules (c.f. Figure 139). Irregular mappings are represented thanks to functions with a name inferred from the source and the target granularities. The mappings dealing with chronons are implementation specific.

• Fixed and variable intervals

Another point to explore with calendars is the use of spans or intervals. A fixed interval represents a fixed duration. On the other hand, a variable interval is a span whose duration varies in relation to an associated event. As for instance, within the Gregorian calendar, month-sized intervals can be variable.

The fixed span for one month is thirty one days.

But fixed size are not always suitable for the month because of the irregular mapping between months and days.

Let us assume that we get an instant, defined at a day-sized granularity or smaller one. Then we move three semantics for the result of the addition of one month-long interval to this instant:

- the same day in the next month
  - For instance: <1996-05-03> + <1> month = <1996-06-03>
  - But what about: <1996-05-31> + <1> month = ?
- the same number of days from the begin of the month
  - For instance: <1996-05-30> + <1> month = <1996-06-30>
  - But what about: <1996-05-31> + <1> month = ?
- the same number of days from the end of the month
  - For instance: <1996-05-31> + <1> month = <1996-06-30>
  - But what about: <1996-05-01> + <1> month = ?

We will follow the characteristics of the MultiCal project, which is mentioned in TSQL2 as a concrete realization of supporting calendars. The solution is a combination of the two last semantics. If the instant is located in the first half of a month then the operation will be performed using the number of days from the beginning of the month. Conversely, if the instant is located in the second half of a month, then the operation will be performed using the number of days from the end of the month. The middle of the month is chosen at midnight of the sixteenth day, except for February where the middle is at midnight of the fifteenth day.

Then the previous unresolved operations have now the following results:

- <1996-05-31> + <1> month = <1996-06-30>
- <1996-05-01> + <1> month = <1996-06-01>

Regarding fractional month span addition, the operation is performed in two steps. First the addition of the integer part is computed, following the previous rules. Second, the fractional part is calculated in the result month. For instance, '1996-05-10' + '1,5' month, gives as result '1996-06-10' + '0.5' month, and finally '1996-06-25' since June contains 30 days and 30*0.5=15. If we get more days than there are in the destination month, we convert the excess to a fraction of the first month and apply this fraction to the next month. For instance, '1996-06-21' + '0.5' month, gives as result '1996-07-01' + '1/6' month since (30*0.5=15, 15-10=5 and 5/30=1/6), and finally '1996-07-06 04h04min48sec'.

Leap seconds are used in variable interval computations only after the destination month and day are determined. The final time is computed by multiplying the percentage of day left by the number of seconds in that day, including any leap seconds. With this policy the addition of an integer number of days within a month will result in the same time as the starting day.

- **Timestamp operations**

Associating a granularity to the time segment creates a discrete view of this time segment. However the time segment could be continuous, dense or discrete.

The timestamp operations are defined with respect to this discrete view of the time segment. For example, a comparison operation carried out at the day-sized granularity will compare some days; a duration calculus operation carried out at the second-sized granularity will return a duration in seconds.

As we saw before, an instant-sized granularity can not exist. The smallest granularity is composed of chronons, which are segments of time. Therefore the timestamp operations can not be carried out at the level of the instants. For instance, we could not answer to a question such as « Does an instant precede another instant, in an only instant point of view? ». It is impossible to reply precisely since an instant timestamp does not store the precise location of the event, but only the granule where is it located. Of course, we could answer precisely to the question « Does an instant precede another instant, in an instant timestamp point of view? », since here we compare two instant timestamps.

• Granularities in operations

Inside a temporal domain, the timestamp operations are carried out at the granularity of this domain. On the other hand, in order to extend these operations to the whole temporal universe, i.e. to operate with timestamps from different temporal domains, it is necessary to be able to manipulate one of the two timestamps in the granularity of the other one. This is possible thanks to the lattice of the calendar granules in which are expressed the timestamps. Then we have to consider three possible semantics, about the choice of the granularity at which the operation will be carried out.

• The operation can be performed at the granularity of the first or left operand. That is be done in SQL-92. The disadvantage is that symmetric operations are no longer symmetric.

• The finer operand granularity can be chosen, i.e. the coarser granularity timestamp will be plunged into the finer one. The problem is that converting a timestamp to a smaller granularity results in an indeterminacy, as described before in this chapter.

• The coarser operand granularity can be chosen, i.e. the finer granularity timestamp will be plunged into the coarser one. This solution avoids indeterminacy, but some information of the finer granularity are discarded.

TSQL2 adopted the semantic that the operation are performed at the granularity of the left operand, in order to remain consistent with SQL-92.

SQL/Temporal should require the granularities to be identical. This is a simplest way but also a more compelling one.

As default semantic, TM will adopt the policy of TSQL2 which is more flexible with regards to the user. So operations will be performed at the granularity of the left operand. Of course, the user can overload this behavior by casting the operands to his wanted Granularities.

4.4.1.3 ***Absolute, Relative and Periodic Times***

The temporal expressions can be classified in three groups : absolute, relative and periodic times.

• Absolute time

Locating a fact in the time line with an anchored quantity of time on the time line is called *absolute dating*. Periods and Instants are expressed in a given granularity of a given calendar. For instance, the absolute instant *'1996-07-06 04 :04 :48'* is expressed at the *second* granule of the *Gregorian* calendar. In addition [1996-01-01, 1996-03-31) is an example of absolute period at the *day* granule of the *Gregorian* calendar.

• Relative time

Locating a fact in the time line with a reference to another temporal event is called *relative dating*.

A relative instant is related to a temporal mark, and delayed from it of a given quantity of time, either in the future or in the past. This quantity of time is in fact an interval. A basic temporal mark can be composed of an instant and of its name. Then we introduce the *Temporal Mark* type to represent a basic temporal mark. A relative instant will be linked to a temporal mark, with an interval as delay.

In the relative instant type, the instant value has to be computed from the value of the temporal mark. There is no restriction about the dating of the temporal mark. Then the temporal mark can contain a relative instant, and the value computation will be performed recursively. Two pre-defined relative instants are ***tomorrow*** and ***yesterday***, and they are related to the ***now*** temporal mark respectively with a delay of one positive day and a delay of one negative day.

Late binding expression

The model of time will introduce '***now'*** as a late binding expression. The timestamp instant will only be valued when it is required, and will be the current instant.

For instance, the following valid time period [**now,'now'**) will represent a period from the current date until the request time. The *now* expression is evaluated and valued when the period is constructed whereas the *'now'* expression is evaluated when the value of the period is requested. This is used by the system to update the transaction timestamp.

• Periodic time

The periodic time is required to manage periodic activities of database applications. A periodic time is formally defined as an infinite serie of instants or periods (time series) where the distance between two elements is defined as an interval. For instance, *"the 25th of each month"* is a periodic time expression. The management of the periodic time is required to trigger treatment when an element of this time serie occurs. This requirement allows us to restrict a periodic time to its last element (base) and its distance between two elements (distance). Considering a current date equal to *"1996-03-12"* and the previous example, its distance is *a month* and the base is the instant *"1996-02-25"*.

In this temporal extension, a periodic time can be limited by a period. For instance, *"each day from the order date to the delivery date"* is a valid periodic time expression.

4.4.1.4 Time Dimensions

The literature on temporal databases identifies three time dimensions :

-*valid time* of a fact is the time when it is true in the modeled reality. For instance, Dupond has been employed by Expert Company from 1995-02-01 to 1996-12-10. This definition allows to represent either the instant or the period when the fact is valid. The period of validity here is [1995-02-01, 1996-12-10).

-*transaction time* of a database fact is the time when the fact is current in the database and may be retrieved. For instance, the employment of Mr. Dupond by the Expert company is available in the database from 1995-01-10 14:56:50 to 1996-10-15 10:50:10. The transaction time of this fact is the period [1995-01-10 14:56:50 , 1996-10-15 10:50:10).

-*user-defined time* is an uninterpreted attribute domain of time. It is useful for semantic of time different of the valid and the transaction times. For instance, the birthday of a person is user-defined time. It is materialized in the temporal database by an attribute with a temporal domain which is not interpreted by the temporal DBMS but by the application layer.

According to these types of time, an Object-Oriented database extended to time requires the management of :

snapshot objects : the last state of an object is present in the database,

historical objects : all the valid states object during its life span in the application are kept in the database. Each of these states are time stamped with the valid time, i.e. time during which this state is/was/will be true in the application domain. The historical object is restricted to state changes valid according to the semantic of the application (correction of mistake is not considered as a valid state change), *rollback objects* : all the states taken by an object during its life span are kept in the database. Each of these states are time stamped with the transaction time, i.e. time during which this state is/was/will be true in the database,

temporal objects : all the states of an object during its life span are kept in the database. Each of these states are time stamped with the valid and the transaction time. Correction of the valid time is considered as a state change of an object.

We present the properties of each kind of objects.

• **Snapshot objects** are managed by conventional modeling of the real world as a snapshot relative to the current state of the real world. Database changes are due to insertion or update or deletion operations. Update or deletion operations imply the lost of previous state of data which are the targets of the operations. Managing explicitly temporal information is a decision of the application designer.

• **historical objects** allows to keep track of the history of data**.** But, the semantic attached to the history is different from one of rollback objects. The timestamp managed in the historical objects is the valid time instead of the transaction time. This divergence between these two kinds of object implies the following characteristics :

- Mistakes detected in an historical object can be corrected. These corrections are not kept in the history. It means that the « erroneous state » is replaced by the « corrective state ». An historical object behaves as a snapshot object when a correction operation is requested. When the operation represents an evolution of the object, it behaves as a rollback object.
- Past, present and future states of an historical object can be updated. It means that the knowledge about past states can be improved at any time. An historical object aims at providing the past states of the real world as they are known at the current time (the « erroneous states are lost).

employee	**name**	**function**	**valid time**	
			begin	**end**
(a)	Durand	programmer	94-12-12	95-12-20
(b)	Durand	project leader	95-12-20	forever
(c) et (d)	Martin	engineer	95-02-01	forever
(e)	Dupond	consultant	95-01-02	96-10-12

Figure 140 : Examples of historical objects.

• **Rollback objects** allow to preserve successive states of an object. Each of this state is stamped with the transaction time. An operation (insertion, modification and deletion) must modify the last state of the database, and generate a new state. Last states cannot be modified and future states are forbidden. Data are no more updated in such objects.
In addition data become valid when they are stored in the database. Rollback objects keep the history of their updates rather than the history of their evolution in the real world.

employee	name	function	transaction time	
			begin	end
(a)	Durand	programmer	95-01-01 9:20:15	96-01-01 15:56:50
(b)	Durand	project leader	96-01-01 15:56:50	*'now'*
(c)	Martin	analyst	95-01-01 9:20:15	95-01-10 14:56:50
(d)	Martin	engineer	95-01-10 14:56:50	*'now'*
(e)	Dupond	consultant	95-01-10 14:56:50	96-10-15 10:50:10

Figure 141 : Examples of rollback objects.

Figure 141 shows the management of a rollback object. The state of objects result from the following operation :

-the object *Durand* (a) et object *Martin* (c) are created at *95-01-01 9:20:15*,

-the object *Dupond* (e)is created at *95-01-10 14:56:50*,

-at *95-01-10 14:56:50*, the object *Martin* (c) is corrected because of a mistake, his function is not *analyst* but *engineer*. Consequently, the state (c) exists *until 95-01-10 14:56:50* and the state (d) is created with the following *timestamp [95-01-10 14:56:50, ‘now’).*

-the object *Durand* is modified at *96-01-01 15:56:50* : his function evolves to *project leader*. This logical update implies :

-the update of the timestamp associated to the state (a) (the end of the transaction time is set to *96-01-01 15:56:50*), and

-the creation of the state (b) with the following transaction time [*96-01-01 15:56:50*, *'now')* ;

-and finally, at *96-10-15 10:50:10* the object *Dupond* (e) is logically deleted. This deletion induces the update of the transaction time (end of the transaction time is set *96-10-15 10:50:10*.

Remarks : it is impossible in rollback objects to differentiate an update due to an evolution of the real world or a correction of a mistake. In order to manage this distinction, we propose two operations : evolution and correction. A mark indicating if the state is due to an evolution or a correction is managed.

The objects presented in Figure 140 result from the same operations of Figure 141. Differences between these two kinds of objects come from the semantical difference of the history.

The operations leading to the objects of Figure 140 are the following :

-the objects *Durand* (a) and *Martin* (c), created at *95-01-01 9:20:15* (cf. Figure 140), are not valid in the real world at the same time. The object *Durand* is *programmer* in the real world since 94-12-12 (begin of its validity) whereas the object *Martin* will be *analyst* from 95-02-01,

-the objet *Martin* is initially created at *95-01-01 9:20:15* (cf. Figure 140) with the following instant 95-02-01 as valid time. But his function is not correct. The corrective operation is executed at *95-01-10 14:56:50*. This leads to replace the « wrong value » by the « correct value » ; the history does not keep track of this modification,

-at *96-01-01 15:56:50*, the objet *Durand* change his function, this change is valid from 95-12-20 the valid time of the state (a) becomes 95-12-20 and the state (b) is created with the following valid time [95-12-20, *forever*).

-at *96-10-15 10:50:10*, the object *Dupond* is logically deleted with the following valid time 96-10-12.

• **Temporal objects** combine the advantages of rollback and historical objects. These objects can be target of rollback requests (concerning the state of the object at a given instant) as well as historical requests (concerning the valid state of the real world at a given time).

They manage a timestamp composed of the valid and the transaction times. They permit to represent the valid states of the real world at a given instant as they are known by the system at another given instant.

Consequently, each operation (insertion, modification and suppression) leads to insert a new state in the history (append-only).

Temporal objects allow us to take into account retroactive and postactive operations. The notions of retroaction (delayed) and postaction (in advance) come from the delay between the valid and the transaction time. It means that the application has to observe the difference between the time occurrence of a fact in the real world and its effective time in the database.

	name	**function**	**valid time**		**transaction time**	
			begin	**end**	**begin**	**end**
(a)	Durand	programmer	94-12-12	*forever*	95-01-01 9:20:15	96-01-01 15:56:50
(b)	Durand	programmer	94-12-12	95-12-20	96-01-01 15:56:50	*'now'*
(c)	Durand	project leader	95-12-20	*forever*	96-01-01 15:56:50	*'now'*
(d)	Martin	analyst	95-02-01	*forever*	95-01-01 9:20:15	95-01-10 14:56:50
(e)	Martin	engineer	95-02-01	*forever*	95-01-10 14:56:50	*'now'*
(f)	Dupond	consultant	95-01-02	*forever*	95-01-10 14:56:50	96-10-15 10:50:10
(g)	Dupond	consultant	95-01-02	96-10-12	96-10-15 10:50:10	*'now'*

Figure 142 : Examples of temporal objects.

The operations applied to obtain the objects of Figure 142 are the same operations presented in Figure 141 and Figure 140.

In the case of temporal objects, each operation generates a new state. The creation of the object *Durand* is retroactive because its valid time *(94-12-12)* begins before the begin of its transaction time (*95-01-01 9:20:15*). *Durand* became project leader with a valid time 95-12-20 but this fact is known in the database since *96-01-01 15:56:50*. This evolution is also retroactive and leads to :

- modify the transaction time of the state (a),
- insert a new state (b) because of the valid time update,
- insert a new state (c) with the new function value and the new valid time.

The state (a) is no longer updatable, it represents the past knowledge we have in the database. During the following period [*95-01-01 9:20:15* , *96-01-01 15:56:50*) the database has the following information : *Durand was programmer from 94-12-12 until forever*.

This kind of objects is the semantically more powerful than the others, but they require a lot of space as you can imagine.

The classification of objects dependent to time are depicted by Figure 143:

	transaction time not managed	transaction time managed
valid time not managed	**Snapshot objects**	**Rollback objects**
valid time managed	**historical objects**	**Temporal objects**

Figure 143 : Classification of objects.

Notice that user-defined times are not used in this classification. User-defined times have no signification in the DBMS and are managed as ordinary attributes. In fact, User-defined time is offered by all DBMS.

4.4.2 The syntax of the TOOM specification language

<u>Lexical conventions</u> :

<u><comment></u> :

- comment which can appear anywhere :

/* <text> ***/**

- comment which can appear only between atomics OOPL :

// <text> **end-of-line**

<<u>identifier</u>> : an identifier is an arbitrary long sequence of letters or digits. The first character must be a letter; the underscore is counted as a letter. Upper and lower case letters are different. All characters are significant. The identifier with a double underscore is reserved for use by implementation and standard libraries should be avoided by users.

<<u>keywords</u>> :

single character : **^ * ( ) - + = { } [ ] / ; ' : " < > , . **

character combinations : **<= >= := <> ::**

<<u>literals</u>> :

<char-constant> : a character constant is one or more characters enclosed in single quotes : 'a',

<string-constant> : a string constant is a sequence of characters surrounded by double quotes : "assss",

<integer-constant> : a sequence of digits which is in base ten : 100

<real-constant> : consists of an integer part a decimal point and a fractional part

4.4.2.1 Application definition

<application-definition>::=

application <application-name> <specification-part>

<specification-part>::=

definition <specification-list> **enddefinition**

<specification-list> ::=

<object-class-definition-list>

<actor-class-definition-list>

[<calendar-class-definition-list>]

[<domain-definition-list>]

[<triggering-condition-definition-list>]

[<triggering-factor-definition-list>]

<domain-definition-list> ::=

<domain>

| <domain-definition-list> <domain>

<object-class-definition-list> ::=

<object-class-definition>

| <object-class-definition-list> <object-class-definition>

<actor-class-definition-list> ::=

<actor-class-definition>

| <actor-class-definition-list> <actor-class-definition>

<calendar-class-definition-list> ::=

<calendar-class-definition>

| <calendar-class-definition-list> <calendar-class-definition>

<triggering-condition-definition-list> ::=

<condition-definition>

| <triggering-condition-definition-list> <condition-definition>

<condition-definition> ::=

triggering-condition

<boolean-fonction-signature>

comment <text>

endtriggering-condition

<triggering-factor-definition-list> ::=

<factor-definition>

| <triggering-factor-definition-list> <factor-definition>

<factor-definition> ::=

triggering-factor

<factor-signature>

comment <text>

endtriggering-factor

4.4.2.2 Domain definition

<domain>::=

domain <domain-name> '**:**' <domain-definition>

[**operations** <operation-list>] **enddomain**

<domain-name>::=

<identifier>

<domain-definition>::=

<enumerated-domain-type>

| <domain-elaborated-type-specifier>

<domain-elaborated-type-specifier> ::=

<domain-final-type>

| <constructor> '(' <domain-elaborated-type-specifier> ')'

| '**[**' <domain-field-specifier-list> '**]**'

<domain-final-type> ::=

<simple-type-name>

| <domain-name>

| <temporal-domain>

temporal-domain::=

INSTANT '**<**' <calendar-name> '**,**' <granule-name> '**>**'

| **INSTANT-A** '**<**' <calendar-name> '**,**' <granule-name> '**>**'

| **INSTANT-R** '**<**' <calendar-name> '**,**' <granule-name> '**>**'

| **PERIOD** '**<**' <calendar-name> '**,**' <granule-name> '**>**'

| **PERIOD-A** '**<**' <calendar-name> '**,**' <granule-name> '**>**'

| **PERIOD-R** '**<**' <calendar-name> '**,**' <granule-name> '**>**'

| **INTERVAL** '**<**' <calendar-name> '**,**' <granule-name> '**>**'

<granule-name>::=

<identifier>

<calendar-name>::=

<identifier>

<simple-type-name> ::=

<constructor> ::=

set | bag | list | multilist

<domain-field-specifier-list> ::=

<domain-field-specifier>

| <domain-field-specifier-list> '**,**' <domain-field-specifier>

<domain-field-specifier> ::=

<field-name> **':'** <domain-definition>

<enumerated-domain-type>::=

'**{**' <string-constant-list> '**}**'

| '**[**' <integer-constant> '**..**' <integer-constant> '**]**'

| '**[**' <real-constant> '**..**' <real-constant> '**]**'

| '**[**' <char-constant> '**..**' <char-constant> '**]**'

| '**[**' <date-constant> '**..**' <date-constant> '**]**'

<string-constant-list>::=

<string-constant>

| <string-constant-list> '**,**' <string-constant>

4.4.2.3 Object class definition

<object-class-definition> ::=

<snapshot-basic-class-definition>

| <snapshot-derived-class-definition>

| <temporal-basic-class-definition >

| <temporal-derived-class-definition >

| <temporal-dictionary- basic-class-definition >

| <temporal-dictionary- derived-class-definition >

<snapshot-basic-class-definition> ::=

basic-class <snapshot-class-name> **':'** <inherited-class-list>

extents <persistent-collection-list>

[**clusters** <cluster-list>]

[**properties** <object-properties>]

[**constraints** <invariant-list>]

[**operations** <operation-list>]

[**queries** <operator-list>]

[**events** <internal-event-list>]

endbasic-class

<snapshot-derived-class-definition> ::=

derived-class <snapshot-class-name> **':'** <inherited-class-list>

extents <persistent-collection-list>

[**properties** <object-properties>]

[**derivation function** <operator-list>]

[**queries** <operator-list>]

[**events** <internal-event-list>]

endderived-class

<temporal-basic-class-definition> ::=

temporal-class for basic class <time-class-name> **of** <snapshot-class-name>

extents <persistent-collection-list>

time dimensions <time-dimension-definition>

[**properties** <object-properties>]

[**constraints** <invariant-list>]

[**operations** <operation-list>]

[**queries** <operator-list>]

[**events** <internal-event-list>]

endtemporal-class

<time-dimension-definition> : :=

VT <temporal-domain> **as** <valid-time-definition>

| **TT** <nature>

| **VT** <temporal-domain> **as** <valid-time-definition>, **TT** <transaction-time-definition>

<valid-time-definition>::=

state

| **history <typeVT-history>**

<typeVT-history> ::=**complete** | **partial** [**with extrapolation**]

<transaction-time-definition>::=

state | **history**

<temporal-derived-class-definition> ::=

temporal-class for derived class <time-class-name> **of** <snapshot-class-name>

extents <persistent-collection-list>

time dimensions <time-dimension-definition>

[**properties** <object-properties>]

[**derivation function** <operator-list>]

[**queries** <operator-list>]

[**events** <internal-event-list>]

endtemporal-class

<temporal-dictionary-basic-class-definition> ::=

temporal dictionary for basic class <time-class-name> **of** <snapshot-class-name>

extents <persistent-collection-list>

key VT PERIOD '**<**' <calendar-name> '**,**' <granule-name> '**>**'

[**time dimension TT** <transaction-time-definition>]

[**properties** <object-properties>]

[**constraints** <invariant-list>]

[**operations** <operation-list>]

[**queries** <operator-list>]

[**events** <internal-event-list>]

endtemporaldictionary-class

<temporal-dictionary-derived-class-definition> ::=

temporal dictionary for derived class <time-class-name> **of** <snapshot-class-name>

extents <persistent-collection-list>

key VT PERIOD '**<**' <calendar-name> '**,**' <granule-name> '**>**'

[**time dimension TT** <transaction-time-definition>]

[**properties** <object-properties>]

[**derivation function** <operator-list>]

[**queries** <operator-list>]

[**events** <internal-event-list>]

endtemporaldictionary-class

<persistent-collection-list>::=

<persistent-collection-name>

| <persistent-collection-list> '**,**' <persistent-collection-name>

<inherited-class-list>::=

<object-class-name>

| <inherited-class-list> '**,**' <object-class-name>

<object-class-name> : :=

<snapshot-class-name>

| <time-class-name>

<cluster-list>::=

<cluster>

| <cluster-list> <cluster>

<cluster>::=

<cluster-name> '(' <object-class-list> ')'

<cluster-name> ::=

<identifier>

<object-class-list>::=

<object-class-name>

| <object-class-list> '**,**' <object-class-name>

<invariant-list>::=

<invariant> <type1> [‘**,**’ <type2>]

|<invariant-list> <invariant>

<type1> : := **intra-object | inter-object**

<type2> : := **intra-VT | inter-VT | intra-TT | inter-TT | intra-Bi | inter-Bi**

<invariant>::=

<boolean-function-signature>

| **unique** '(' <expression-list> ')'

| **partitioning** '(' <expression1-list> ')'

| **covering** '(' <expression1-list> ')'

| **disjoint** '(' <expression1-list> ')'

| **<**TOQL-statement**>** // temporal existence constraint

| **temporal_unique** '(' <type_scope> ‘**,**’ <expression-list> ')'

<type_scope> : : = **local | global**

<expression1-list>::=

<expression1>

| <expresion1-list> <expression1>

<expression1>::=

<expression>

| <cluster-name>

<operation-list>::=

<operation>

| <operation-list> <operation>

<operation> ::=

<signature-operation> [**':' aggr '('**<operation-reference-list> **')'**]

[**':'** <operation-reference-list>] [**comment** <text>]

<signature-operation> ::=

<function-signature>

| <procedure-signature>

<cardinality_agg> ::=

one | many

<operation-reference-list> ::=

<operation-spec>

| <operation-reference-list> **','** <operation-spec>

<operation-spec> ::=

<cardinality_agg> **','** <operation-reference>

<operation-reference> ::=

<operation-name>

| <object-class-name> **'::'** <operation-name>

<operator-list> ::=

<function-signature>

| <operator-list> <function-signature>

<internal-event-list> ::=

<internal-event>

| <internal-event-list> <internal-event>

4.4.2.3.1 Definition of Attributes

<object-properties>::=

<attribute-definition>

| <attributes> <attribute-definition>

<attribute-definition> ::=

<attribute-name> **':'** <attribute-type-specifier> ['=' <default_value>]

<default_value> ::= <constant>

<attribute-type-specifier> ::=

<attribute-final-type> [**derived**]

<attribute-final-type> ::=

<domain-final-type>

| **aggr** '(' [<cardinality> **','**] <object-class-name> ')'

| **asso** '(' [<cardinality> **','**] <object-class-name> [**',' inverse of** <attribute-name>] ')'

| **state asso** '(' [<cardinality> **','**] <temporal-class-name> [**',' inverse of** <attribute-name>] ')'

| **inverse** '(' [<cardinality> **','**] <object-class-name> **'::'** <attribute-name> ')'

<cardinality> ::=

one | zeroone | many | zeromany

4.4.2.3.2 Function and procedure signatures

<function-signature> ::=

<type-specifier> <function-name> '(' [<argument-list>] ')' [**type create**] **|**

<type-specifier> **$** <function-name> '(' [<argument-list>] ')'

<procedure-signature> ::=

<operation-name> '(' [<argument-list>] ')' [**type update | type correction | type evolution | type delete | type vacuum**]

<boolean-function-signature> ::=

boolean <boolean-function-name> '(' [<argument-list>] ')'

<factor-signature> ::=

<constructor> '(' <type-specifier> ')' <factor-name> '(' [<argument-list>] ')'

<function-name> ::=

<operation-name>

| <conversion-function-name>

| <operator-function-name>

<boolean-function-name> ::=

<invariant-name>

<condition-name>

<argument-list> ::=

<argument-declaration>

| <argument-declaration> '**,**' <argument-list>

<argument-declaration> ::=

[<variable-name> '**:**'] <type-specifier> ['**:=**' <expression>]

<operation-name> ::=

<identifier>

<factor-name> ::=

<identifier>

<invariant-name> ::=

<identifier>

<condition-name> ::=

<identifier>

4.4.2.4 Internal events and trigger part

<internal-event> ::=

<event-name> **on** <type-internal-event>

[**induction** <induction-list>]

[**concurrent events** <concurrent-list>]

predicate '(' <text> ')'

<trigger>

<type-internal-event>::= **object | history**

<event-name> ::=

<identifier>

<trigger> ::=

trigger <statement-list> **endtrigger**

<statement-list> ::=

<statement>

| <statement-list> <statement>

<statement> ::=

[**IF** <call-condition> **THEN**] [**FOREACH** <variable-name> **in** <call-factor>] <call-operation>

4.4.2.5 Actor class and Calendar class definitions

4.4.2.5.1 Actor class definition

<actor-class-definition> ::=

actor-class <actor-class-name>

properties <object-properties>

external operations <external-operation-list>

external-event <external-event-list>

endactor-class

<actor-class-name> ::=

<identifier>

<external-operation-list>

<external-operation>

| <external-operation-list> <external-operation>

<external-operation> ::=

<operation-name> '(' [<argument-list>] ')'

4.4.2.5.2 Calendar class definition

<calendar-class-definition> ::= <user-defined-calendar> | <Gregorian-calendar-overloading>

<user-defined-calendar>::=
calendar-class <calendar-class-name>
origin <string-constant>
basic unit '**<**' <granule-name>'**,**' <calendar-name> '**>**'
granules <granules-list>
[**external formats** <external-format-list>]
operations <operator-list>
[**temporal-events** <temporal-event-list>]
endcalendar-class

<Greorian-calendar-overloading>::=
calendar-class Gregorian
[**external formats** <external-format-list>]
[**temporal-events** <temporal-event-list>]
endcalendar-class

<calendar-class-name> ::=
<identifier>

<granules-list>::=<granule-definition>
| <granule-list> <granule-definition>

<granule-definition> ::= <granule-name> '**{**' <mapping-function> '**,**' <mapping-function> '**}**'

<mapping-function> ::= <irregular-mapping> | <regular-mapping>

<regular-mapping>::= <granule-name> '=' <float-constant> <granule-name>

<irregular-mapping>::= <function-name> '(' ')'

<external-formats-list>::=<external-format-definition>
| <external-format-list> <external-format-definition>

<external-format-definition>::=

<external-format-name> '**=**' <string-constant> '**{**' <mapping-signature> '**,** ' <mapping-signature> '**}**'

<calendar-class-name> ::=

<identifier>

<operator-list>::= <operator-signature>

| <operator-list> <operator-signature>

<operator-signature>::= <type-specifier> **operator+** '(' [<argument-list>] ')'

<mapping-signature> ::=

<type-specifier> <function-name> '(' [<argument-list>] ')'

4.4.2.5.3 Temporal event definition

<temporal-event-list> ::=

<temporal-event>

| <temporal-event-list> <temporal-event>

<temporal-event> ::=

<event-name> **with** <type-temporal-event>

[**induction** <induction-list>]

[**concurrent events** <concurrent-list>]

predicate '(' <text> ')'

<trigger>

<type-temporal-event>::=

periodic time | absolute time | time relative to object | time relative to event

4.4.2.5.4 External event definition

<external-event-list> ::=

<external-event>

| <external-event-list> <external-event>

<external-event> ::=

<event-name>

[<VT-message>] **message** <message-name>

attributes <message-attributes>

[**induction** <induction-list>]

[**concurrent events** <concurrent-list>]

predicate '(' <text> ')'

<trigger>

<VT-message>::= **in time | future | past**

<message-attributes>::=

<message-attribute-definition>

| <message-attributes> <message-attribute-definition>

<message-attribute-definition> ::=

<attribute-name> '**:**' <message-type-specifier>

<message-type-specifier> ::=

<message-final-type>

| <constructor> '(' <message-type-specifier> ')'

<message-final-type> :

<domain-final-type>

| **structure** '(' <OODL-type-name> ')' // O*-type-name : object class or domain

| '[' <message-field-specifier-list> ']'

<message-field-specifier-list> ::=

<message-field-specifier>

| <message-field-specifier-list> '**,**' <message-field-specifier>

<message-field-specifier> :

<field-name> '**:**' <message-type-specifier>

4.4.2.6 Call of conditions, factors and operations

<call-condition> ::=<condition-name> '**(**' [<expression-list>] '**)**'

<call-factor> ::=<factor-name> '**(**' [<expression-list>] '**)**'

<call-operation> ::= <procedure-expression> '**(**' [<expression-list>] '**)**'

<procedure-expression> ::=
<name2>
| <object-expression> '**.**' <name2>

<object-expression> ::=
<variable-name>
| <object-expression> '**.**' <name1>
| <object-expression> **inverse of** <name1> '**.**' <object-class-name>
| <procedure-expression> '**(**' [<expression-list>] '**)**'

<name1> ::= // name of attribute for an object class
<attribute-name>
| <qualified-class-name> '**::**' <name1>

<name2> ::= // name of an encapsulated function or a procedure
<operation-name>
| <operator-function-name>
| <conversion-function-name>
| <qualified-class-name> '**::**' <name2>

<expression-list> ::=
<expression>
| <expression-list> '**,**' <expression>

<expression> ::=
<primary-expression>
| <postfix-expression>

<postfix-expression> ::=
<final-expression>
| <direct-fonction-name> '**(**' [<expression-list>] '**)**'

| <postfix-expression> **'.'** <fonc-name> '**(**' [<expression-list>] '**)**'

| <postfix-expression> '**.**' <name>

| <postfix-expression> **inverse of** <name1>'**.**' <object-class-name>

<final-expression> ::=

<variable-name>

| <persistent-collection-name>

| context | old | new | self

<primary-expression> ::=

<literal>

| '('<expression> '**)**'

<literal> ::=

<integer-constant>

| <character-constant>

| <symbol-constant>

| <date-constant>

| <floating-constant>

| <string-constant>

| <boolean-constant>

| <reference-constant>

<boolean-constant> ::=

true | **false**

<reference-constant> ::=

nil

4.4.2.7 Event induction graph & Event concurrence graph

<induction-list>::=

<induction-link>

| <induction-list> <induction-link>

<concurrent-list>::=

<concurrent-link>

| <concurrent-list> <concurrent-link>

<induction-link>::=<modality> [**IF** <call-condition>] [**with delay** <quantity-of-time>] [<cardinality>] <call-event>

<modality> ::= **may** | **must**

<cardinality> ::= **one** | **more**

<concurrent-link>::= [**before** | **after**] <event-name>

<call-event>::= <proc-expression> **'('** [<expression-list>] **')'**

<proc-expression> ::=
<namef>
| <object-expression> **'.'** <namef>

<object-expression> ::=
<variable-name>
| **CLOCK**
| <object-expression> **'.'** <nameatt>
| <object-expression> **inverse of** <nameatt> **'.'** <object-class-name>
| <proc-expression> **'('** [<expression-list>] **')'**

<nameatt> ::= // name of attribute for an object class
<attribute-name>
| <qualified-class-name> **'::'** <nameatt>

<namef> ::= // name of an encapsulated function or a procedure
<event-name>
| <qualified-class-name> **'::'** <namef>

4.4.3 A Case Study :
THE LOAN MANAGEMENT SYSTEM

In this appendix, a case study for a bank loan application is conducted to demonstrate how to specify an IS as a representation of the real world bank loan system.

4.4.3.1 Management of the loan management system

The problem is stated as follows

- **Aims and objectives**

The aim of the project is to organize the functioning of a bank loan management stressing on an automated IS. The IS should permit a coherent management of the loan and inform the bank manager of the present or former states of any loan made by the bank bringing with it, for example, information on:

- the inventory of loans by date of entry;
- the inventory of loans at a given moment;
- the inventory of present or former loan owners , or for a given period;
- the list of fined loans;
- the loans that have been revised in the past two years;
- the average duration of loans, etc.

- **The Organisational Choices**

The loan management consists of a collection of loans, each with a unique reference. For each loan there exists a client of the bank. The loan has an amount, a duration and a rate. There's a theoretic monthly repayment computed at the creation of the loan but the loan can be revised following certain conditions. Fines can be attributed on loan owners for late payments. The loan is closed if the loan amount has been paid and if there is no fines left.

- **Management Rules**

Below is a list of the main rules for managing loans:

- A client can't have more than one loan at a given instant.
- A client who has a current loan is said "active". On the contrary, if he has no loan, he is said "inactive".
- The loan request is put on the waiting list only if the client that makes the demand has a regular monthly salary (managed by the bank). It is necessary also that the client had never been suspended. If the client doesn't satisfy these conditions, the loan request is refused.
- After one month, the manager' will send its agreement or refusal.

- There is a monthly repayment (on a day decided by the client : the due day).

- The repayment is computed at the creation of the loan with its amount, its duration and its rate.

- The repayment can be recomputed when the loan is revised.

- Each month, at the due day, the bank checks the repayments made by the client for his loan. The loan statement is sent to the client.

- The maximum delay allowed for the repayment is 3 months. After this delay, there's a fine applicable on the loan. This fine is computed proportionally to the loan amount.

- The client who has an unpaid tax is said "suspended".

- When the client makes a repayment, the fine is paid first. If the tax can't be totally paid, the repayment is taken into account for the loan.

- If the fine is not paid at the due day, it is increased.

- If the client has paid 3 monthly repayments in advance, there is a revision of the loan.

- The loan is closed when the loan amount is totally paid and when there is no due fines.

The various object classes identified for the loan management application schema are:

1. *LOAN* - a snapshot class of all loans kept in the bank.
2. *DEMAND* - a snapshot class of all demands of loans.
3. *CLIENT* - a snapshot class of all clients having loans.
4. *MONTHLY SALARY* - a temporal class of the monthly salary of the clients.
5. *CLIENT STATUS* - a temporal class of a the family status of the clients.
6. *LOAN REQUEST STATUS* - a temporal class of the status of the loan.
7. *REPAYMENT* - a temporal class of all the repayments made for the loans.
8. *FEATURES* - a temporal class of the characteristics of the loans.
9. *BILL BOOK* - a temporal class of the bill book of the loans.
10. *TAX* - a snapshot class grouping all taxes of the loans.
11. *TAXM* - a temporal class grouping all the taxes repayments.

4.4.3.2 Global structure graph

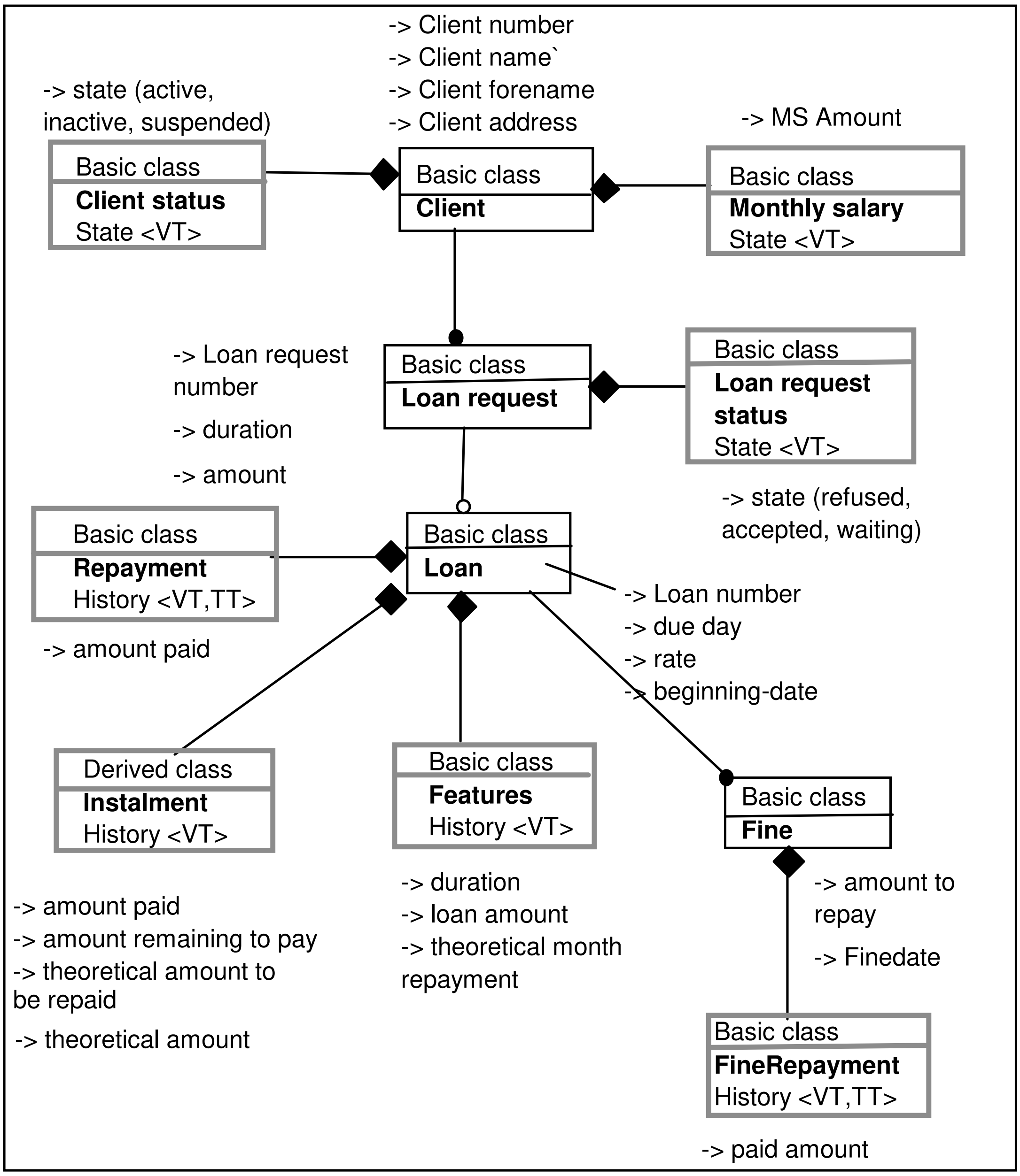


Figure 144: Global structure graph

Static view

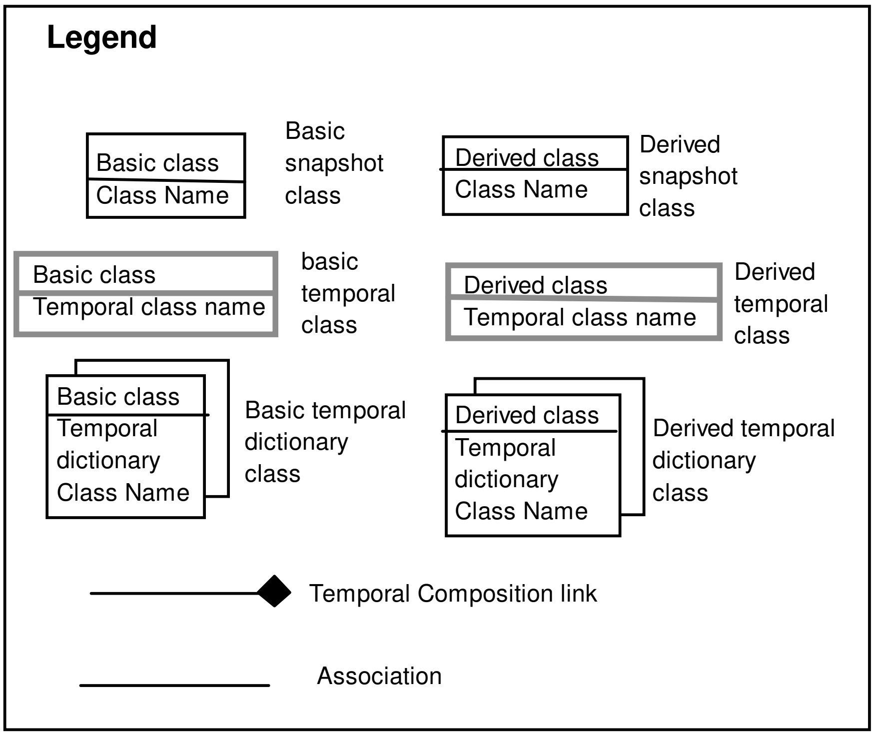


Figure 145: Legend of the global structure graph

4.4.3.3 Global behavior graph

Dynamic view

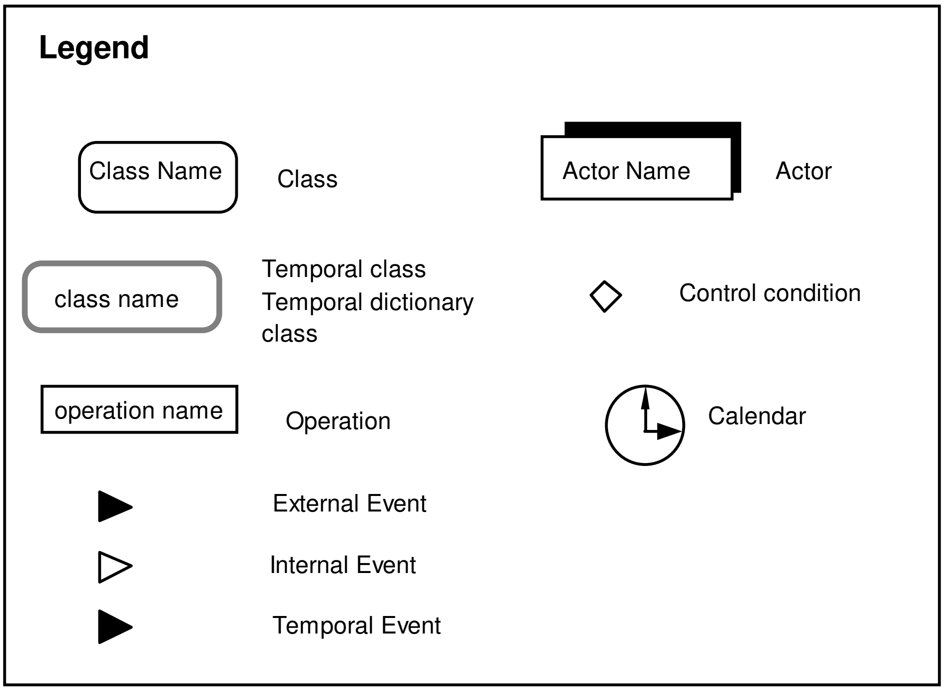


Figure 146: Legend of the global behavior graph

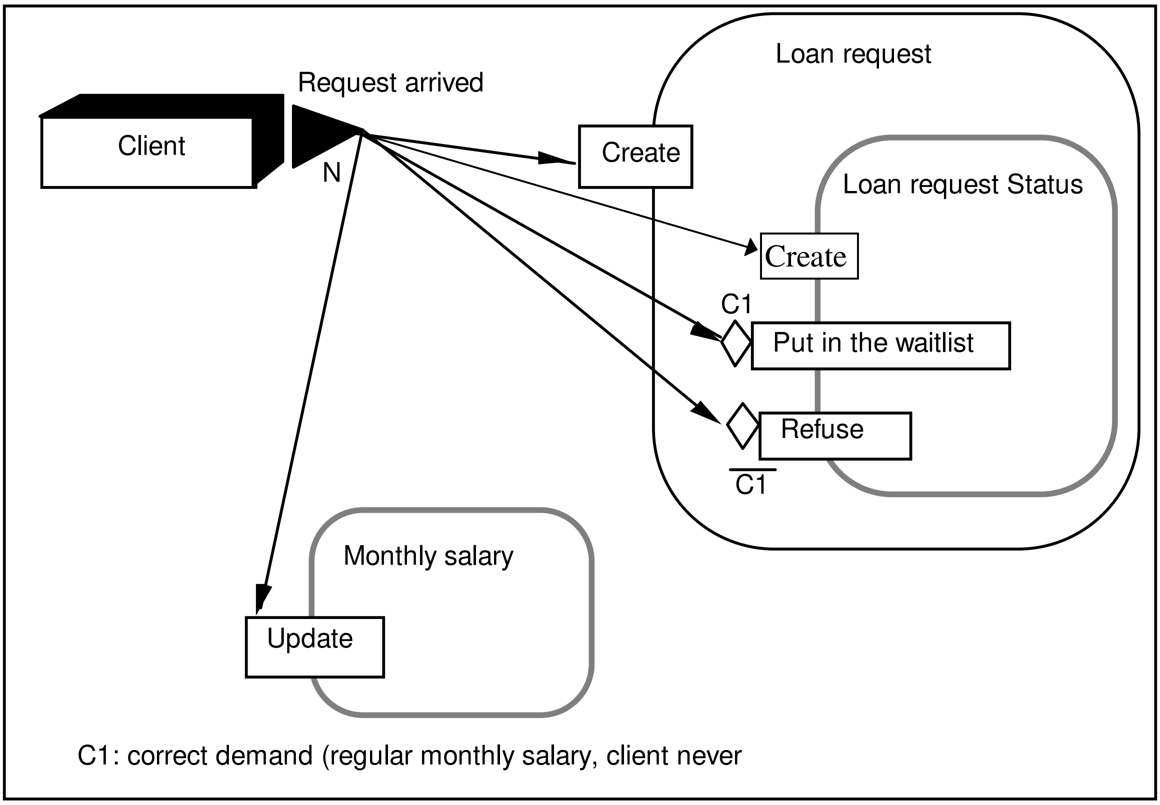


Figure 147: Global behavior graph (part 1)

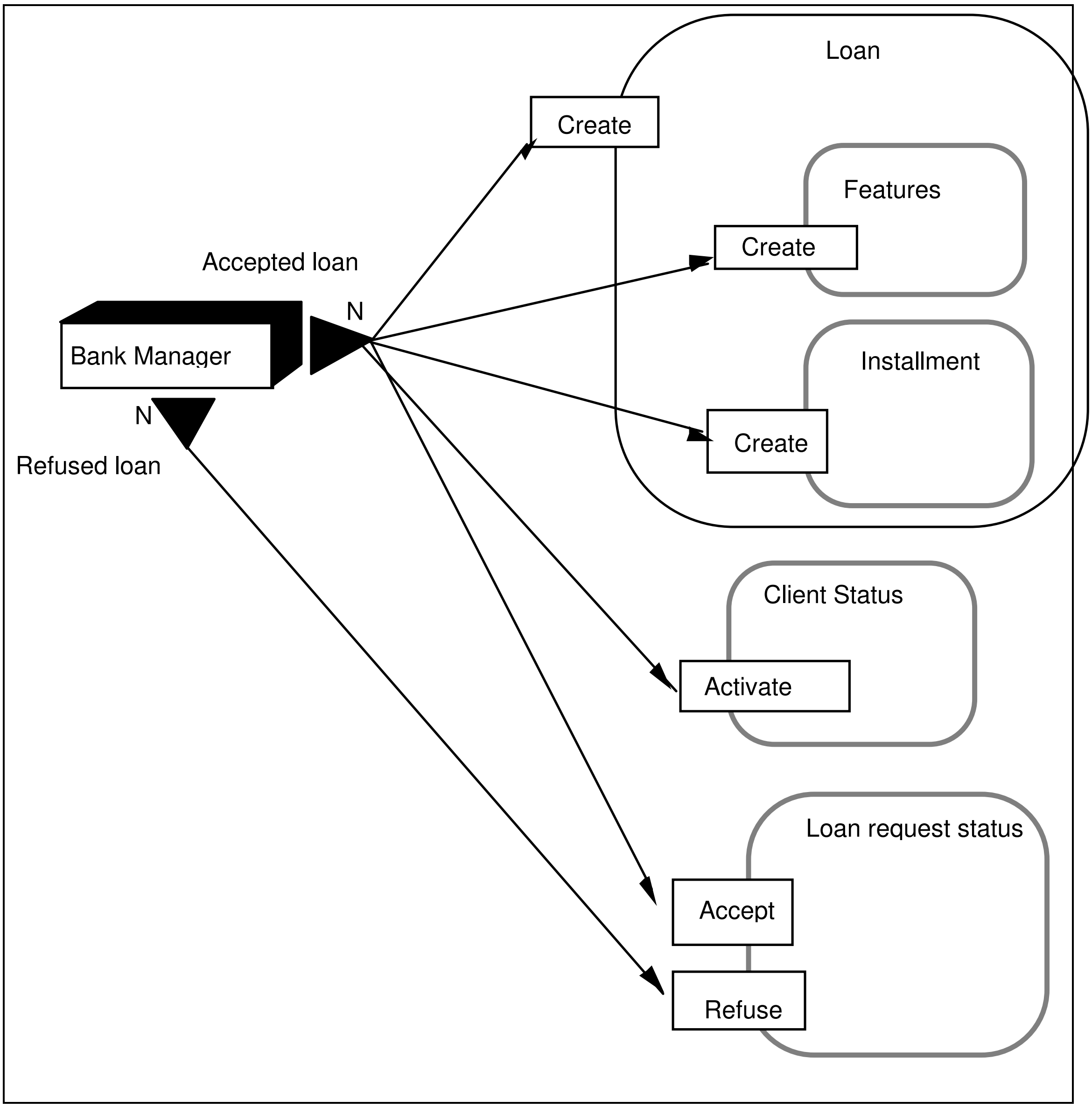


Figure 148: Global behavior graph (part 2)

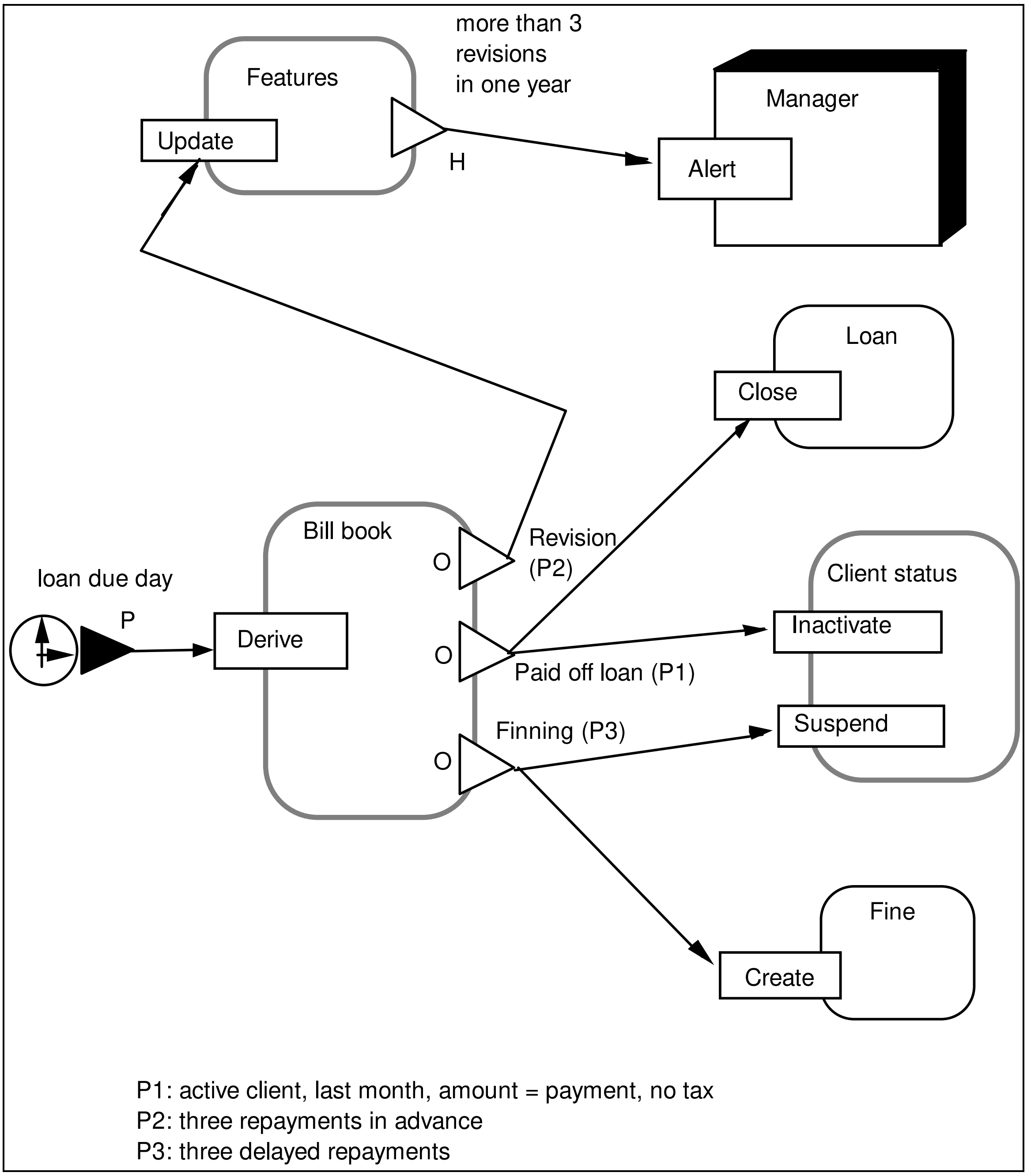


Figure 149: Global behavior graph (part 3)

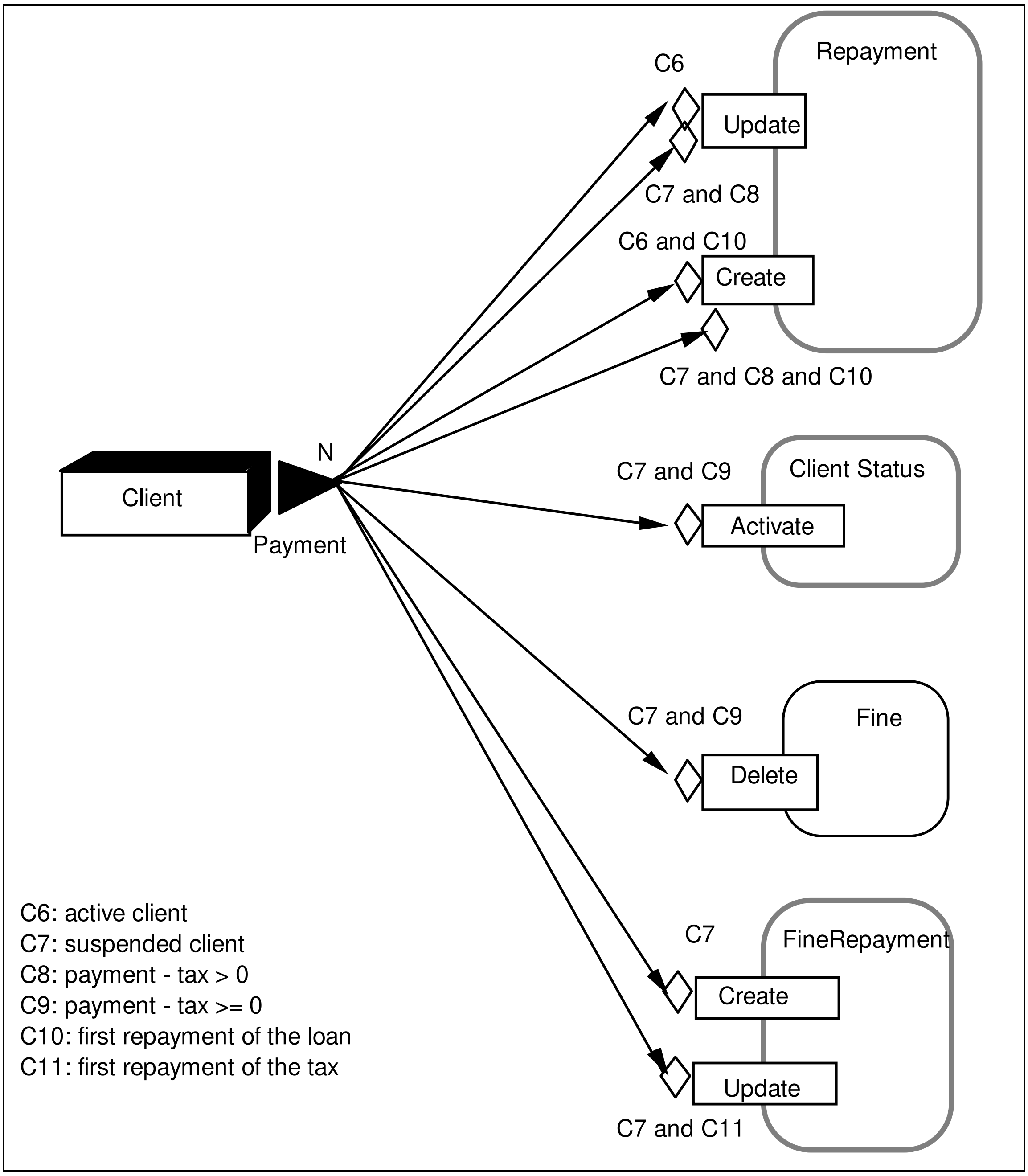


Figure 150: Global behavior graph (part 4)

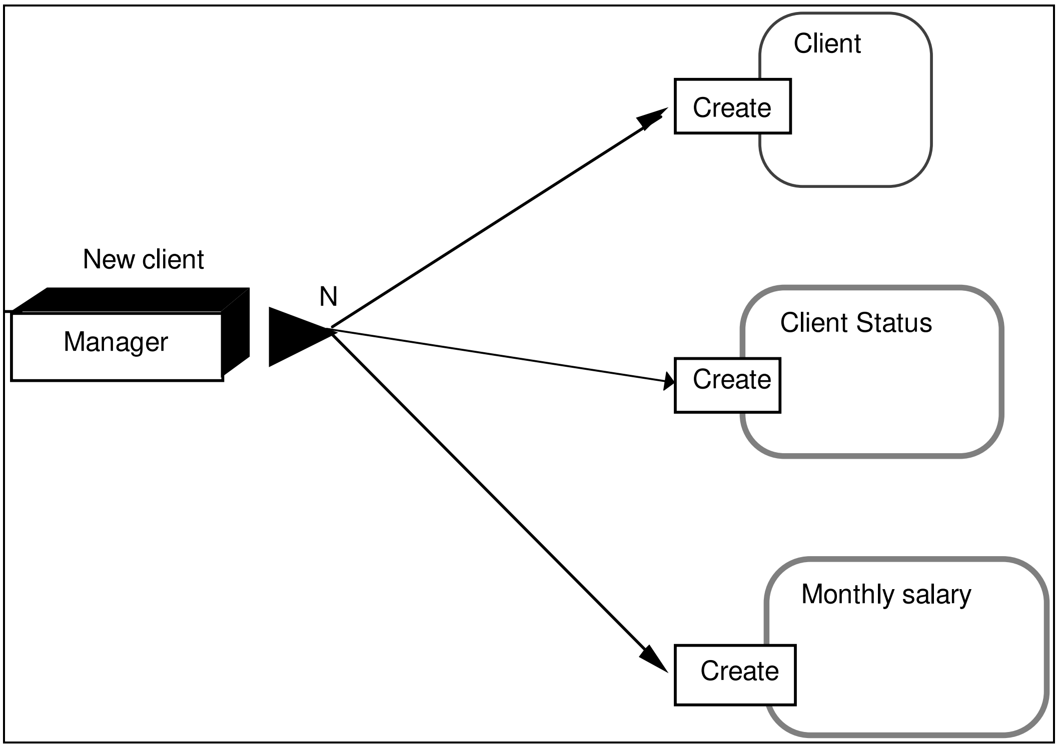


Figure 151: Global behavior graph (part 5)

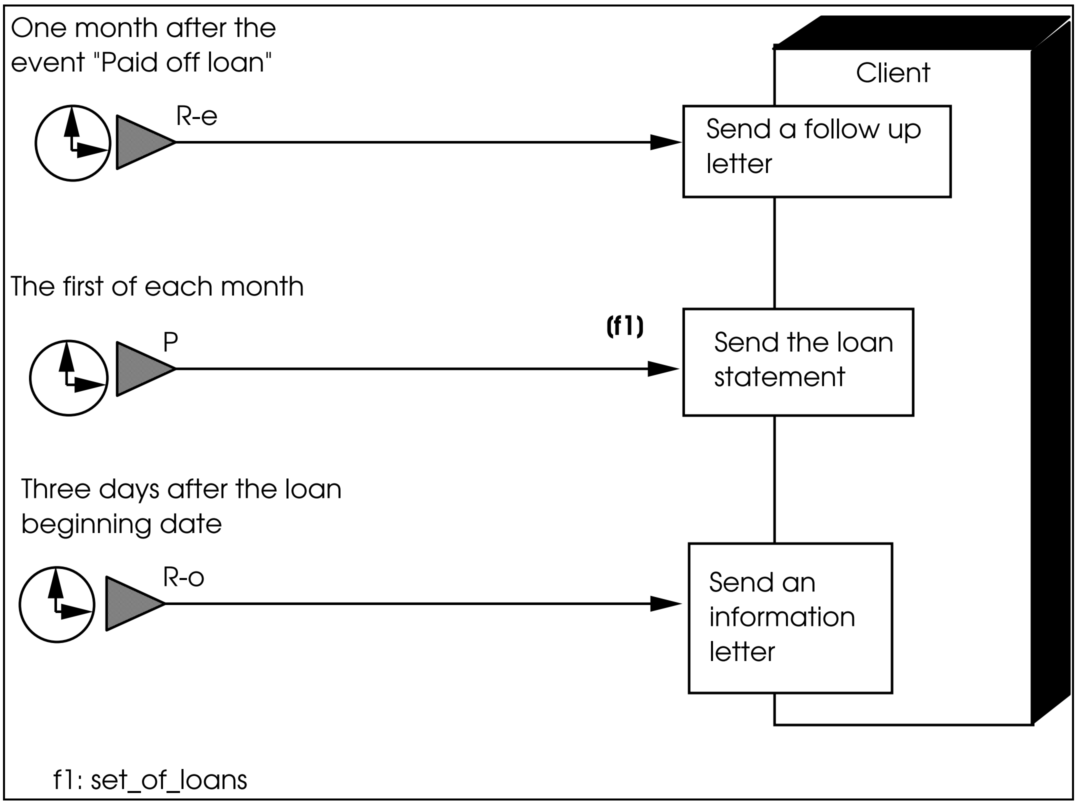


Figure 152:b Global behavior graph (part 6)

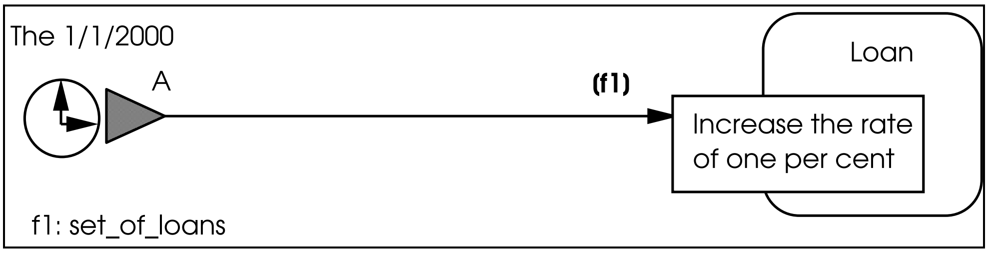


Figure 153: Global behavior graph (part 7)

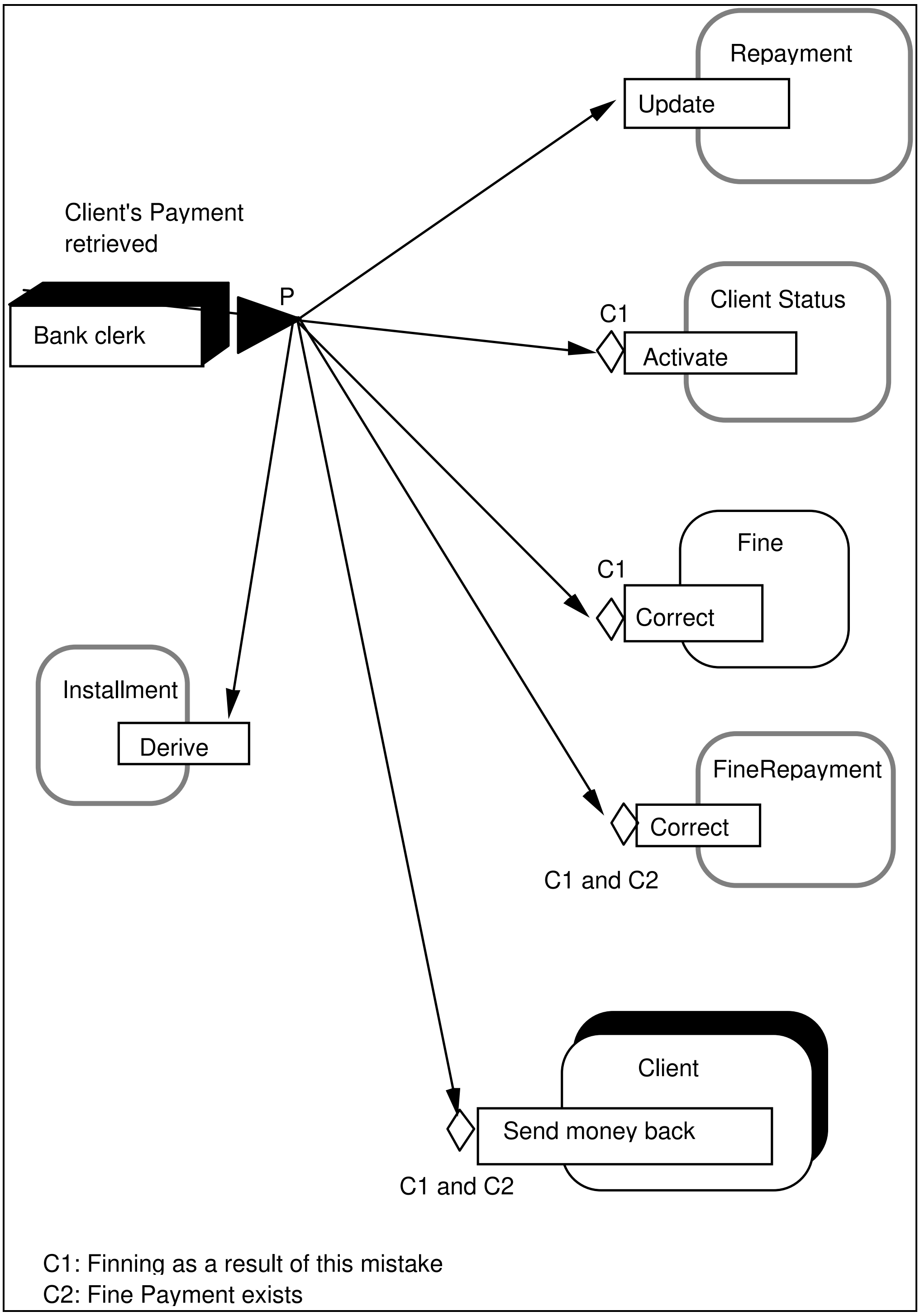


Figure 154: Global behavior graph (part 8)

4.4.3.4 Event induction graph

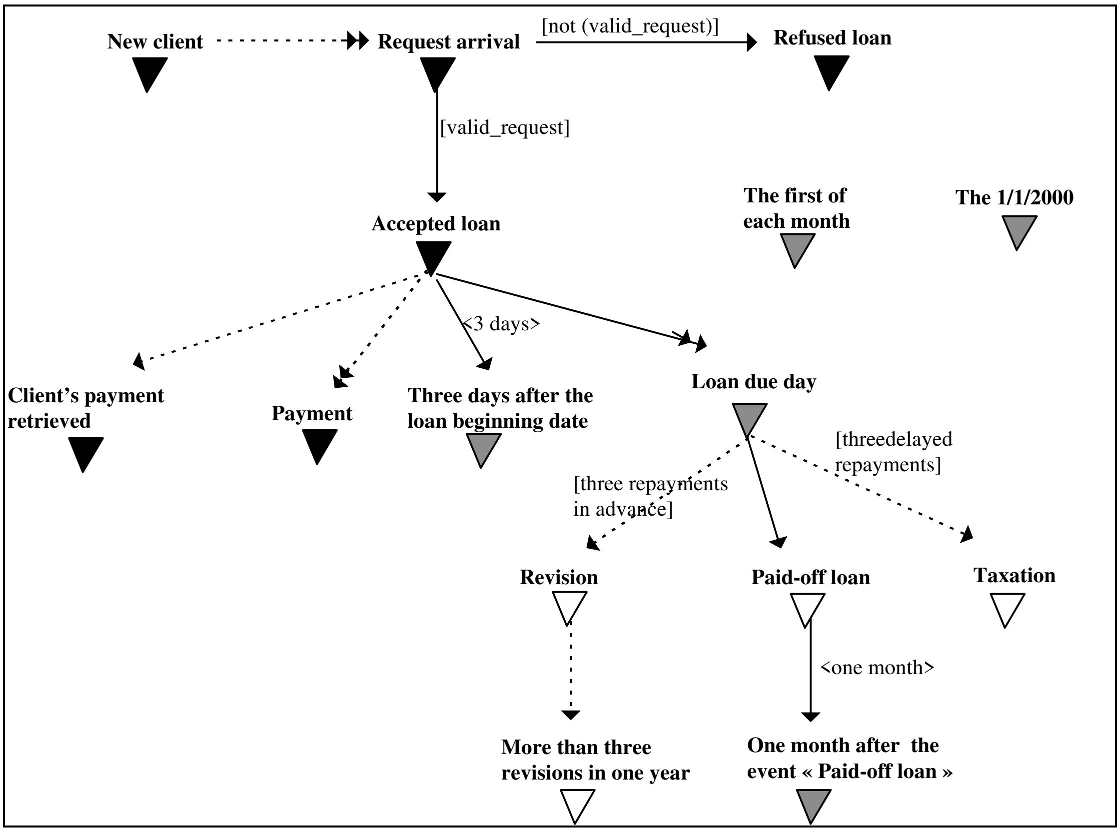


Figure 155: Event induction graph

4.4.3.5 Event concurrence graph

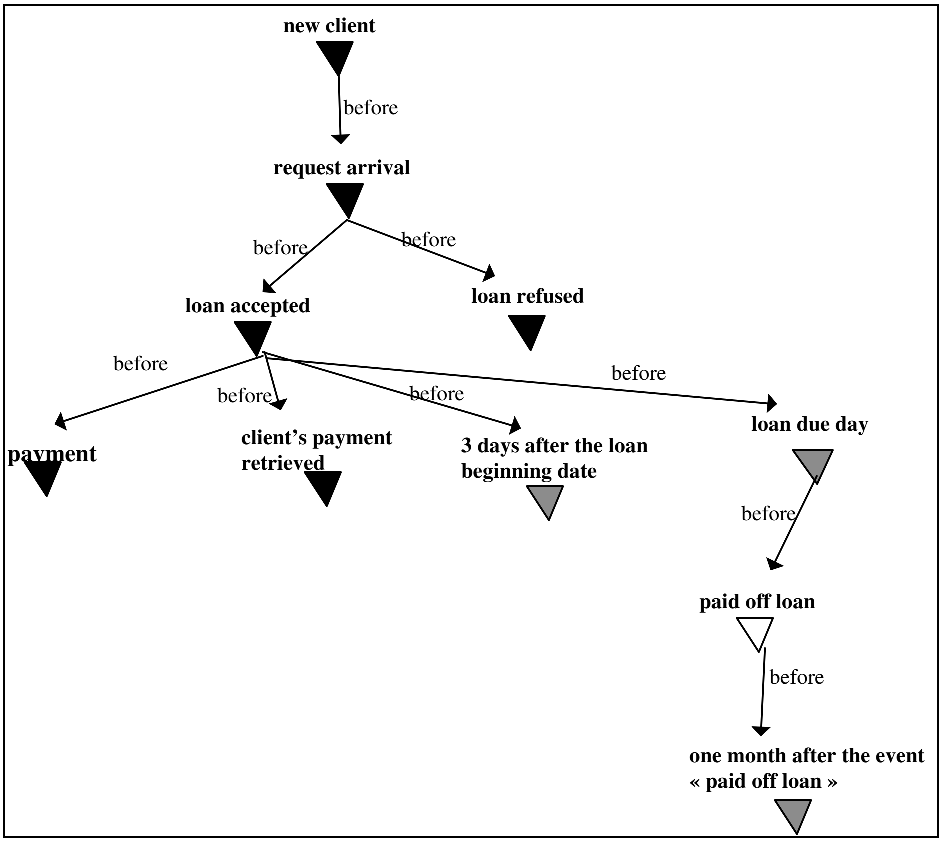


Figure 156: Event concurrence graph

4.4.3.6 Textual specification

basic class CLIENT **extents** clients

properties

client_number : **INTEGER**

name : **STRING**

forename : **STRING**

address : **ADDRESS**

requests : **inverse** (**zeromany**, LOAN_REQUEST::client)

constraints

unique(client_number)

operations

CLIENT create (c: structure(CLIENT)) **type create**

endbasic-class

temporal class for basic class MONTHLY_SALARY **of** CLIENT **extents** monthly_salaries

time dimensions VT INSTANT-A<Gregorian, month> **as state**

properties

ms_amount : **REAL**

operations

MONTHLY_SALARY create (m: structure (MONTHLY_SALARY)) **type create**

update(m: structure (MONTHLY_SALARY)) **type evolution**

endtemporal-class

temporal class for basic class CLIENT_STATUS **of** CLIENT **extents** clients_status

time dimensions VT INSTANT-A<Gregorian, month> **as state**

properties

state : **CLIENTSTATE**

operations

CLIENT_STATUS create (c: structure(CLIENT_STATUS)) **type create**

activate () **type evolution**

inactivate () **type evolution**

suspend () **type evolution**

endtemporal-class

basic class LOAN **extents** loans

properties

loan_number : **INTEGER**

due_date : **INSTANT**<Gregorian,day>

rate : **PERCENTAGE**

loan_request : **asso** (**one**, LOAN_REQUEST, **inverse of** loan)

fines :**inverse** (**zeromany**, FINE : :loan)

constraints

unique(loan_number)

operations

LOAN create (p:structure(LOAN)) **type create**

close() **type evolution**

increase_rate_one_per_cent () **type update**

queries

LOAN $Find_loan (INTEGER)

endbasic-class

basic class LOAN-REQUEST **extents** loan_requests

properties

loan_request_number : **INTEGER**

client : **asso** (**one**, CLIENT, **inverse of** requests)

loan : **inverse** (**zeroone**, LOAN::loan_request) rate : **PERCENTAGE**

amount **: REAL**

duration **: INTERVAL <Gregorian, month>**

constraints

unique(loan_request_number**)**

operations

LOAN_REQUEST create (l: structure(LOAN_REQUEST)) **type create**

queries

LOAN_REQUEST $Find_loan_request (INTEGER)

endbasic-class

temporal class for basic class LOAN_REQUEST_STATUS **of** LOAN_REQUEST **extents** loan_requests_status

time dimensions

VT PERIOD-A<Gregorian, month> **as state**

properties

state : **REQUESTSTATE**;

operations

LOAN_REQUEST_STATUS create() **type create**

put_in_the_waitlist () **type evolution**

refuse () **type evolution**

accept () **type evolution**

endtemporal-class

temporal class for basic class REPAYMENT **of** LOAN **extents** repayments

time dimensions

VT PERIOD-A<Gregorian, month> **as history partial**

TT as history

properties

amount_paid : **REAL**

operations

REPAYMENT create (r: structure(REPAYMENT)) **type create**

update (r: structure(REPAYMENT)) **type evolution**

endtemporal-class

temporal class for basic class FEATURE **of** LOAN **extents** features

time dimensions

VT PERIOD-A<Gregorian, month> **as history partial**

properties

duration : **PERIOD**<Gregorian,month>

amount : **REAL**

theoretic_repayment : **REAL**

operations

FEATURE create (f: structure(FEATURE)) **type create**

update () **type evolution**

events

more_than_three_revisions_in_one_year

predicate « more than 3 revisions in one year »

trigger

bank_manager.alert (context.of)

end-trigger

endtemporal-class

temporal class for derived class INSTALMENT **of** LOAN **extents** instalments

time dimensions

VT PERIOD-A<Gregorian, month> **as history complete**

properties

amount_paid : **REAL**

theoretical_amount_to_repay : **REAL**

amount_remaining : **REAL**

theoretical_amount_paid : **REAL**

derivation function

INSTALMENT create (b : structure(INSTALMENT)) **type create**

derive (INSTANT-A<Gregorian, day>) **type evolution**

events

revision

induction

may context.of.inverse of FEATURE::of..more_than_three_revisions_in_one_year

predicate « Three repayments in advance »

trigger

context.of.inverse of FEATURE::of.update ()

end-trigger

paid_off_loan

induction must with one month **delay**

gregorian.one_month_after_the_event_paid_off_loan (context)

concurrent events

after loan due day

before one month after the event « paid_off loan »

predicate « context.amount_paid=context.of.inverse of FEATURE::of.amount »

trigger

context.of.loan_request.client.inverse of CLIENT_STATUS::of.inactivate ()

context.of.close ()

end-trigger

Finning

predicate « Three delayed repayments »

trigger

context.of.loan_request.client.inverse of CLIENT_STATUS::of.suspend ()

fines.create (context)

end-trigger

endtemporal-class

basic class FINE **extents** fines

properties

fine_number : **INTEGER**

amount : **REAL**

loan **: asso (one,** LOAN**, inverse of** fines**)**

constraints

unique(fine_number**)**

operations

FINE create (p:FEATURE) **type create**

correct () **type correction**

delete () **type delete**

queries

$Find_fine (p: structure (PAYMENT))

endbasic-class

temporal class for basic class FINE_REPAYMENT **of** FINE **extents** fineRepayments

time dimensions

VT PERIOD-A<Gregorian, month> **as history partial**

TT as history

properties

paid_amount : **REAL**

operations

FINE_REPAYMENT creation(p:structure(FINE_REPAYMENT)) **type create**

correct (pold:structure(FINE_REPAYMENT, pnew:structure(FINE_REPAYMENT)) **type correction**

update (p:structure(FINE_REPAYMENT)) **type update**

endtemporal-class

actor-class CLIENT

properties

id of client: **INTEGER**

external operations

send_money_back (b: BANK_CLERK)

send_follow_up_letter ()

send_loan_statement (l:LOAN)

send_information_letter (l:LOAN)

external-events

request_arrived

message REQUEST_ARRIVAL

p : structure (LOAN_REQUESTED)

induction

must IF valid_request (**context**)

bank_manager.accepted_loan (Loan_requests.Find_loan_request(context))

must IF not valid_request (**context**)

bank_manager.refused_loan (Loan_requests.Find_loan_request(context))

concurrent events

after new client

before accepted loan

before refused loan

predicate The client must exist

trigger

loan_request.create (Loan_requests.Find_loan_request(context).inverse of loan_request_status::of)

if valid_request (**context**) **then**

loan_request_status.put_in_the_waitlist (**context**)

if not(valid_request (**context**)) **then**

Loan_requests.Find_loan_request(context).inverse of loan_request_status::of.refuse (**context**)

end-trigger

payment

message PAYMENT

id of loan : INTEGER

amount : REAL

concurrent events

after accepted loan

predicate The loan must exist and must not be closed

trigger

if active_client (context) and first_payment(context) **then**

repayment.create (context)

if active_client (context) and not first_payment(context) **then**

repayment.update (context)

if (suspended_client (context) and difference_positive (context)) **then**

repayment.update (context)

if (suspended_client (context) and difference_positive_or_zero (context)) **then**

Loan_requests.Find_loan(context.id-ofloan).loan_request.client.inverse of CLIENT_STATUS::of.activate ()

Find_fine(context).close ()

if suspended_client () **then**

fineRepayment.create (context)

end-trigger

endactor-class

actor-class BANK_MANAGER

properties

id of bank_manager: **STRING**

external operations

alert (f:FEATURES)

external-events

accepted_loan

message ACCEPTED_LOAN

id of loan_request : INTEGER

induction

may multiple bank_clerk.client's_payment_retrieved ()

must multiple client.payment (Loans.Find_loan(context))

must gregorian.three_days_after_the_loan-beginning-date ()

must multiple gregorian.loan_due_day ()

concurrent-events

after request arrival

before payment

before client's payment retrieved

before three days after the loan beginning date

before loan due day

predicate the loan request must exist

trigger

Loan_requests.Find_loan_request(context).loan_request_status.accept ()

Loan_requests.Find_loan_request(context).client.inverse of CLIENT_STATUS::of.activate ()

loan.create (Loan_requests.Find_loan_request(**context**))

feature.create (Loan_requests.Find_loan_request(**context**))

instalment.create (Loan_requests.Find_loan_request(**context**))

endtrigger

refused_loan

message REFUSED_LOAN

id of loan_request : INTEGER

concurrent-events

after request arrival

predicate the loan request must exist

trigger

Loan_requests.Find_loan_request(context).loan_status.refuse (**context**)

endtrigger

new_client

message NEW_CLIENT

id of client : INTEGER

c : structure (CLIENT)

m : structure (MONTHLY_SALARY)

induction

may client.request_arrived (client)

concurrents-events

before request arrival

predicate the client does not exist

trigger

client.create (**context**)

client_status.inactivate (**context**)

monthly_salary.create (**context**)

endtrigger

endactor-class

actor-class BANK_CLERK

properties

id of bank_clerk: **STRING**

external-events

client's_payment_retrieved

message CLIENT'S_PAYMENT_RETRIEVED

p : structure (REPAYMENT)

concurrents-events

after accepted loan

predicate the loan must exist

trigger

repayment.update (context)

if taxation_mistake(context) **then**

loans**.**Find_loan(context.id of loan).loan_request.client .inverse of CLIENT_STATUS::of.activate ()

fines.Find_fine(context).delete ()

if taxation_mistake (context) and tax_payment_exist (context) **then**

fines.Find_fine(context). Inverse of fineRepayment::of.delete ()

client.send_money_back (context.idofbank_clerck)

endtrigger

endactor-class

calendar class Gregorian

temporal-events

loan_due_day with **time relative to object**

induction

may IF three_repayments_in_advance(context)

instalment.revision (context)

may IF (three delayed repayment)

instalment.finning (context)

must instalment.paid_off_loan (context)

concurrent events

after accepted loan

before paid off loan

predicate «day(now()) = Loan. due day »

trigger

instalment.derive (context.time) **type evolution**

end-trigger

one_month_after_the_event_"paid_off_loan" **with time relative to event**

concurrent events

after paid off loan

predicate « One month after *paid_off_loan*.time »

trigger

client(context.loan).send_follow_up_letter ()

end-trigger

the_first_of_each_month **with periodic time**

predicate « For each first day of a month »

trigger

FOR EACH c IN set_of_loans c.loan_request.client.send_loan_statement (c)

end-trigger

three_days_after_the_loan_beginning_date **with time relative to object**

concurrent events

after accepted loan

predicate « three days after Loan.beginingdate »

trigger

context.loan_request.client..send_information_letter (context)

end-trigger

the_1/1/2000 **with absolute time**

predicate « at the instant 1-1-2000 »

trigger

FOR EACH l **in** set_of_loans l.loan.increase_the_rate_of_one_per_cent ()

end-trigger

endcalendar-class

domain CLIENTSTATE :

{active, inactive, suspended}

enddomain

domain REQUESTSTATE :

{refused, accepted, waiting}

enddomain

domain PERCENTAGE : [0..100]

enddomain

triggering-condition

boolean valid_request (c:COMING_OF_THE_DEMAND)

comment if the client has a regular monthly salary and has never been syspended

endtriggering-condition

triggering-condition

boolean active_client (p:PAYMENT)

comment if the client is active

endtriggering-condition

triggering-condition

boolean suspended_client (p:PAYMENT)

comment if the client is suspended

endtriggering-condition

triggering-condition

boolean difference_positive_or_zero (p:PAYMENT)

comment if the difference between the payment and the tax is positive or equal to zero

endtriggering-condition

triggering-condition

boolean difference_positive (p:PAYMENT)

comment if the difference between the payment and the tax is positive **endtriggering-condition**

triggering-condition

boolean taxation_mistake (p:CLIENT'S_PAYMENT_RETRIEVED)

comment a payment is retrieved

endtriggering-condition

triggering-condition

boolean tax_payment_exist (p:CLIENT'S_PAYMENT_RETRIEVED)

comment a payment exist for this tax

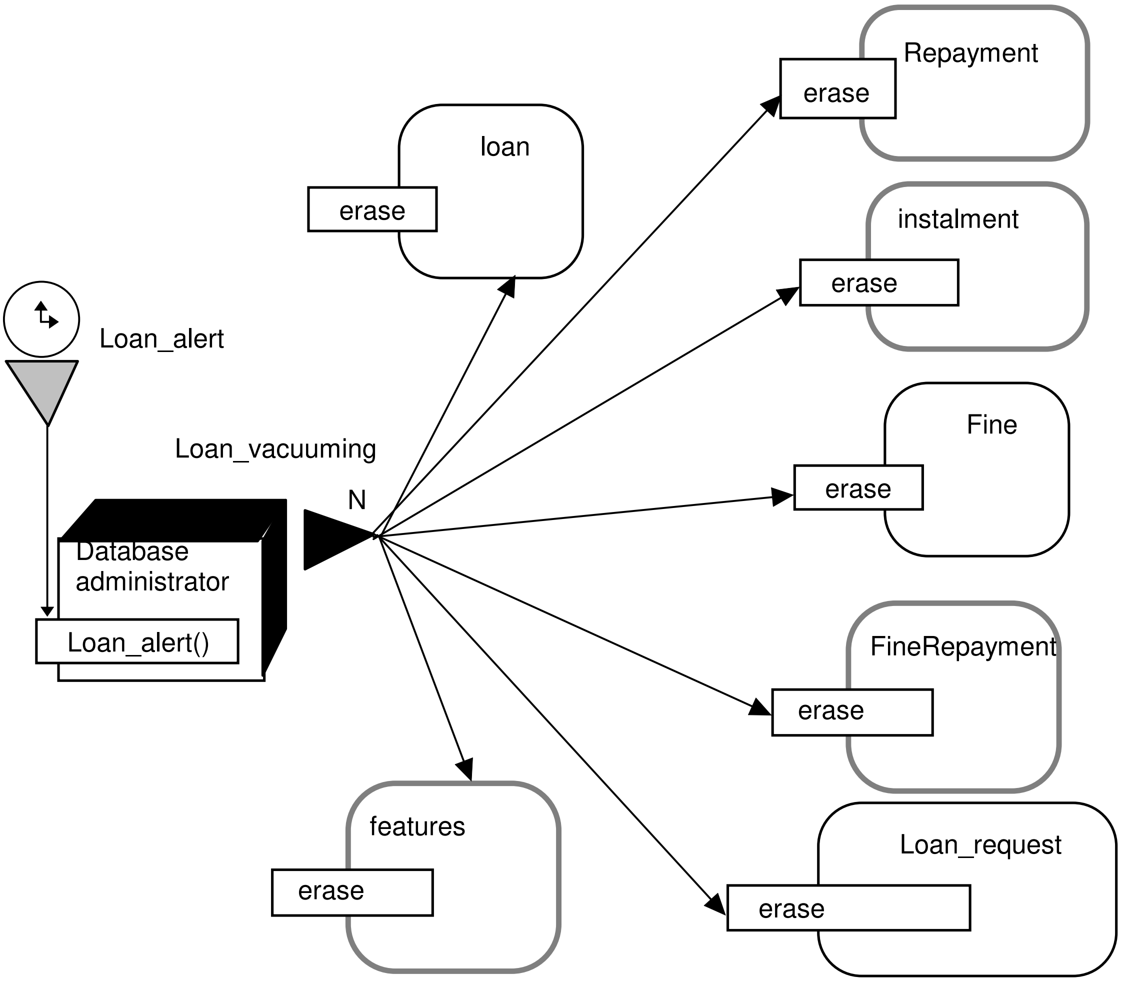
endtriggering-condition

4.4.3.7 Vacuuming rules

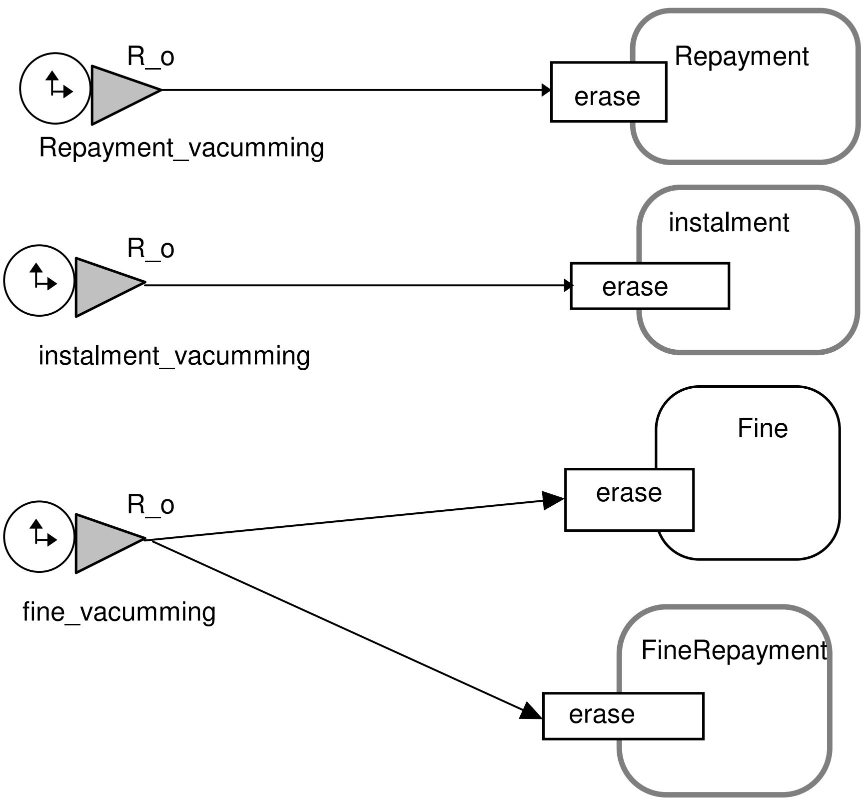
The rules applied for the administration of the database are the following :

- erase temporal information related to current loans when they are more than ten years old (*instalment, repayment and fine* temporal classes). The triggering of these vacuuming activities are automatic when the situation is detected.
- erase the information related to loans when they are closed during more than 5 years. The triggering of the vacuuming activities is done manually by the database administrator when it is alerted

Dynamic schema



Vacuuming schema (Part 1)



Vacuuming schema (Part 2)

calendar class Gregorian

temporal-events

repayment_vacuuming **with time relative to object**

predicate «now() - repayment.firstVT() > 10 years »

trigger

context.erase ()

end-trigger

instalment_vacumming **with time relative to object**

predicate «now() - instalment.firstVT() > 10 years »

trigger

context..erase ()

end-trigger

fine_vacumming **with time relative to object**

predicate « now() - fine.firstVT() > 10 years »

trigger

context..erase ()

end-trigger

loan_alert **with time relative to object**

predicate « now() - loan.lastTT() > 10 years »

trigger

if loan_closed(context) then **context.**.erase ()

end-trigger

endcalendar-class

actor-class database_administrator

properties

id of database_administrator: **STRING**

external-events

loan_vacuuming

message LOAN_VACUUMING

id of loan : INTEGER

predicate « loan must exist »

trigger

Find_loan(context).erase ()

Find_loan(context).inverse of INSTALMENT : :of.erase ()

Find_loan(context).inverse of REPAYMENT : :of.erase ()

Find_loan(context).loan_request.erase ()

Find_loan(context).features.erase ()

FOR EACH f IN **Find_loan(context)**.fines f.erase ()

FOR EACH f IN **Find_loan(context)**.fines f.fine_repayment.erase ()

end-trigger

endactor-class

5. APPENDIX: UML COMPLIANCE

Contents

5.1 Introduction

5.1.1 UML purpose

The Unified Method Language (UML) is owned by Rational Company and it is issued from three well known OO methods :

- OMT designed by James Rumbaugh,
- OO method designed by Grady Booch,
- OOSE of Ivar Jacobson.

UML is a method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. By unifying these three leading object-oriented methods, UML provides the basis for a defacto standard in the domain of object-oriented analysis and design founded on a wide base of user experience.

Version 0.91 of UML is a proposal for an OO method submitted to the OMG group. The final decision is not made but the proposal has a great chance to become a standard in the OO methods area. The UML compliance is an important issue for the TOOM method.

5.1.2 UML notation

UML has defined the external format of diagrams. A diagram is a projection upon the elements of a model; the same elements may appear in multiple diagrams and a given diagram may present some - but not necessarily all - of the details of any particular element.

- *Class diagram*. It is the core of an UML model. This view shows the logical static structure of a system: its contents and their relationships to each other. In its most fundamental form, it shows the elements composing the state of a system (those elements that must be remembered from moment to moment).

- *Use case diagram*. It is a generic description of an entire transaction involving several objects. It can also describe the behavior of a set of objects, such as an organization.

- *Sequence diagram*. It shows the interactions among a set of objects in temporal order. A text dialogue can accompany or replace such a diagram. It emphasizes the temporal flow of behavior.

- *Collaboration diagram*. It is a scenario diagram that shows the sequence of messages that implements an operation or a transaction.

- *State diagram*. It describes the temporal evolution of an object of a given class in response to interactions with other objects inside or outside the system. Each class may have a state diagram to describe its dynamic behavior, although not all classes have interesting dynamic behavior.

- *Components diagram*. It represents the physical modules that provide the defining occurrence of these logical elements.

- *Deployment diagram.* It specifies and visualizes the physical topology upon which a software system executes.

5.1.3 Mapping from OOM and TOOM to UML

Behavior in temporal applications is an important aspect to study. This is why we choose for OOM an event oriented approach and a strong integration of behavioral modeling with the two other perspectives.

In the OOM approach, the behavior of the application is described in terms of events and triggering rules. An event is a particular state change of objects which plays the role of a stimulus implying the system reaction. The reaction of the system to an event is expressed through triggering rules in terms of operations to perform on objects. This is a causal approach of dynamics in the sense of cybernetics and causal theory of organizations. It has numerous advantages among which are (a) the possibility of both a global and a local study of the system behavior and (b) a tight coupling of the three perspectives. The important principle of *minimality* (minimum set of concepts for a maximum representability) which is a well established principle in science is then, fully followed.

On the contrary, UML uses states models. A states model is only local to a class; it exhibits the possible states of an object in the class and the allowed transitions between these states. This approach lacks the representation of the global system behavior; this makes for example, the validation of the specification more difficult. In addition, the behavioral modeling is disconnected from structure and function modeling, which is bad.

The UML method is an open method which can be extended with the concept of stereotype. A stereotype represents the metaclassification of an element. It represents a class within the UML metamodel itself, i.e., a type of modeling element. It is a way to allow users (methodologists, tool builders, and user modelers) to add new kinds of modeling types. Every element in the UML may have at most one stereotype; this stereotype is omitted when the normal predefined UML modeling semantics suffice. There is a separate list of stereotypes for each kind of UML element. Some stereotypes are predefined, but users are allowed to define their own. Stereotypes have semantic implications which may be specified for every specific stereotype value. UML recommends that tools allow every stereotype to be further distinguished by style (color, line thickness, fill pattern, font) as well as by icon. The purpose of this improvement is to permit tools builders and users to tailor the UML's graphical syntax in controlled ways, and to permit model to have special visual cues.

This report is organized as followed. Section 5.2 presents the explanation of the mapping from OOM and section 5.3 the mapping from TOOM.

5.2 Mapping from OOM to UML

The UML notation is prolific of diagrams. We choose to use only three of them for the UML compliance of the OOM, with the purpose to ease the bridge between the two notations.

The *Class Diagram* is constructed with the following stereotypes: the objects, the actors, the calendar and the events. These classes allow to know the structure of the application. As shown in Figure 157, the links between the events and the other, classes aims to show a part of the behavior of this application.

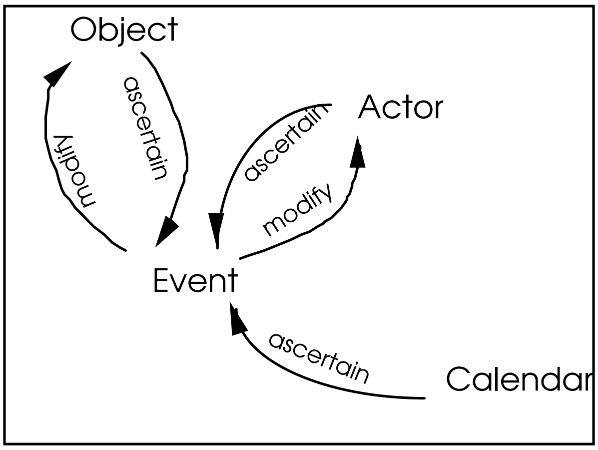


Figure 157: links between the classes in the Class Diagram

The *Sequence Diagrams* permit to show the links between the events, the operations and the modified classes. These diagrams aim to complete the behavioral view of the application.

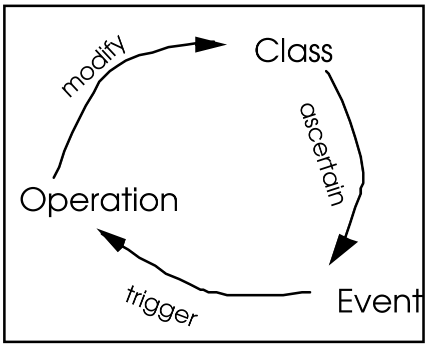


Figure 158: behavior expressed in a Sequence Diagram

The *Use Case Model* allows to have a general view of the application and of the external and temporal events which have an effect on it.

The Section 5.2.1 and 5.2.2 present the mapping to the Class Diagram, Section 5.2.3 the mapping to the Sequence Diagrams and Section 5.2.4 the mapping to the Use Case Model. A Case study is given in Section 5.2.5.

5.2.1 From the Global Structure Graph to the Class Diagram

5.2.1.1 Class

In the OOM, a class is composed of properties, operations, queries, constraints and internal-events. However, in the UML, a class in only composed of properties and operations. If we keep this way of modeling, we loose all the benefits of the event-driven approach. As a consequence, we choose to use the concept of *stereotype* explained in the 0.91 addendum of the UML in order to have the concept of event in the class diagram.

A *class* in the OOM will be represented in the UML as a class with the stereotype *<object>*.

The *properties* and *operations* of the class will be the same in the two models.

The queries are just an other kind of operations, so they will be represented in the set of operations of the class.

The constraints can't be represented in the class, so we will use the concept of *note* to represent them in the diagram.

The internal events are represented in the UML compliance as a class with the stereotype *<event>*. This particular class will automatically have the operation *trigger-off()* in order to have the possibility to set off the event when it is necessary. As a consequence, there is a link between the object class and the event with the association *ascertain*. As for the constraints, the event's predicate is represented graphically with the concept of *note*.

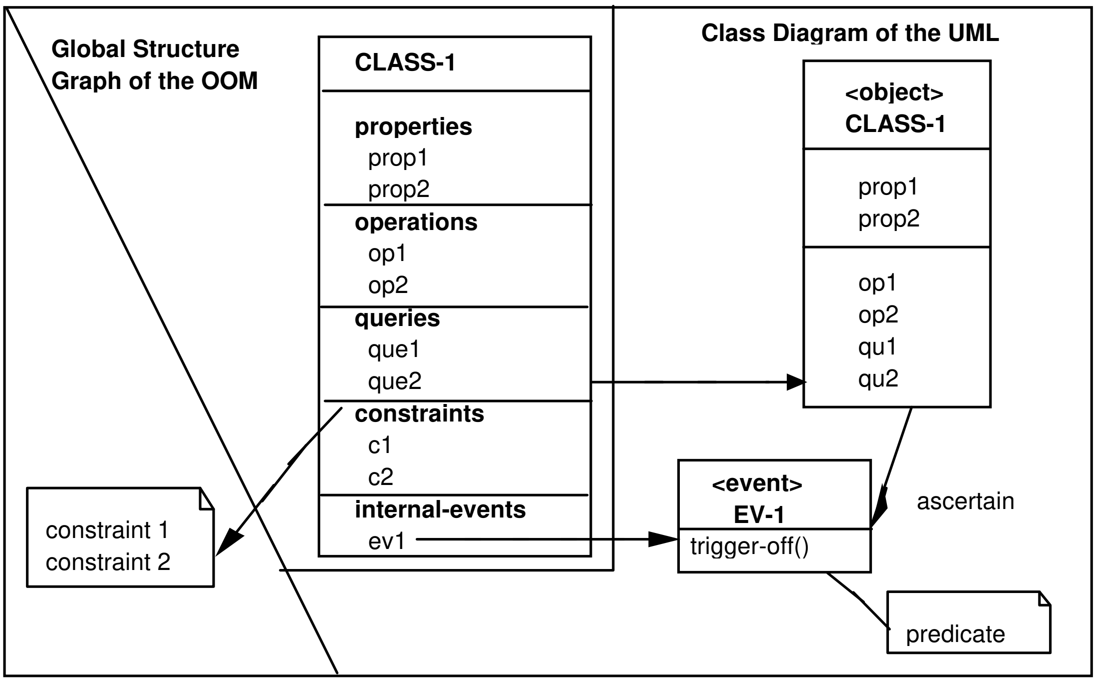


Figure 159: mapping of an object class

5.2.1.2 Actor

The actors can be represented in the global structure diagram of the OOM. It is possible to do the same in the UML using the *stereotype* concept. As a result, an actor will be noted as a class with the stereotype *<actor>*.

The external events will be represented in the same way than the internal events for the classes: a class with the stereotype *<event>* and the operation *trigger-off()*. The actor and the event will be linked by the association *ascertain* and the predicate represented in a *note*.

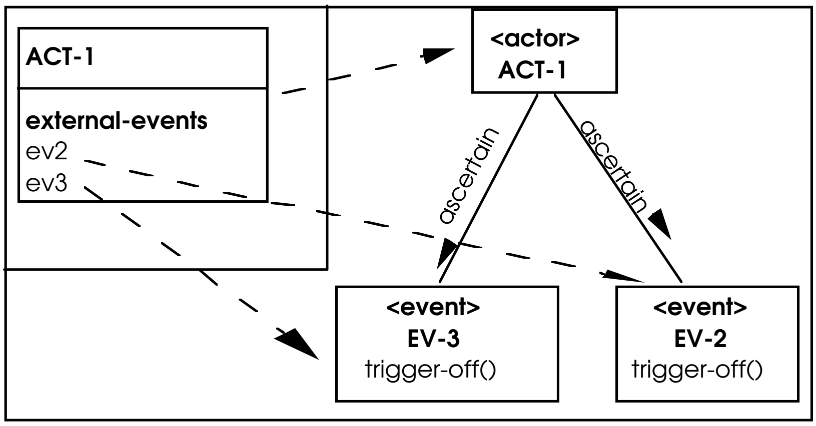


Figure 160: mapping of an actor class

5.2.1.3 Calendar

The representation of the calendar class follow the same principles as the object and actor classes. A new stereotype *<calendar>* will be used to allow its notation in the class diagram.

The temporal events will be extracted as the other events with the stereotype *<event>*, linked at the calendar class with the association *ascertain* and with a *note* explaining the predicate.

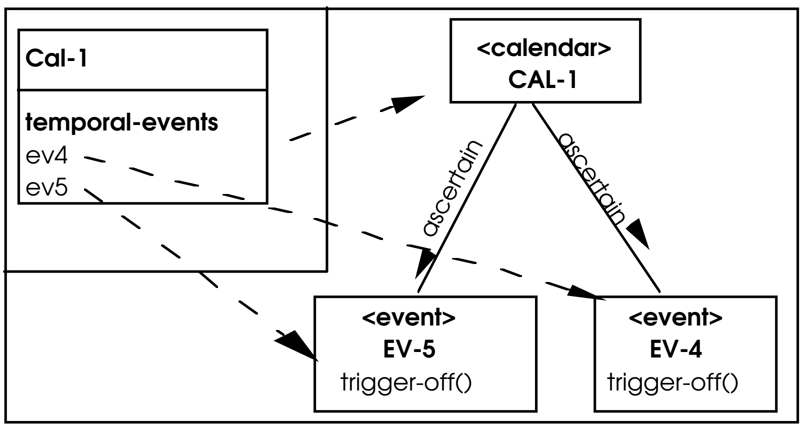


Figure 161: mapping of a calendar class

5.2.1.4 Links

There is no difficulties in the mapping of the structural links since all the links concepts of OOM exist in the notation of UML.

5.2.1.5 Association

The graphical notation of the OOM is replaced by the numeral meaning of it. If the cardinality represents a range, it can be noted with two numbers: one for the minimal and one for the maximal values. If it corresponds to a constant number, this one is represented alone. The cardinality N is represented with the '*' symbol.

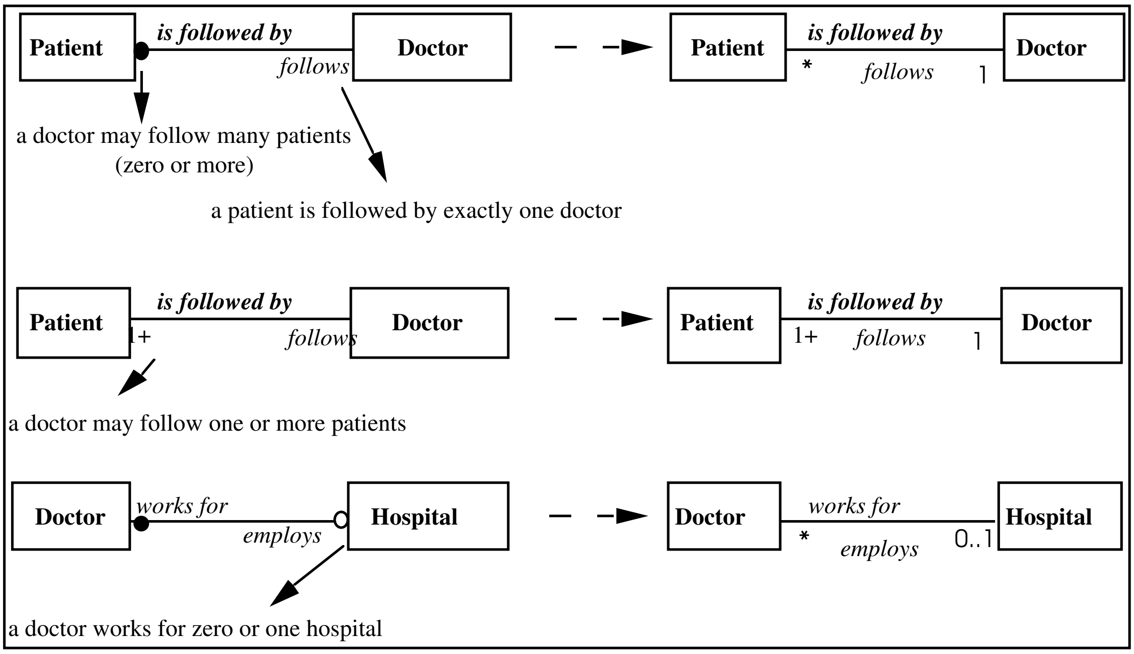


Figure 162: mapping of the association link

5.2.1.6 Aggregation

The aggregation symbol is the same in the two methods. The only notation that changes is the cardinality symbol as for the association link.

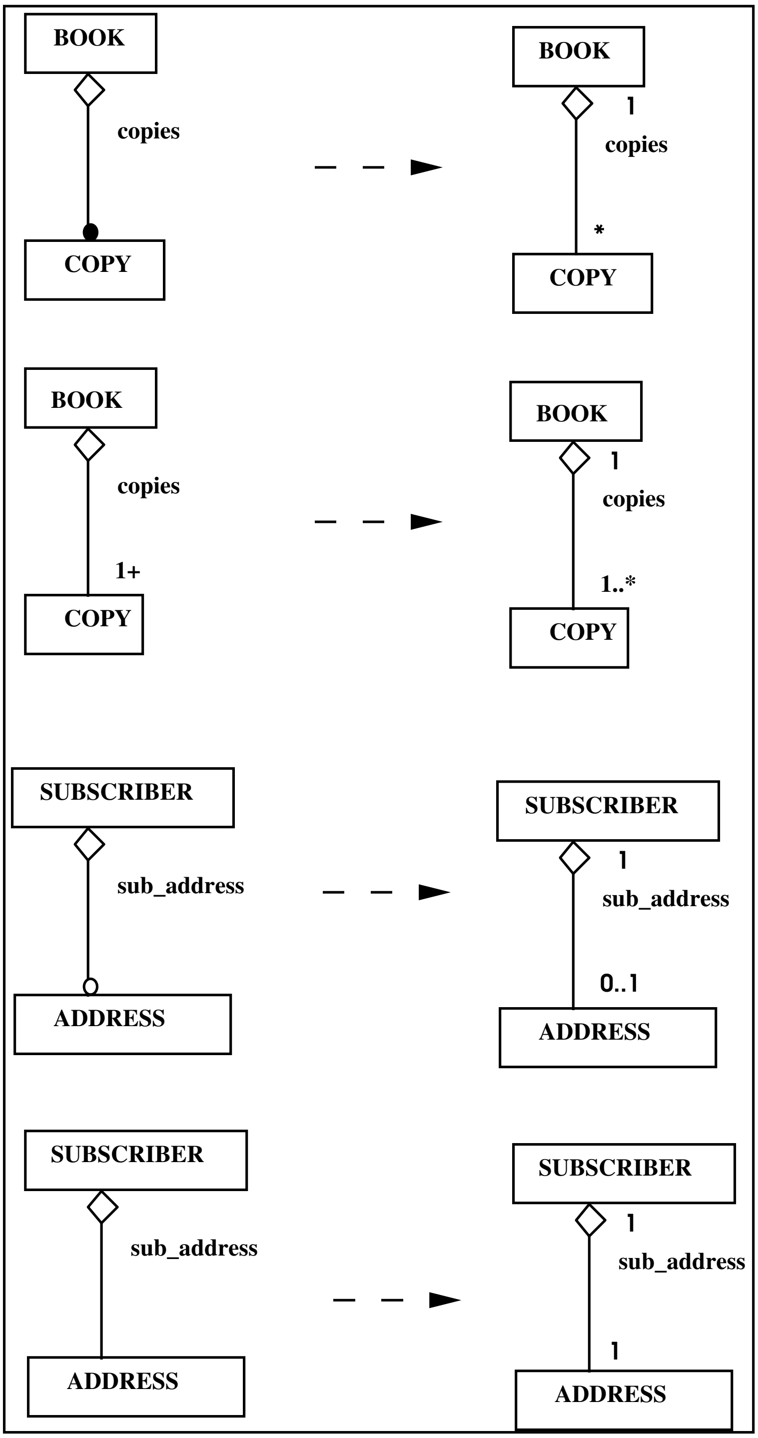


Figure 163: mapping of the aggregation link

5.2.1.7 Inheritance

The inheritance is expressed in UML as a hollow arrow-head instead of a triangle in OOM. The notation for the inheritance constraints in OOM was the hollow triangle for the disjoint constraint, the black triangle for the overlapping constraint, the equal symbol with hollow triangle for the covering constraint and the equal symbol with the black triangle for the partition constraint. In the UML, the corresponding symbols are all represented with the hollow arrow-head adding to the corresponding notation {disjoint}, {overlapping}, {covering} and {partition}.

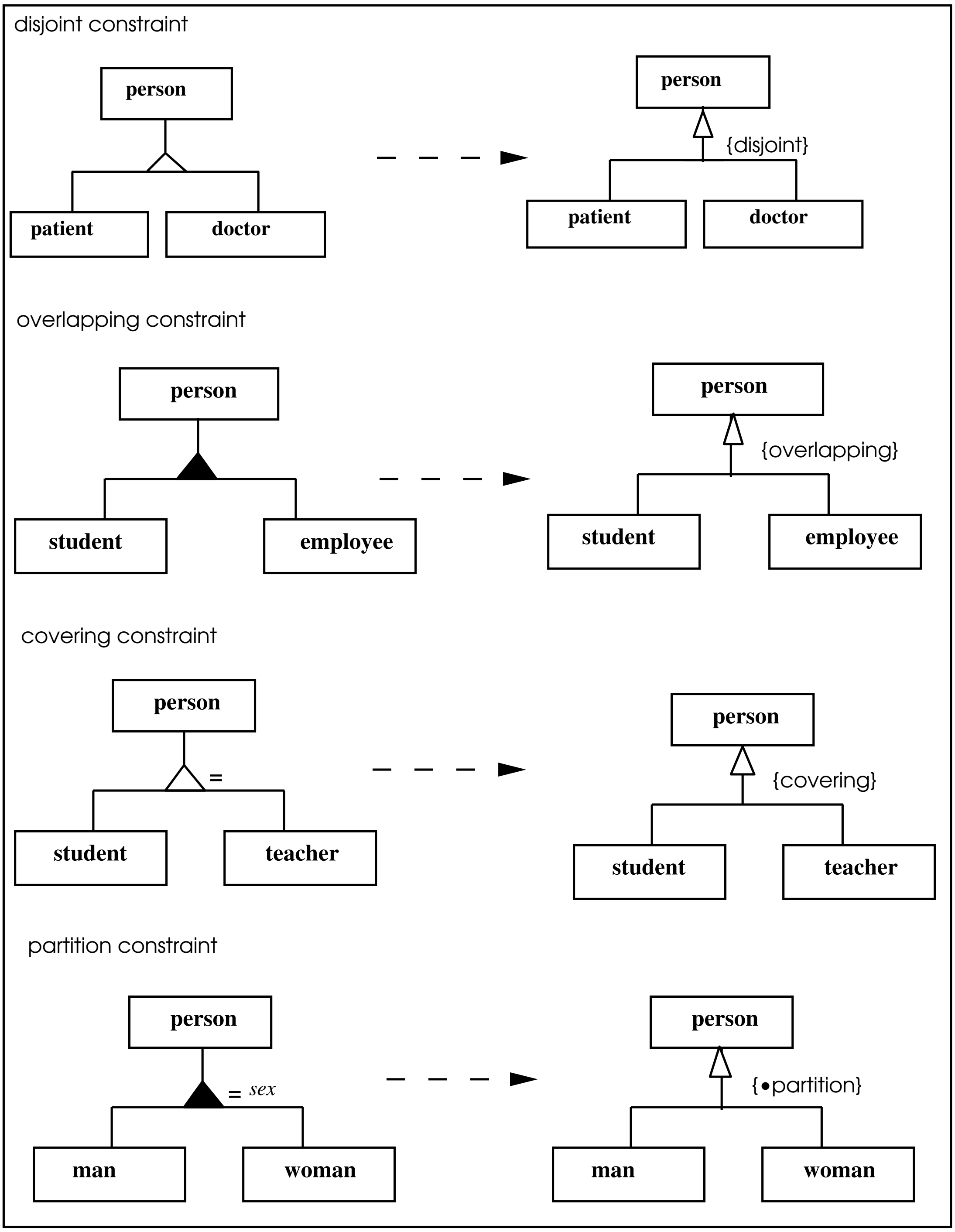


Figure 164: mapping of the inheritance link

5.2.1.8 Summary

Global Structure Diagram of OOM	**Class Diagram of UML**
object class	stereotype <*object*>
attributes of the class	attributes of the class
operation encapsulated in the object class	operation encapsulated in the class
queries encapsulated in the object class	queries encapsulated in the class as operations
event encapsulated in the object class	stereotype <*event*> class *ascertain* event
predicate of the internal event	*note* with the predicate
calendar class	stereotype <*calendar*>
event encapsulated in the calendar class	stereotype <*event*> calendar *ascertain* event
predicate of the temporal event	*note* with the predicate
actor class	stereotype <*actor*>
event encapsulated in the actor class	stereotype <*event*> actor *ascertain* event
predicate of the external event	*note* with the predicate

5.2.2 From the Global Behavior Graph to the Class Diagram

5.2.2.1 Actor

Al though the actors are not every times represented in the global structure graph, they are always found in the global behavior graph of the OOM. As a consequence, if the class diagram doesn't contain the actors, it is possible to complete it with this graph. In the same way, it is possible to find the external events associated with the actor.

The behavior graph can also be useful to find the associations *modify* between the external events and the objects.

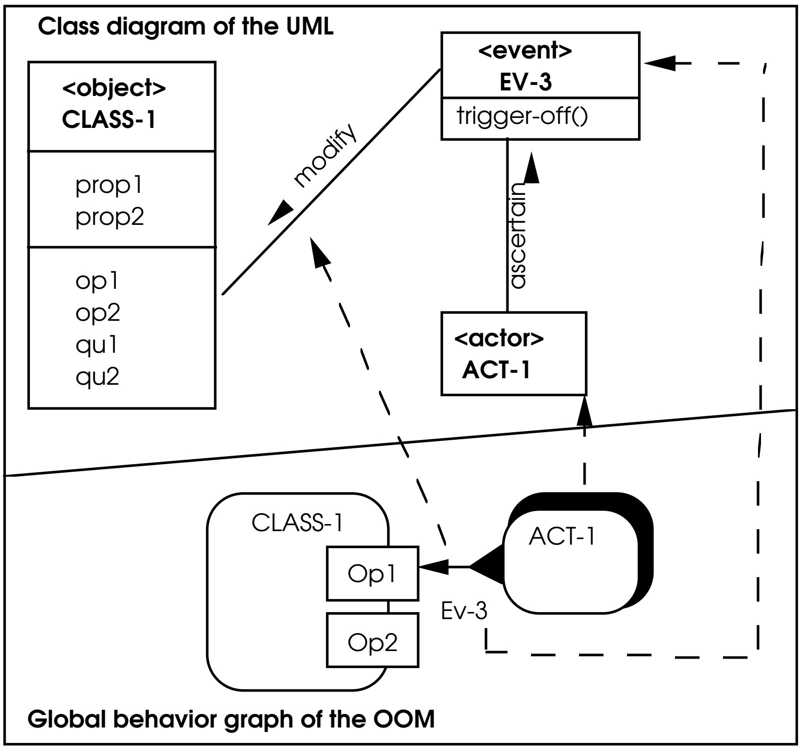


Figure 165: mapping of an actor

5.2.2.2 Calendar

In the same way, the calendar is not always represented in the global structure graph. The global behavior graph can be useful to find the temporal events and to represent the calendar class.

As for the actors, this graph can be useful to find the associations modify between the temporal events and the modified classes.

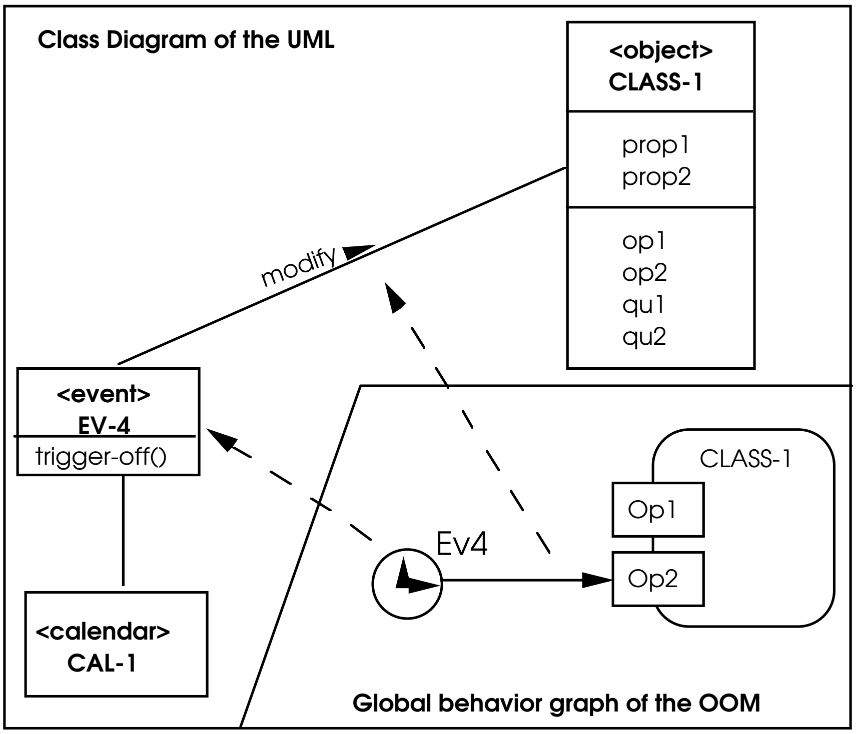


Figure 166: mapping of a calendar

5.2.2.3 Class

On the contrary, the classes are always represented in the global structure graph. The global behavior graph can still be useful to find the associations *modify* between the internal events and the modified classes.

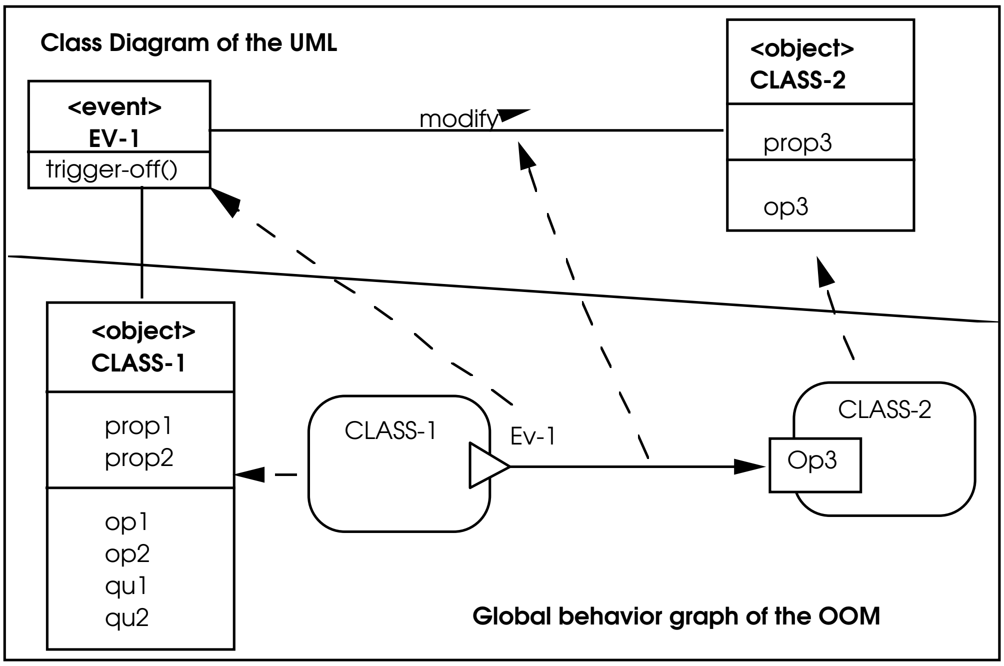


Figure 167: mapping of a class

5.2.2.4 Summary

Global Behavior Diagram of OOM	**Class Diagram of UML**
class	stereotype <*object*>
actor	stereotype <*actor*>
calendar	stereotype <*calendar*>
internal event	stereotype <*event*> class1 *ascertain* event event *modify* class2
external event	stereotype <*event*> actor *ascertain* event event *modify* class
temporal event	stereotype <*event*> calendar *ascertain* event event *modify* class

5.2.3 From the Global Behavior Graph to the Sequence Diagram

The sequence diagrams are useful to show which operations are triggered by the events. We draw one sequence diagram for each event defined.

A sequence diagram contains at least three vertical lines: one for the release object, one for the event and one (or more) for the object(s) modified by this event.

It also contains at least two horizontal lines: one for the *trigger-off()* operation between the release object and the event and one (or more) for the operation(s) that modify the object(s) between this (these) one(s) and the event.

5.2.3.1 External event

The actor is represented as the first vertical line and the external event as the second one. The operation *trigger-off()* is the horizontal line that makes the link between them.

Each triggered operations corresponds an horizontal line from the event to the modified objects (each of them shown as a vertical line).

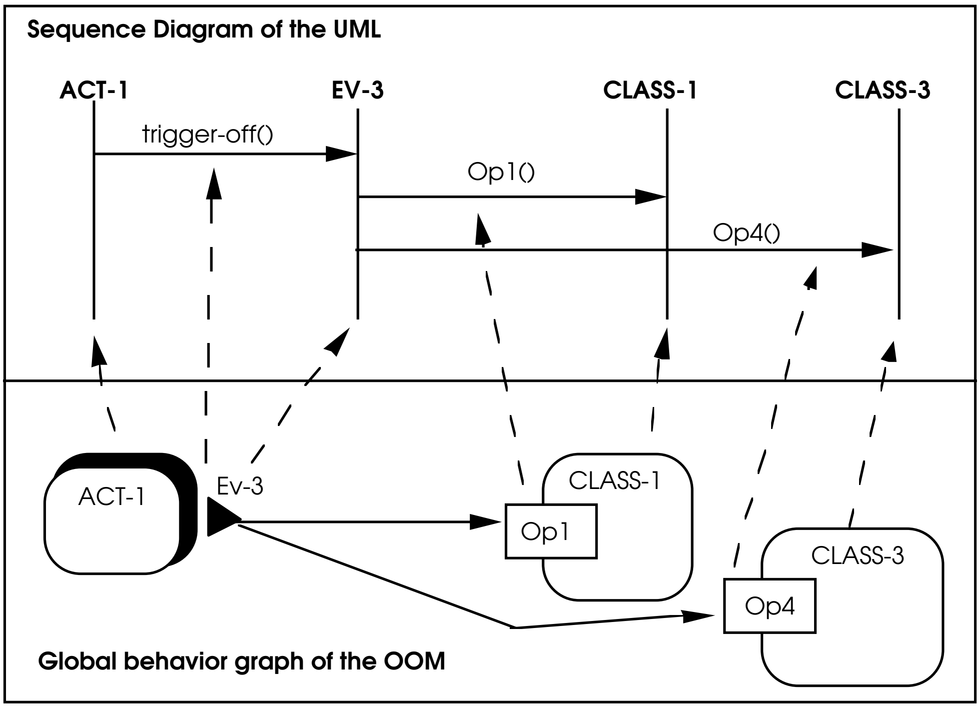


Figure 168: mapping of an external event

5.2.3.2 Temporal event

The way to draw a sequence diagram for a temporal event follows the same process than the one for an external event, except that the first vertical line represents the calendar instead of the actor.

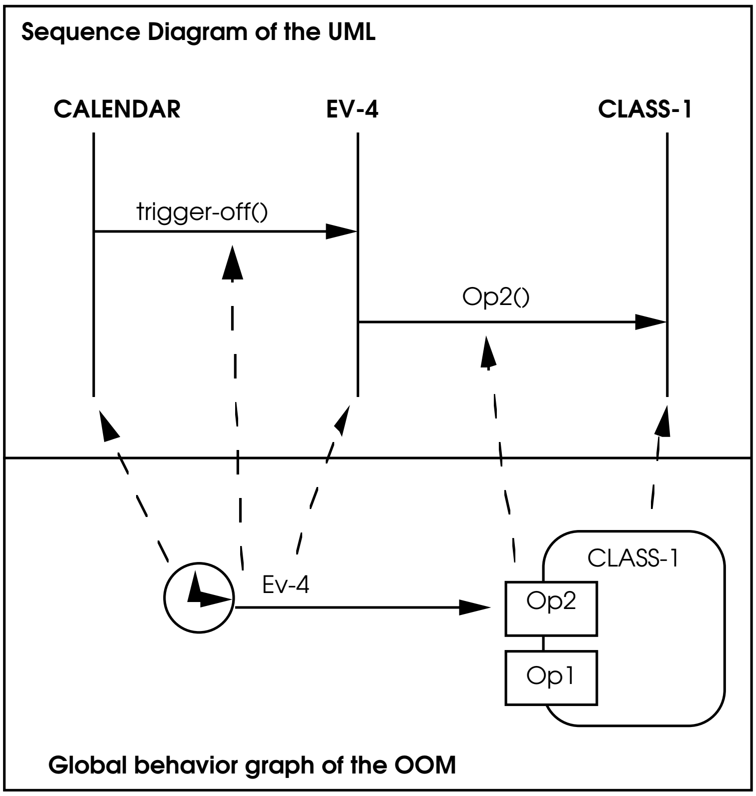


Figure 169: mapping of a temporal event

5.2.3.3 Internal event

The way to draw a sequence diagram for a temporal event follows the same process as for the external and temporal events, except that the first vertical line represents the class instead of the actor or calendar.

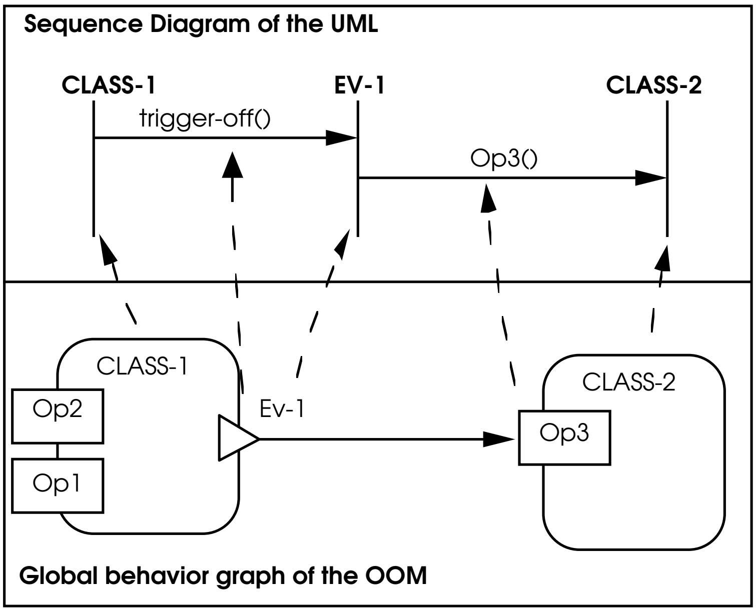


Figure 170: mapping of an internal event

5.2.3.4 Control Condition

It is possible to show the control conditions of the global behavior graph on the sequence diagrams by noting them before the name of the operations and by drawing the operations lines from the same point on the event line.

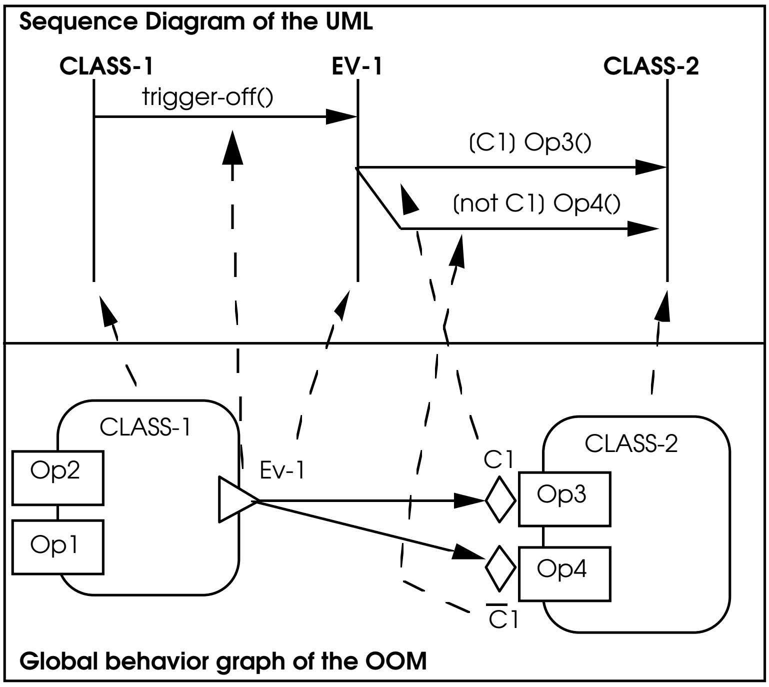


Figure 171: mapping of a condition

5.2.3.5 Summary

Global Behavior Diagram of OOM	**Sequence Diagram of UML**
Actor	Actor on the first vertical line
Calendar	Calendar on the first vertical line
Release object	Object on the first vertical line
Event	Event on the second vertical line *trigger-off()* operation on the horizontal line between the first two vertical lines
Modified object(s)	Object(s) on the third vertical line, on the fourth, on the fifth ...
Operation that modify an object	Operation on the horizontal line between the event and the modified object
Condition before an operation	Condition on the horizontal line before the operation

5.2.4 From the Global Behavior Graph to the Use Case Model

There are two kinds of Use Case Model: the top-level and the low-level. The top-level Use Case Model will show the actors which act on the application and the use cases performed following the external events. The low-level ones will show the same but locally to one object class. These last ones will not be useful for us since the information will be the same as in the Sequence Diagrams. As a consequence, we will only use the top-level Use Case Model.

According to UML, the meaning of a Use Case is often written as an informal text description of the external actors and the sequences of interactions between objects that make up the transaction. We assume that each external and temporal event began a transaction into the application. As a consequence, each Use Case is the sequence of operations triggered by an external or a temporal event and can be named with its name.

The application is represented as a box. The actors and the calendars are placed outside this box and have links with the use cases placed inside.

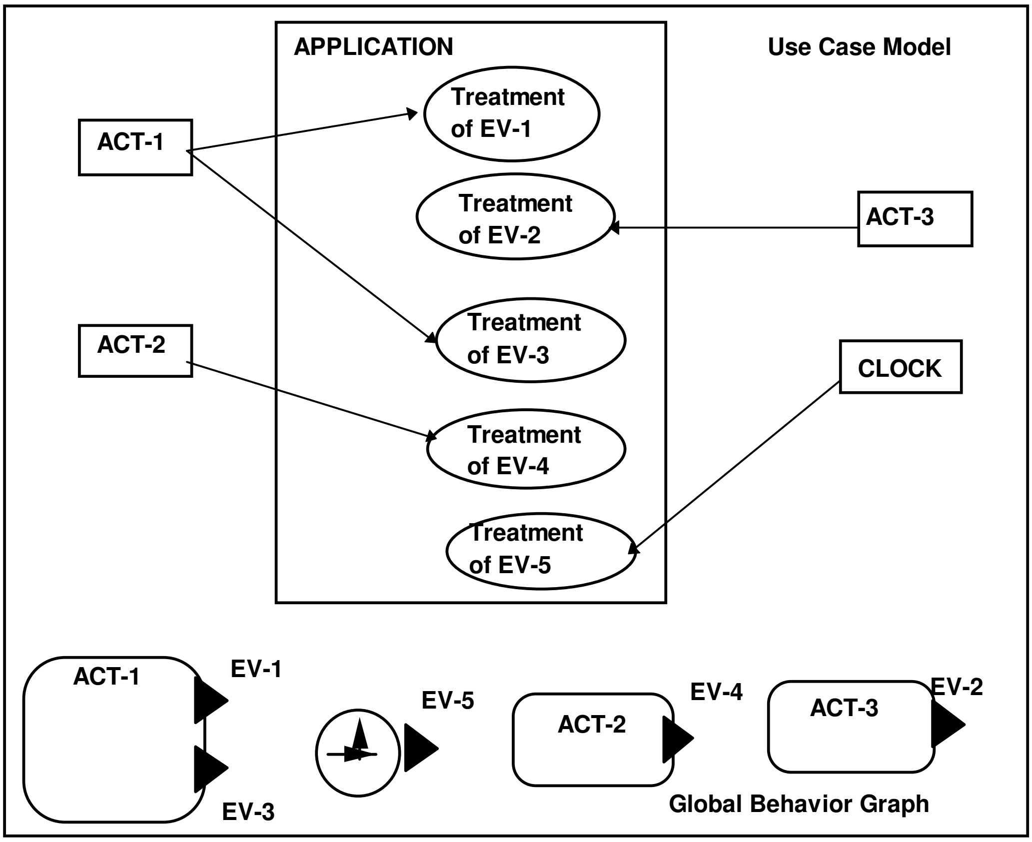


Figure 172: mapping to the Use Case Model

5.2.5 Case study

The library management application is the case study found in chapter 3. This section illustrates the use of the UML compliance rules on the case study « Library management ».

5.2.5.1 Class diagram

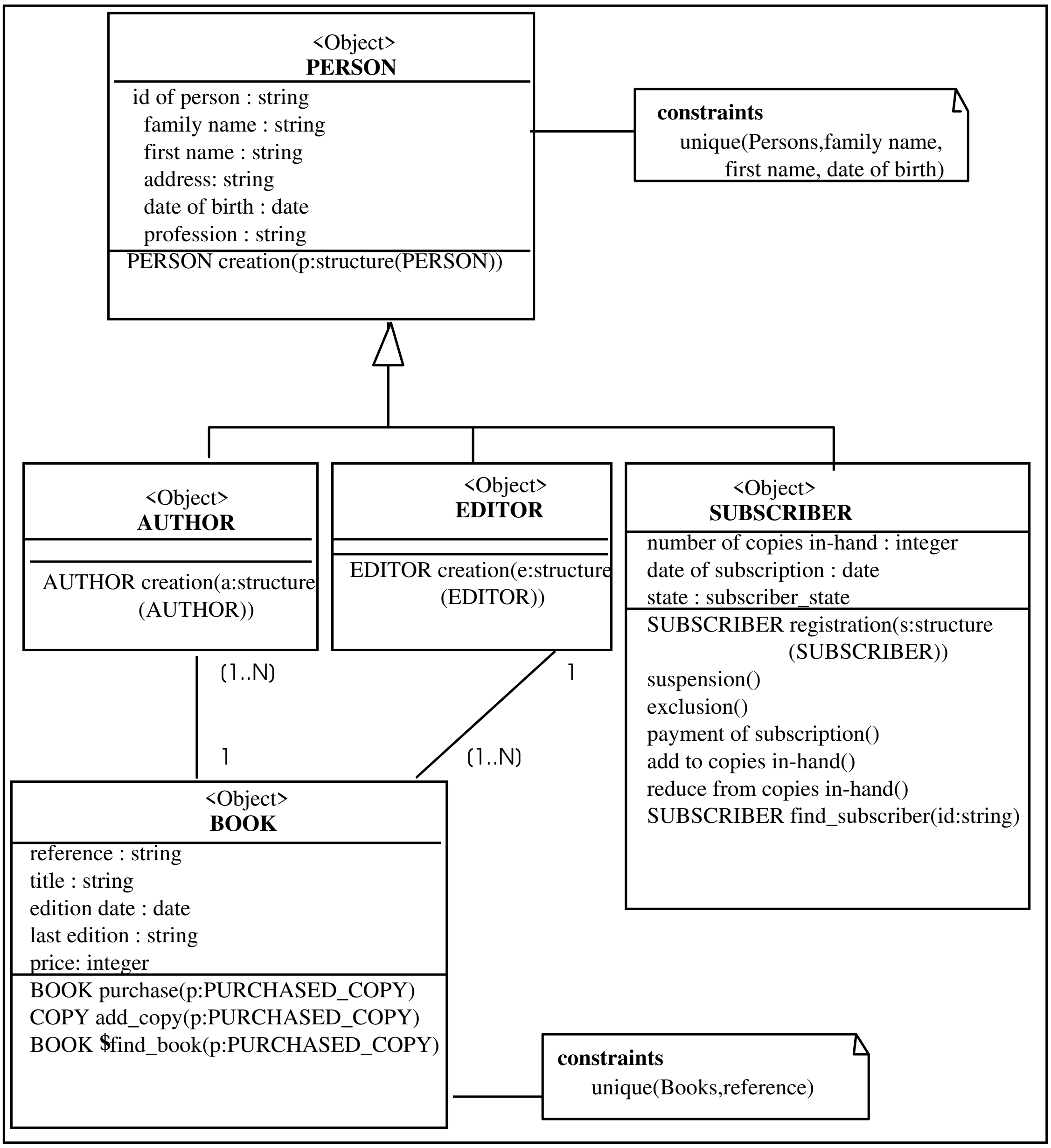


Figure 173: Class diagram (part 1)

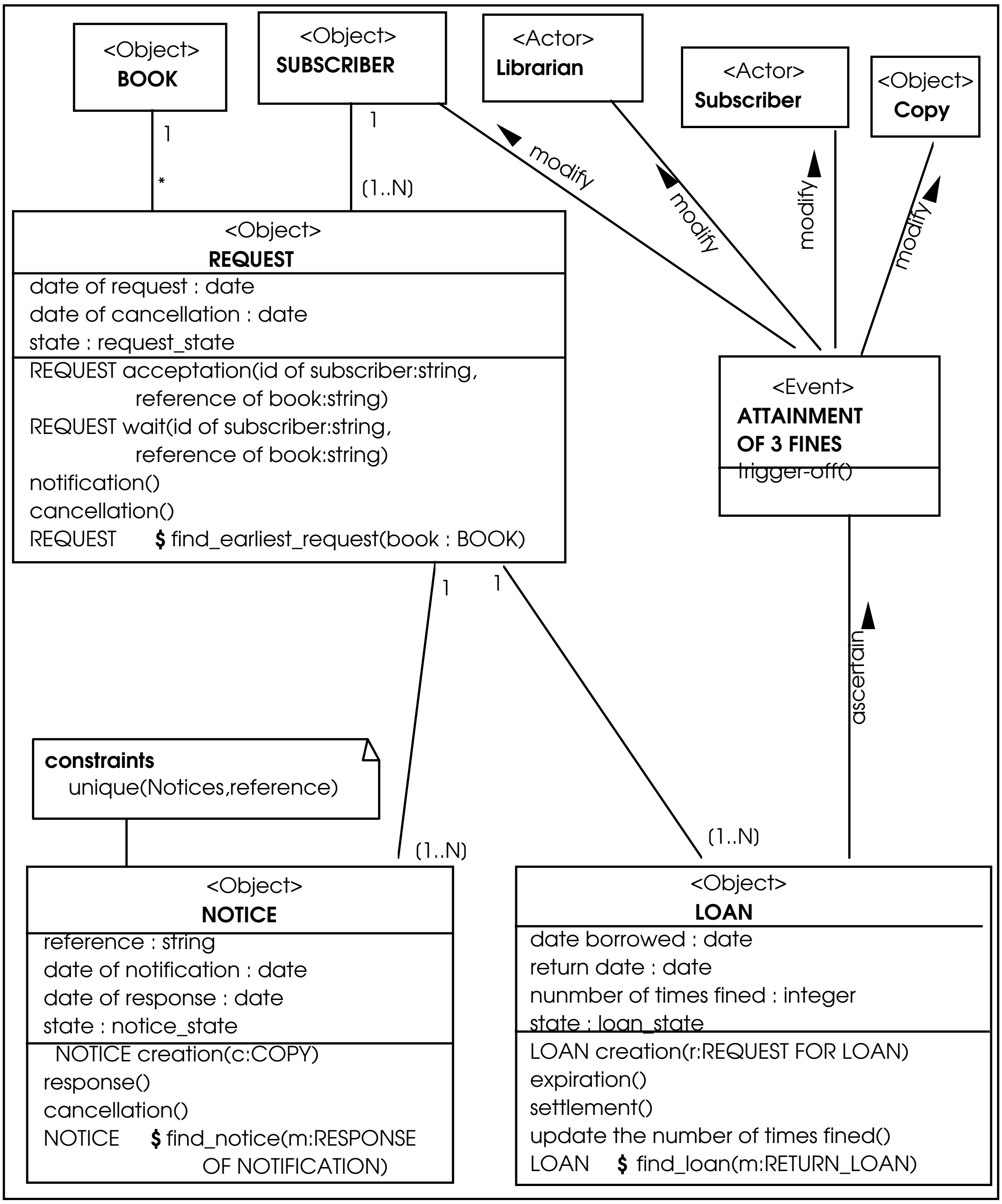


Figure 174: class diagram (part 2)

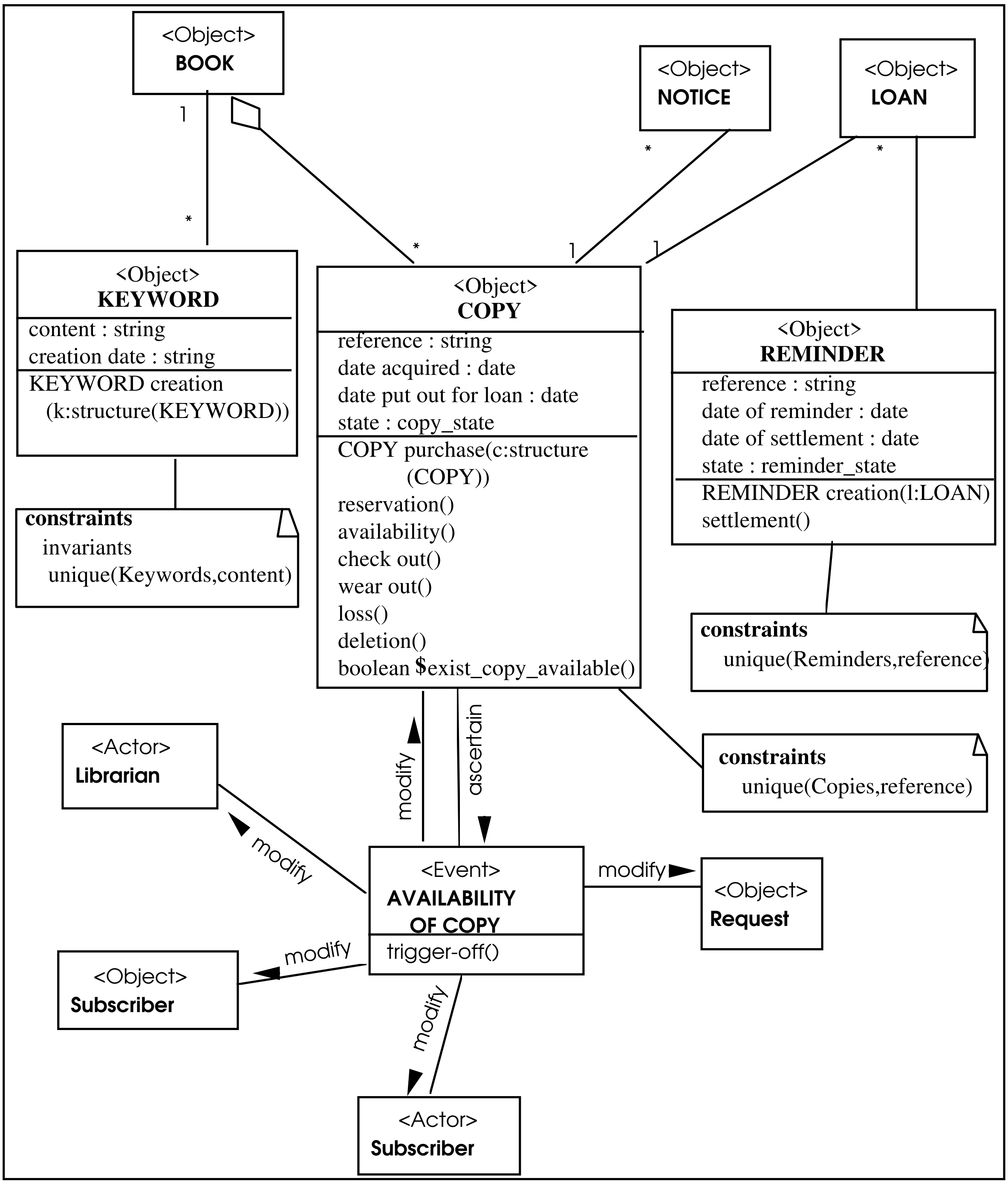


Figure 175: Class diagram (part 3)

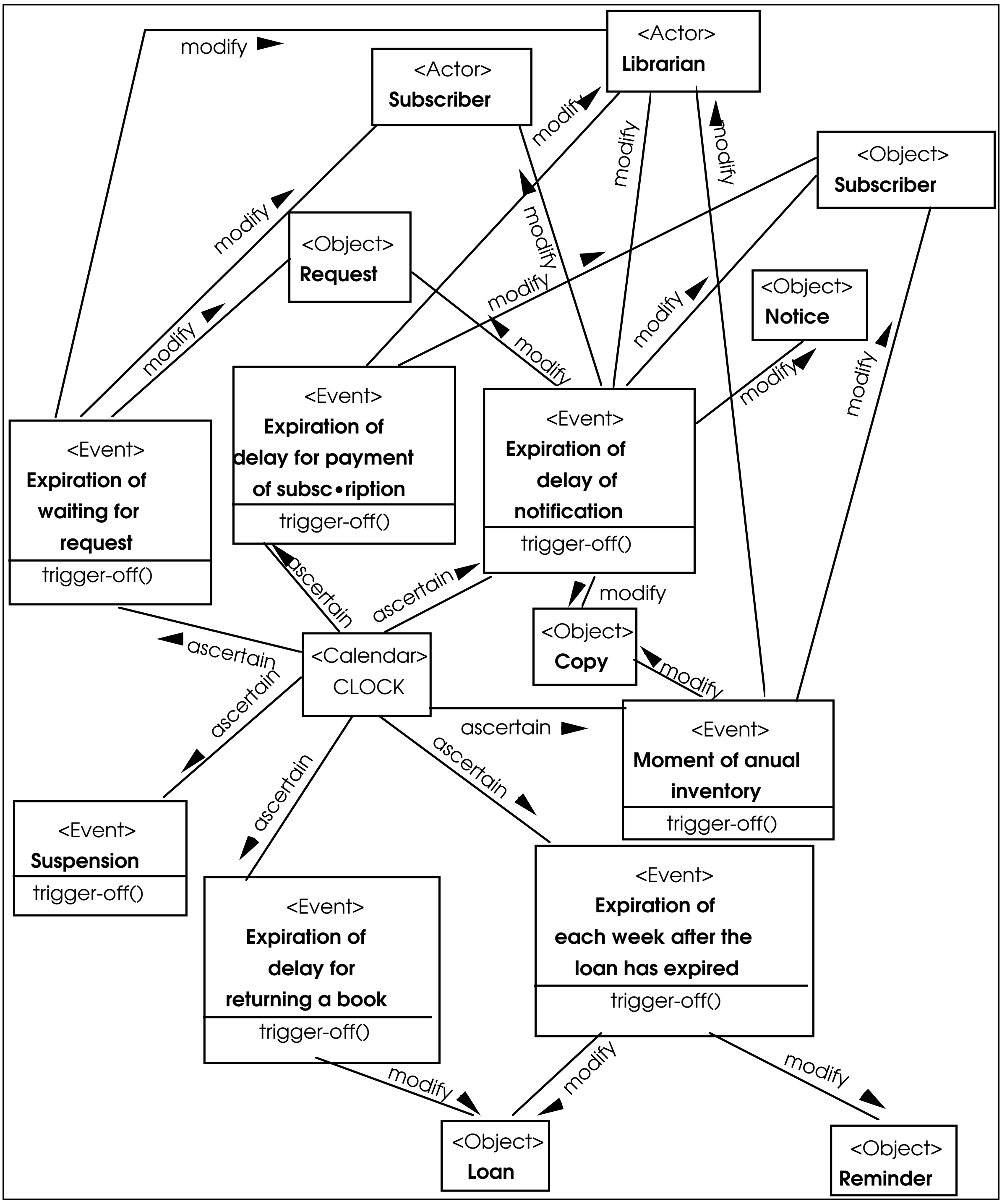


Figure 176: Class diagram (part 4)

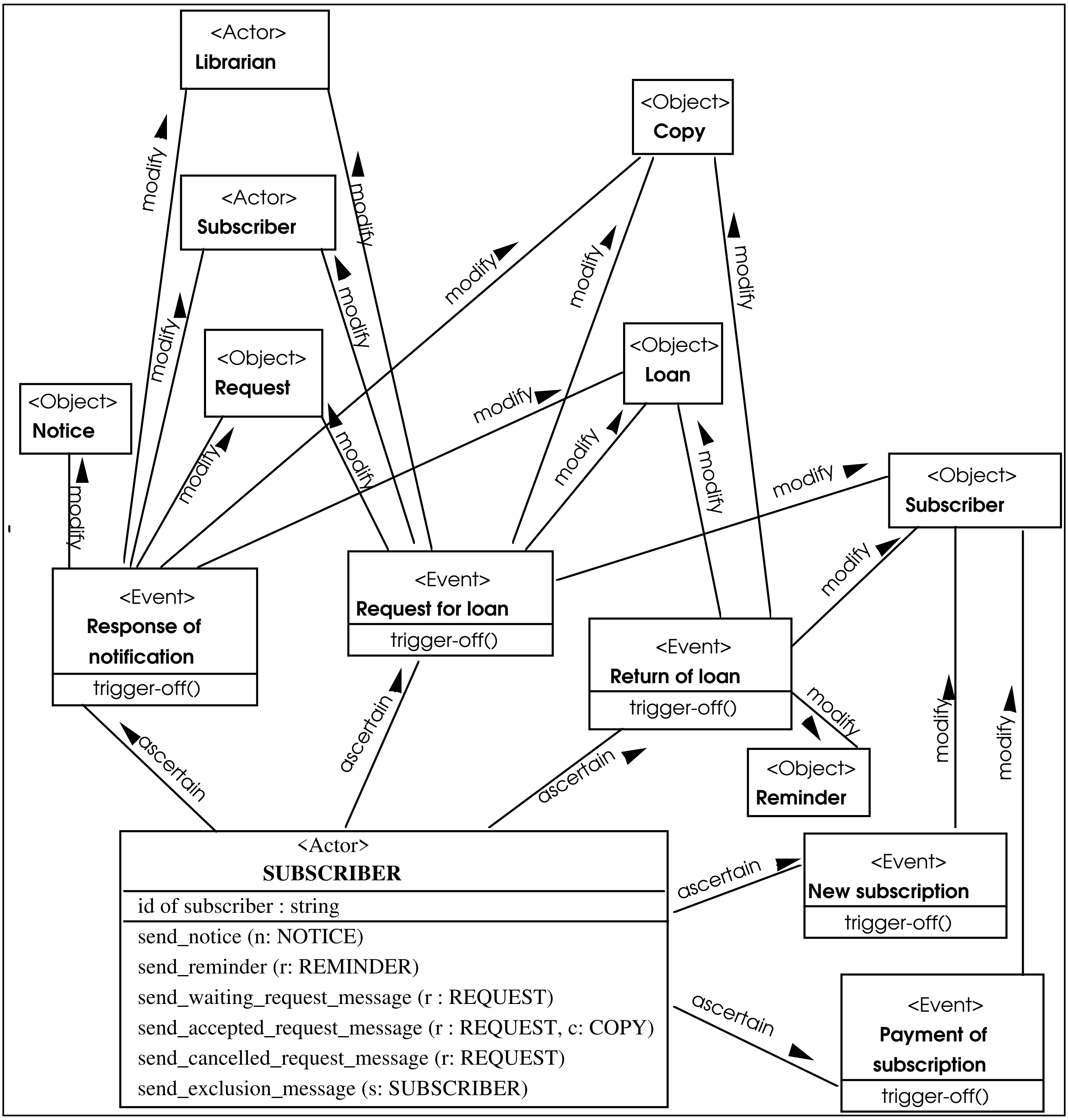


Figure 177: Class diagram (part 5)

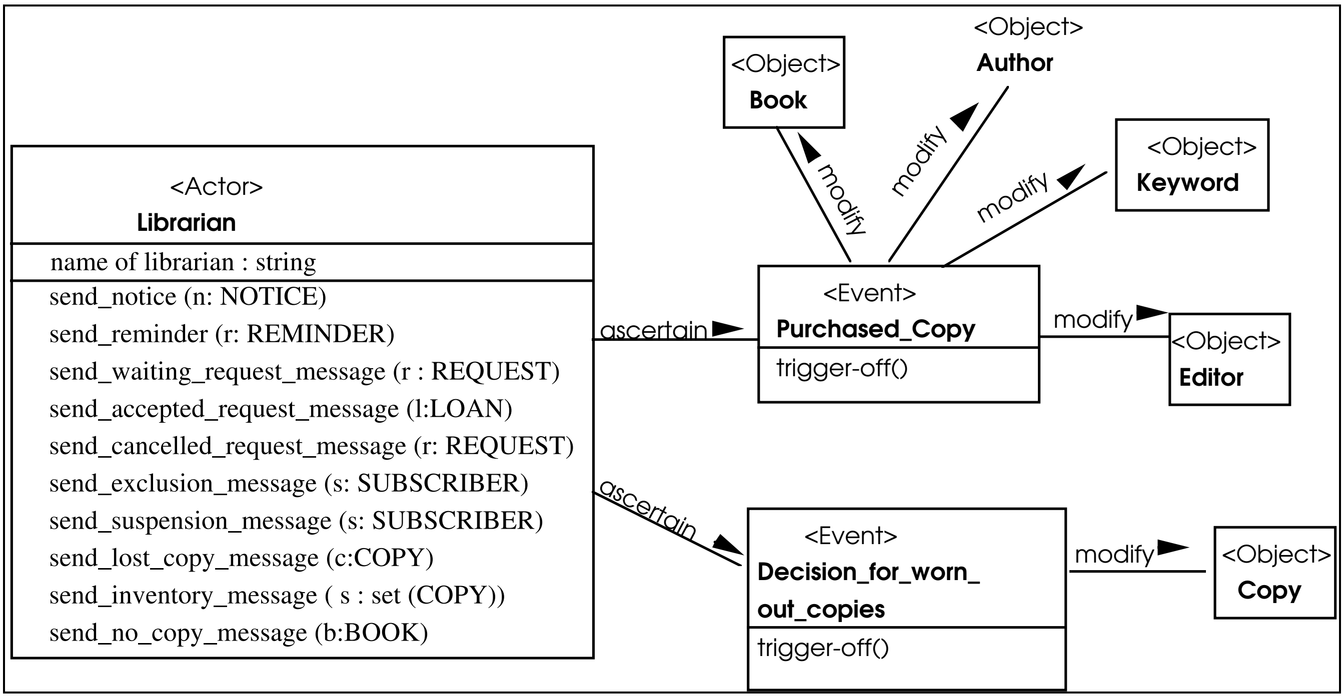


Figure 178: Class diagram (part 6)

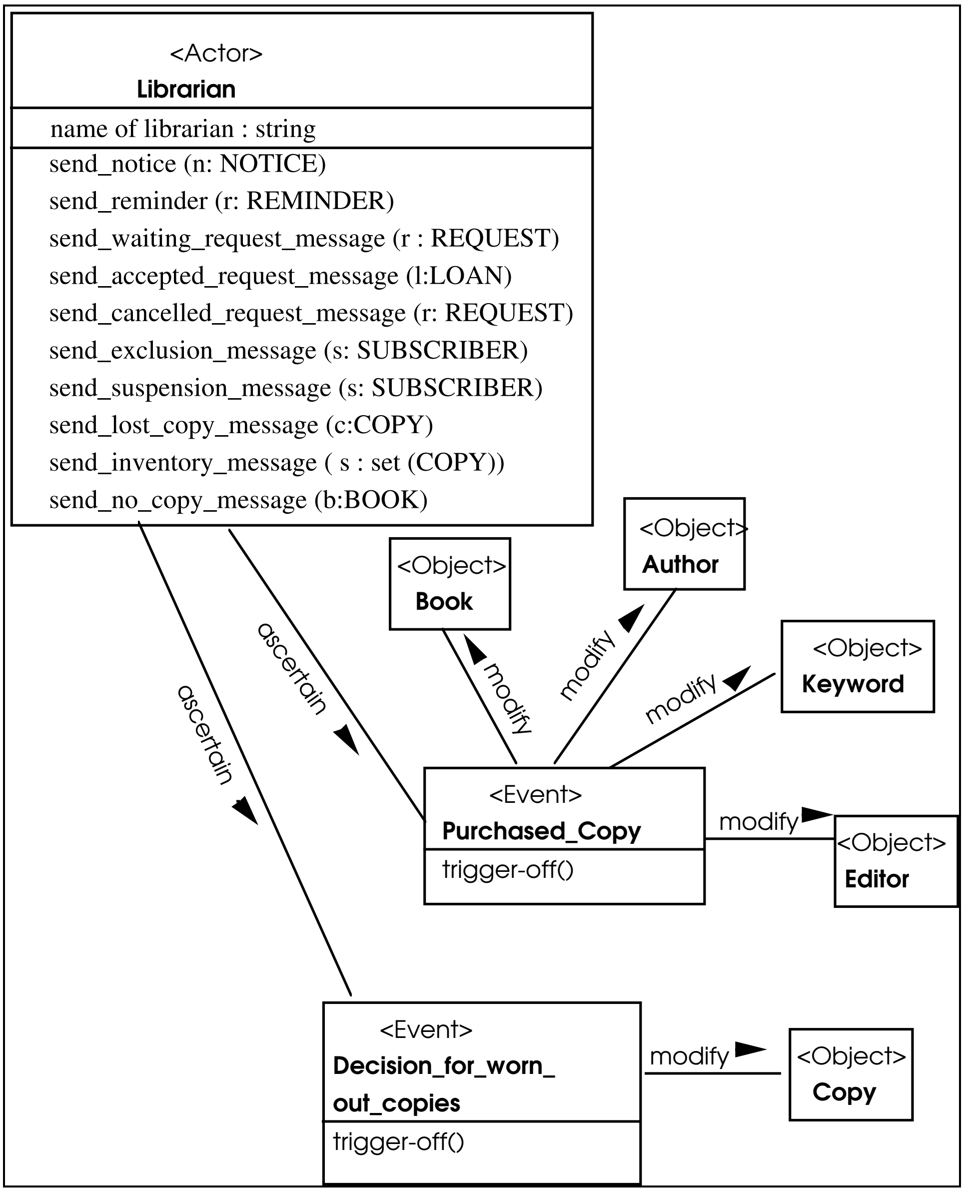


Figure 179: Class diagram (part 7)

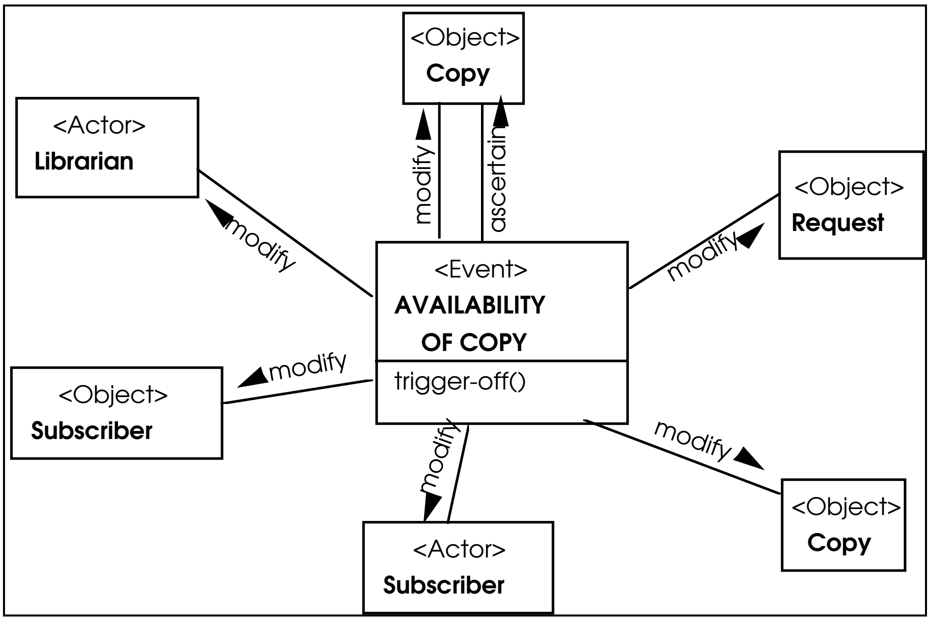


Figure 180: Class diagram, part 8

5.2.5.2 Sequences diagrams

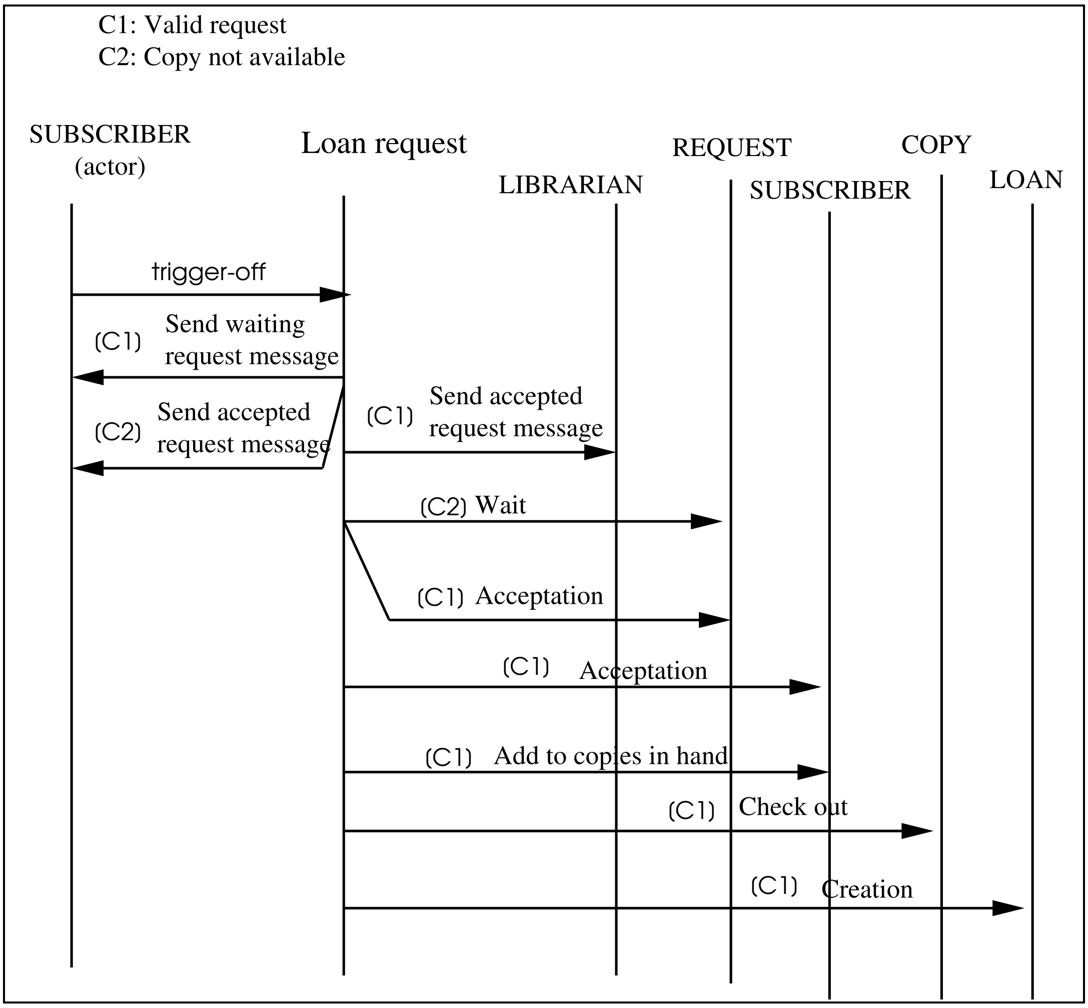


Figure 181: Sequence diagram of the *loan request* event

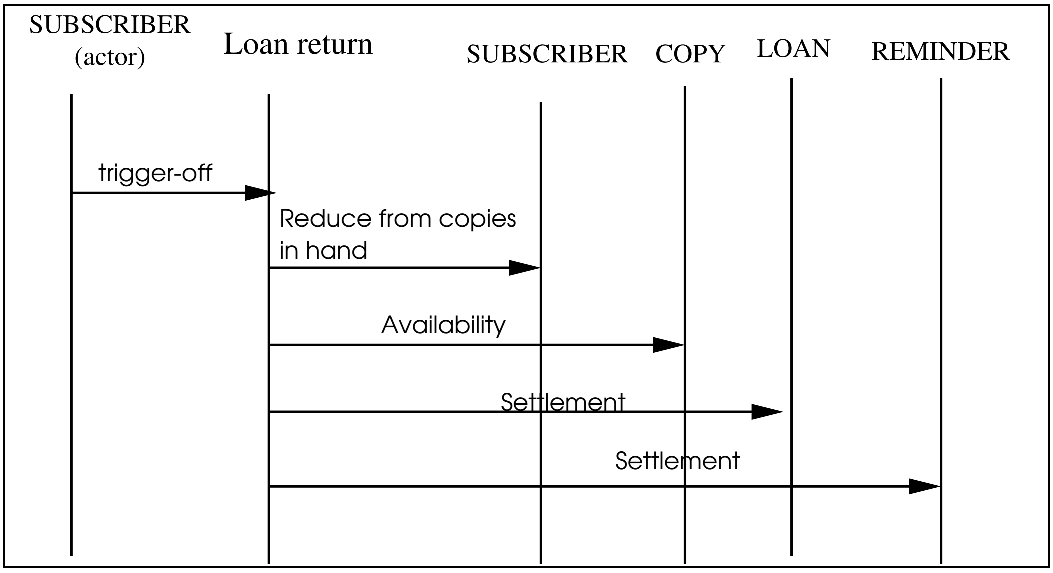


Figure 182: Sequence diagram of the *loan return* event

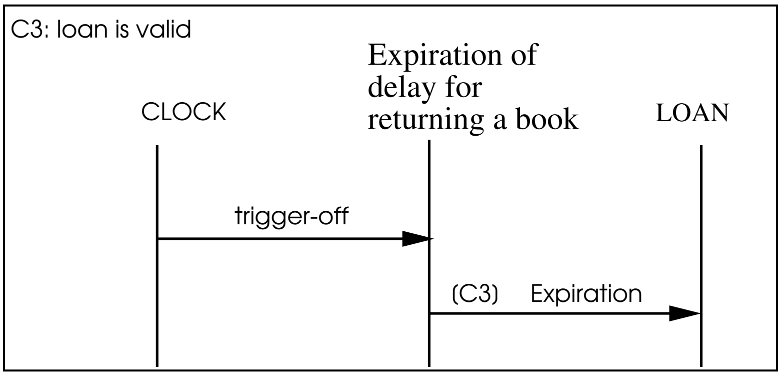


Figure 183: Sequence diagram of the *expiration of delay for returning a book* event

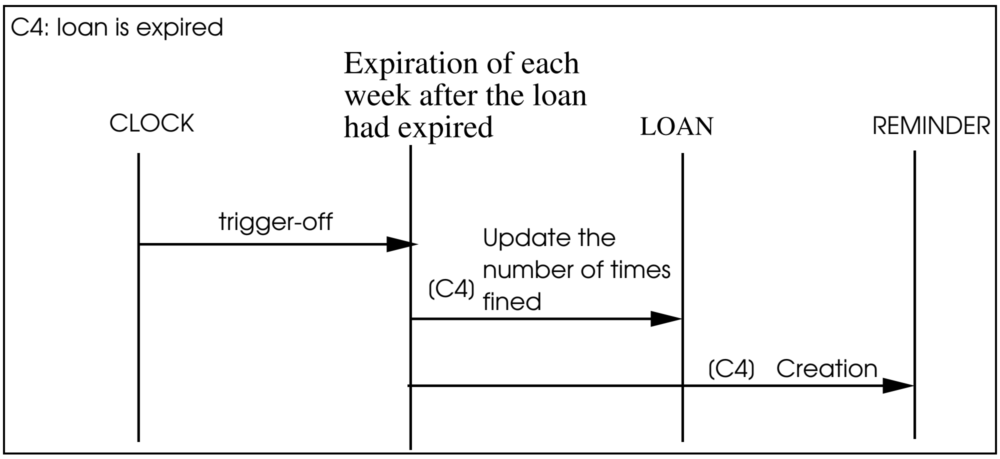


Figure 184: Sequence diagram of the *expiration of each week after the loan had expired* event

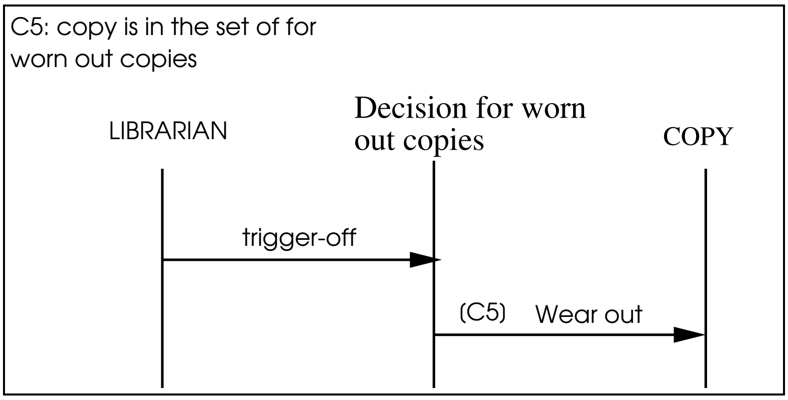


Figure 185: Sequence diagram of the *decision for worn out copies* event

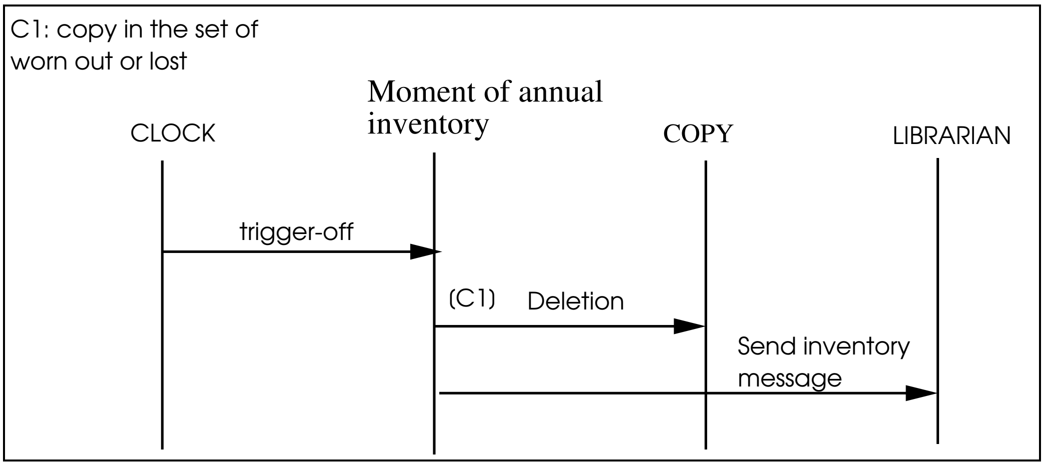


Figure 186: Sequence diagram of the *moment of annual inventory* event

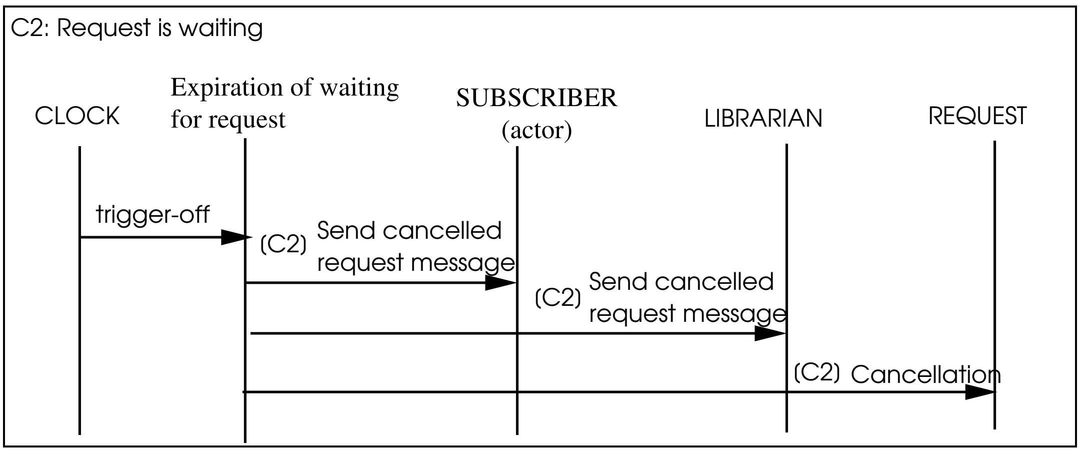


Figure 187: Sequence diagram of the *expiration of waiting for request* event

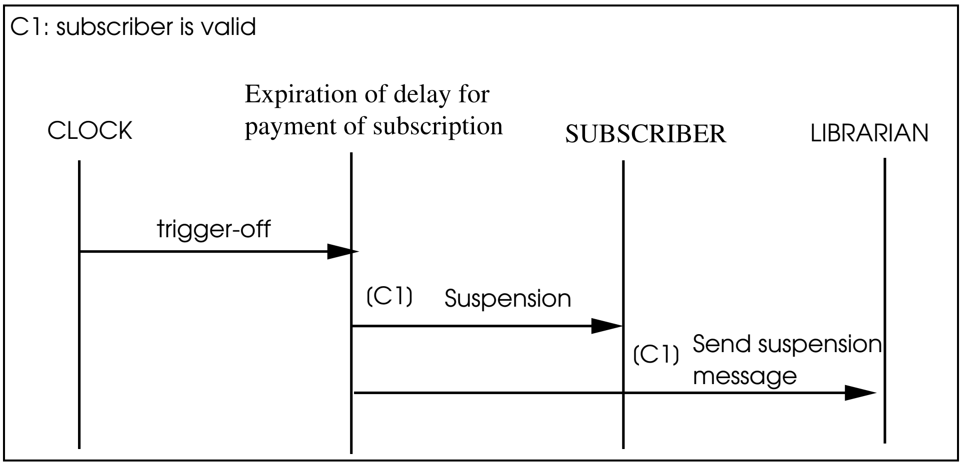


Figure 188: Sequence diagram of the *expiration of delay for payment of subscription* event

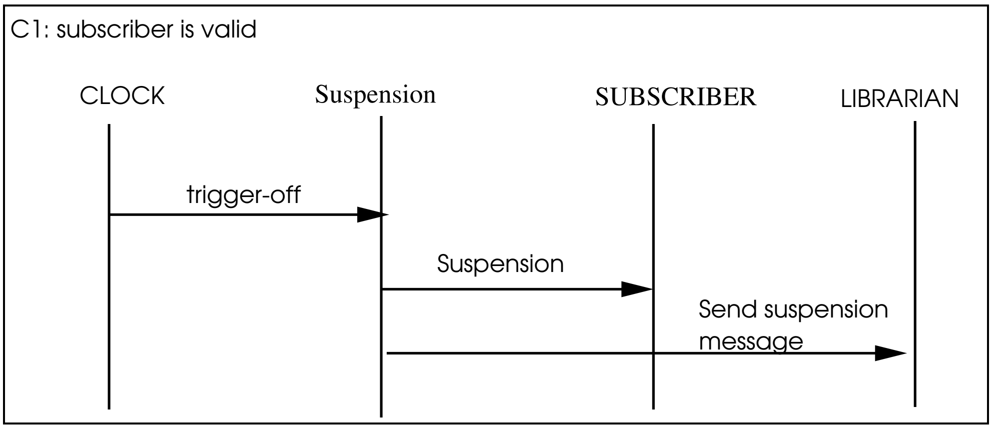


Figure 189: Sequence diagram of the *suspension* event

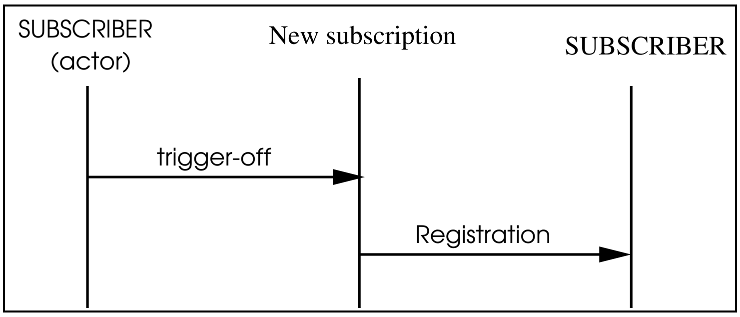


Figure 190: Sequence diagram of the *new subscription* event

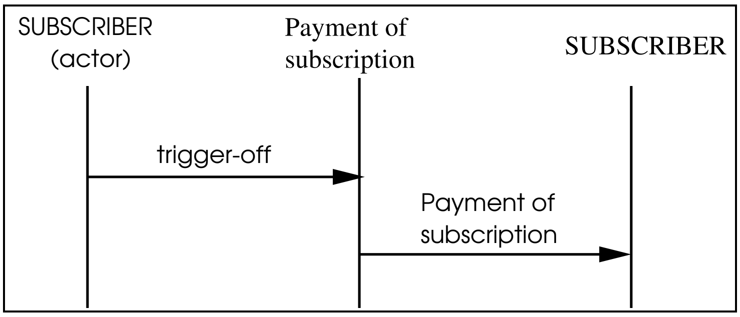


Figure 191: Sequence diagram of the *payment of subscription* event

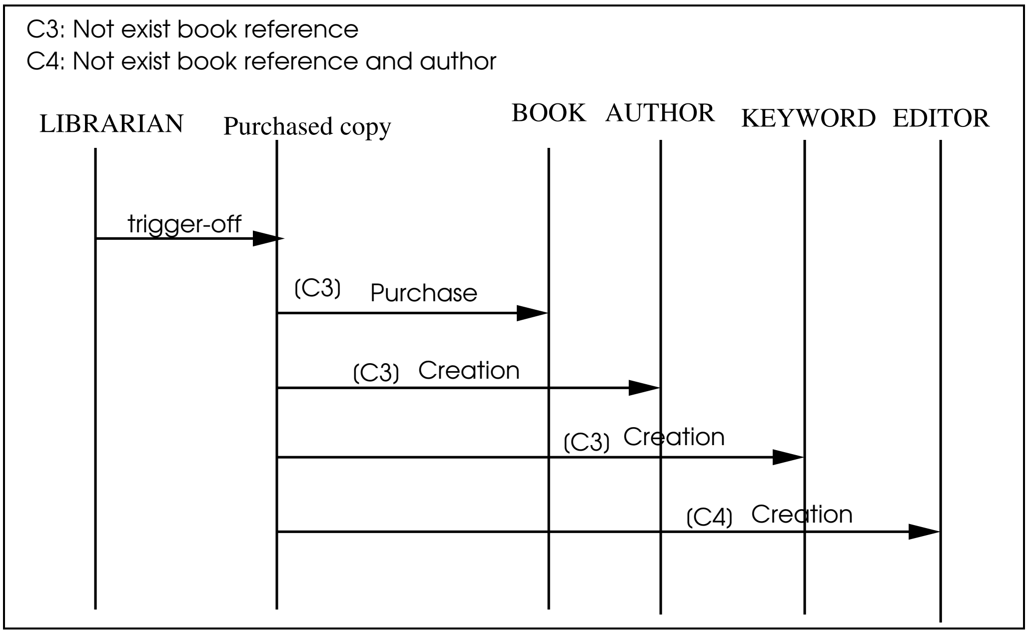


Figure 192: Sequence diagram of the *purchased copy* event

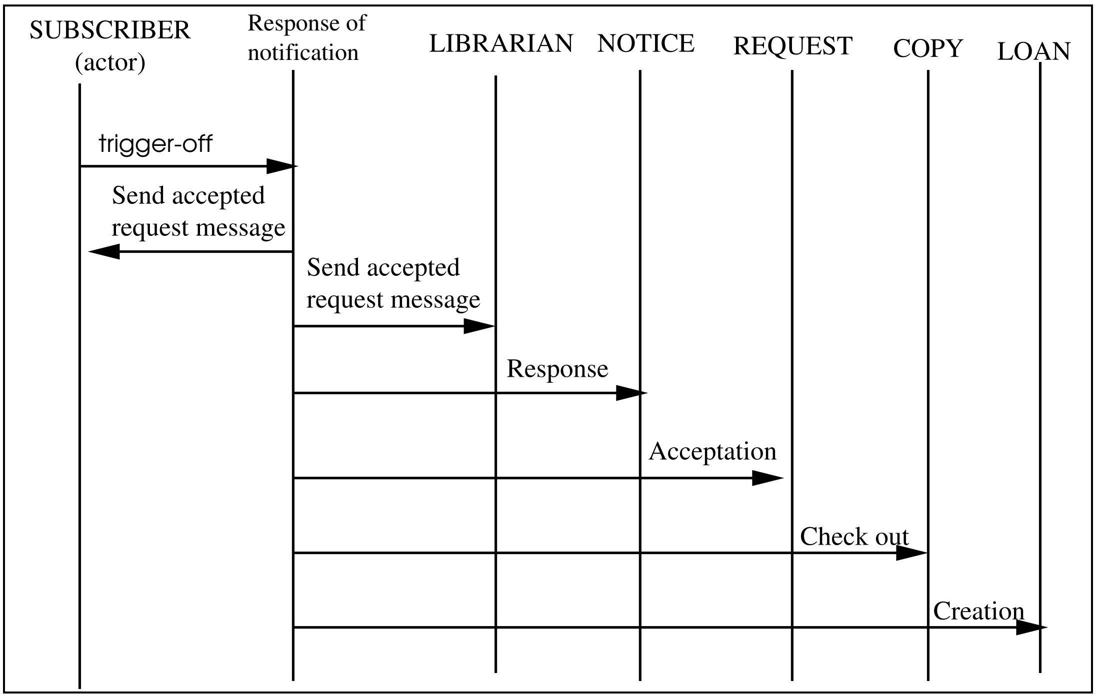


Figure 193: Sequence diagram of the *response of notification* event

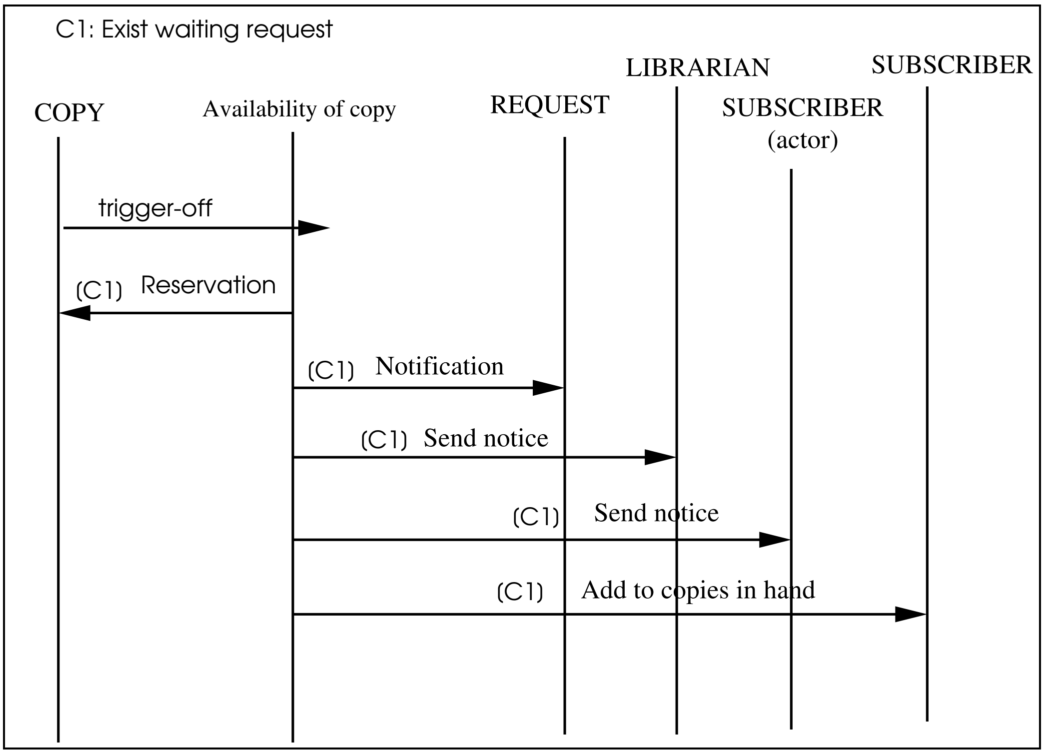


Figure 194: Sequence diagram of the *availability of copy* event

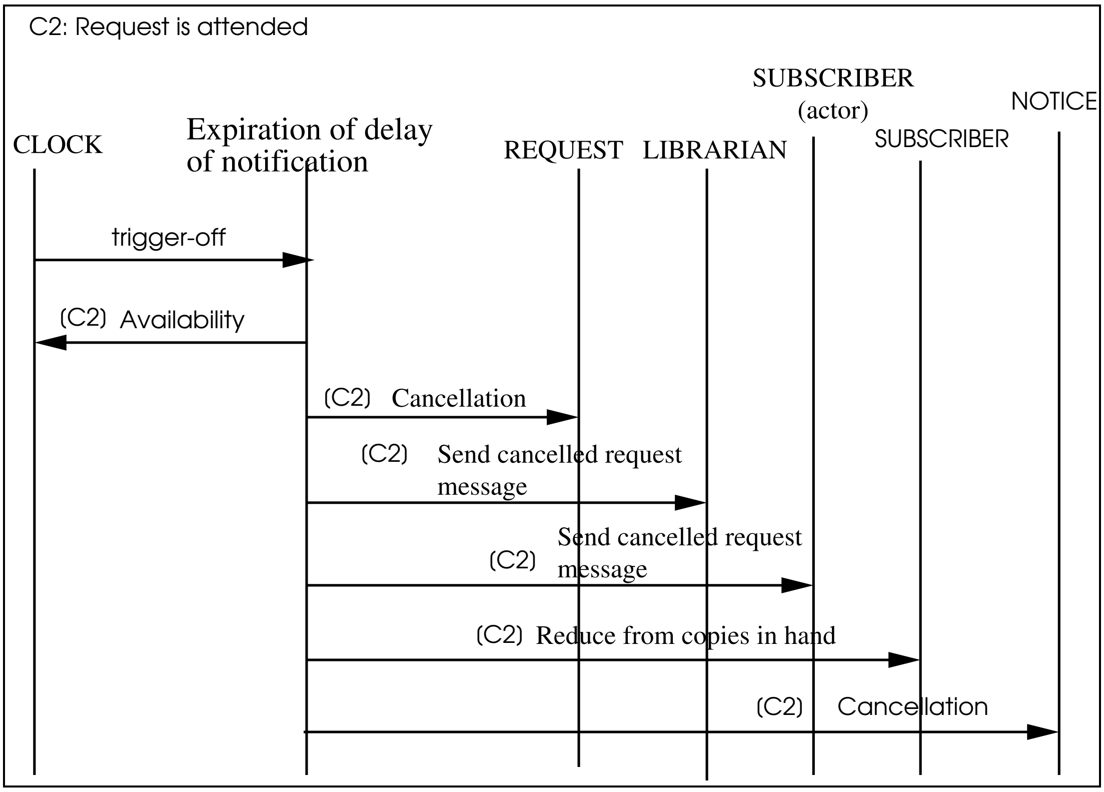


Figure 195: Sequence diagram of the *expiration of delay of notification* event

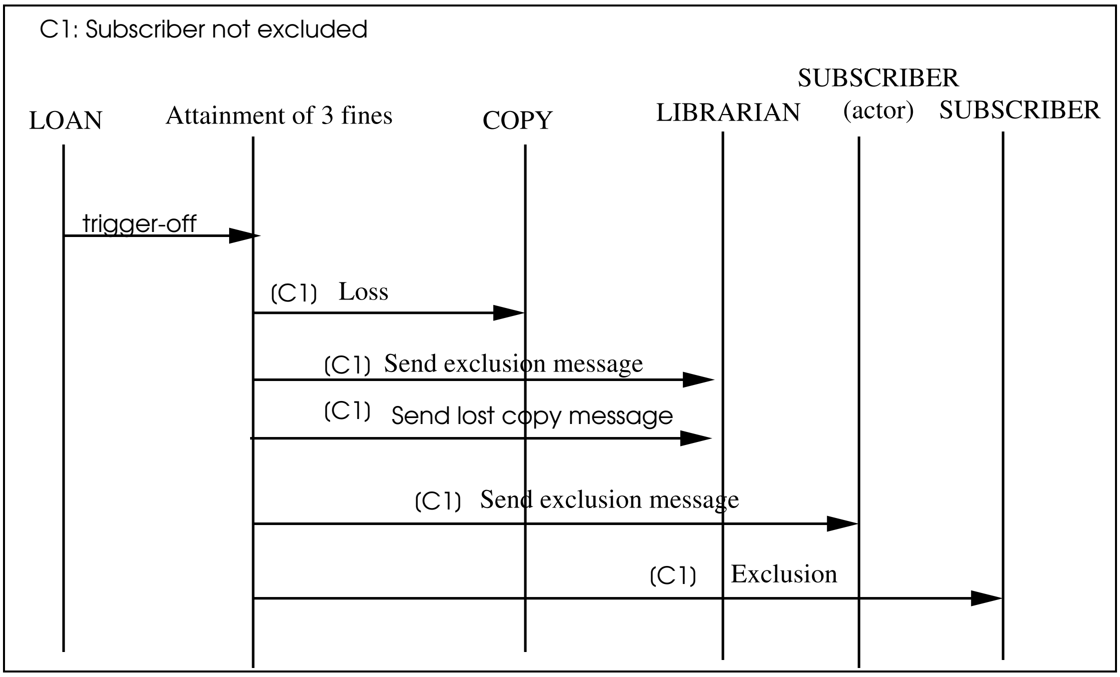


Figure 196: Sequence diagram of the *attainment of 3 fines* event

5.2.5.3 Use case model

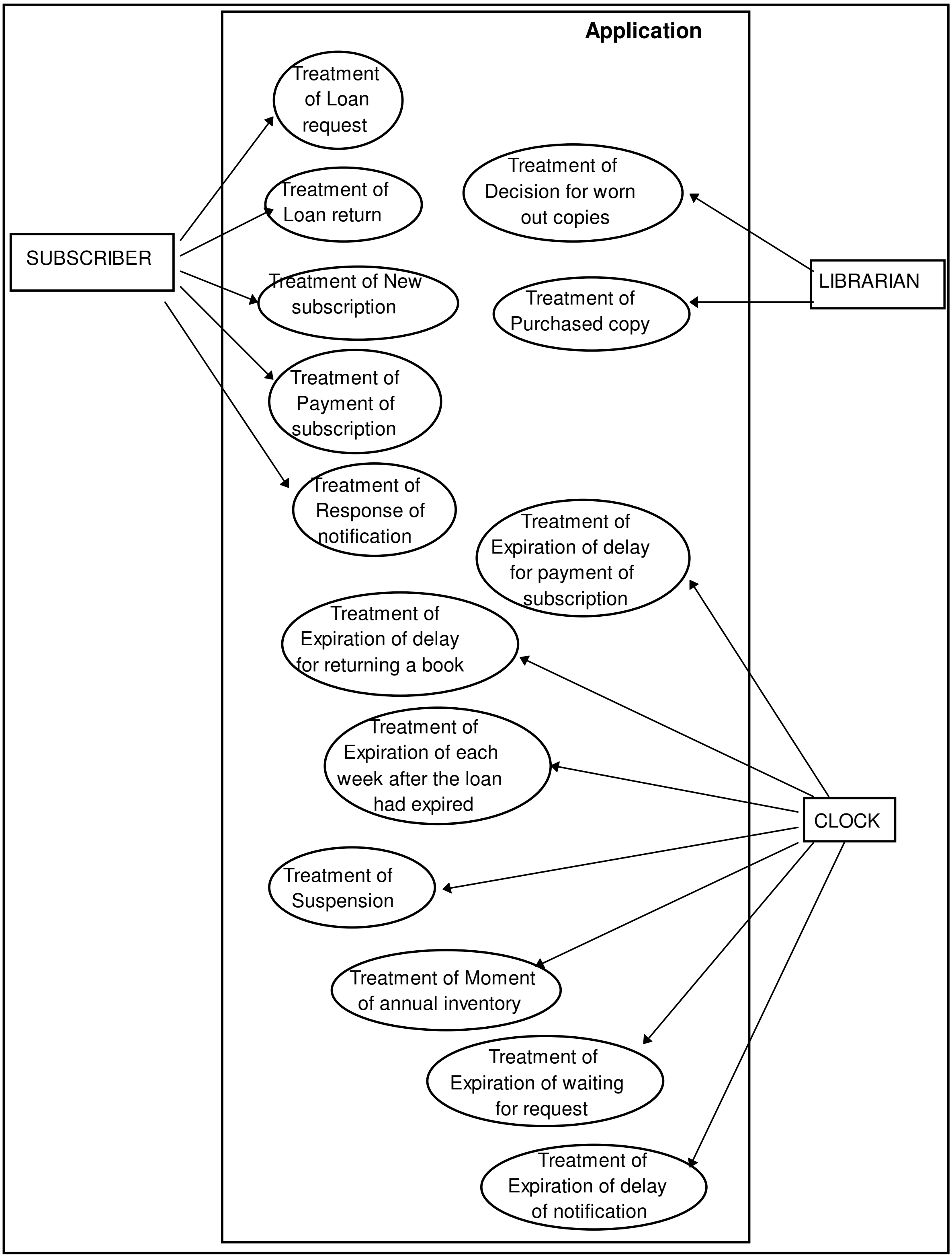


Figure 197: Use case model

5.3 Mapping from TOOM to UML

5.3.1 From the Global Structure Graph to the Class Diagram

5.3.1.1 Snapshot Basic Class

The concept of class in OOM corresponds to the snapshot basic class of TOOM. It is composed of properties, operations, queries, constraints and internal-events. As for the mapping from OOM, we choose to use the concept of *stereotype* explained in the 0.91 addendum of the UML in order to have the concept of event in the class diagram.

A *class* in the TOOM will be represented in the UML as a class with the stereotype *<basic class object>*.

The *properties* and *operations* of the class will be the same in the two models.

The queries are just an other kind of operations, so they will be represented in the set of operations of the class.

The constraints can't be represented in the class, so we will use the concept of *note* to represent them in the diagram.

The internal events are represented in the UML compliance as a class with the stereotype *<internal event>*. This particular class will automatically have the operation *trigger-off()* in order to have the possibility to set off the event when it is necessary. As a consequence, there is a link between the object class and the event with the association *ascertain*. As for the constraints, the event's predicate is represented graphically with the concept of *note*.

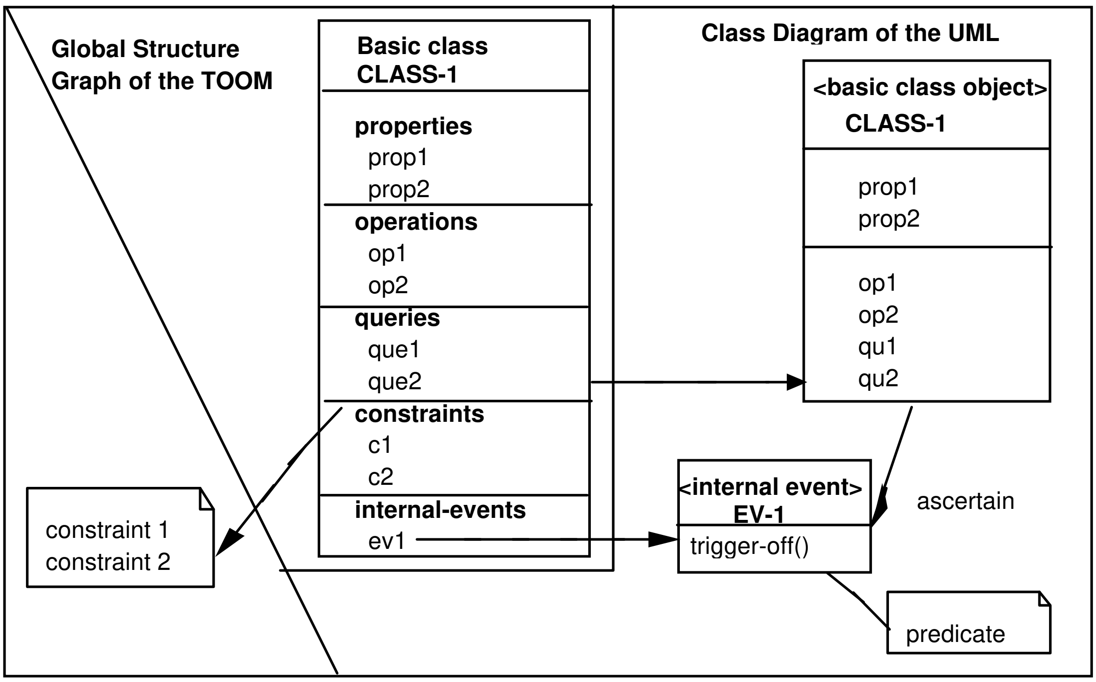


Figure 198: mapping of a basic object class

5.3.1.2 Snapshot Derived class

The concept of derived class doesn't exist in UML. As for the mapping of a basic class, we choose to use the concept of *stereotype*.

A *derived class* in the TOOM will be represented in the UML as a class with the stereotype *<derived class object>*.

The *properties* will be the same in the two models.

The *derivation function* of the TOOM derived class will be placed in the operations of the UML class.

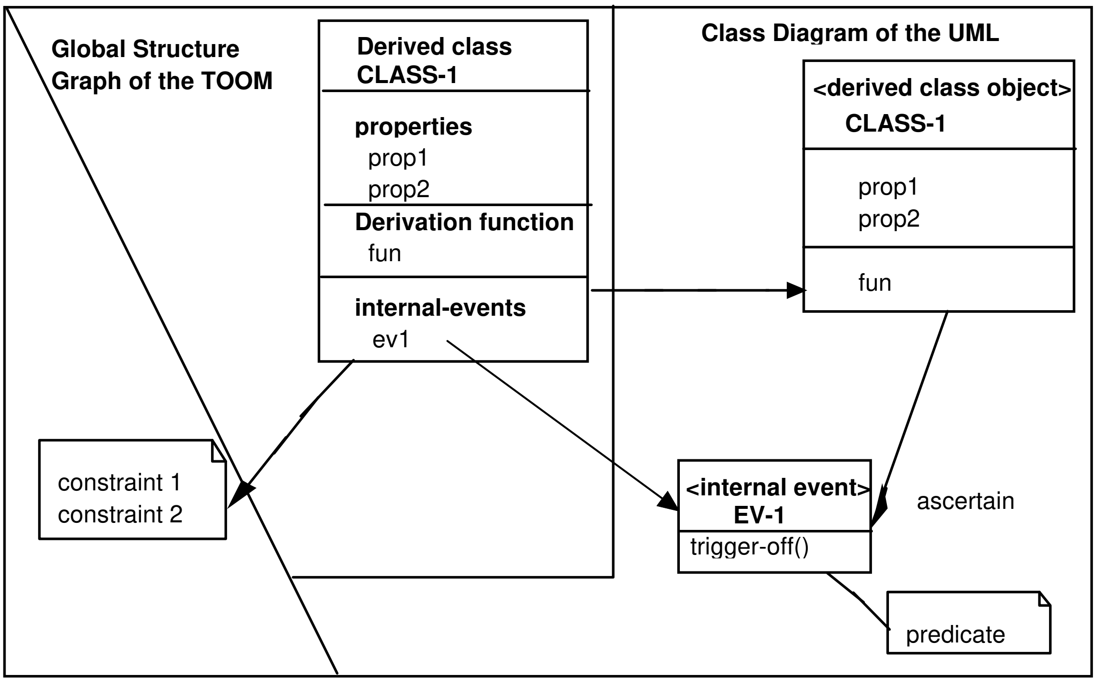


Figure 199: mapping of a derived object class

5.3.1.3 Temporal basic class

A TOOM temporal basic class may have six different headers : state<VT>, State<TT>, History<VT>, History<TT> and History<VT,TT>. A note is added on the class in order to have the distinction in UML.

The mapping of the contents of the TOOM class in UML class follows the same principles than for the snapshot classes, as shown in Section 5.3.1.1.

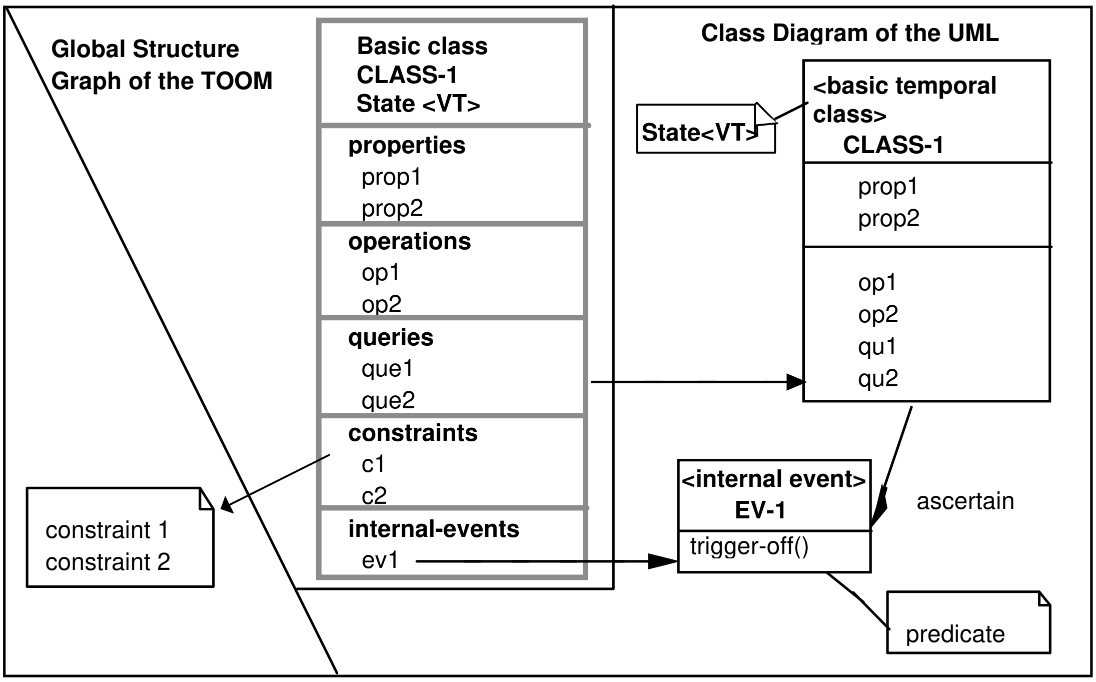


Figure 200: mapping of a temporal basic class

5.3.1.4 Temporal derived class

The mapping from a temporal derived class follows the same principles than the mapping from a snapshot derived class. The *properties* will be the same in the two models and the *derivation function* of the TOOM derived class will be placed in the operations of the UML class.

A TOOM temporal derived class, as the temporal basic classes, may have six different headers. On the same way a note is used to specify it in UML.

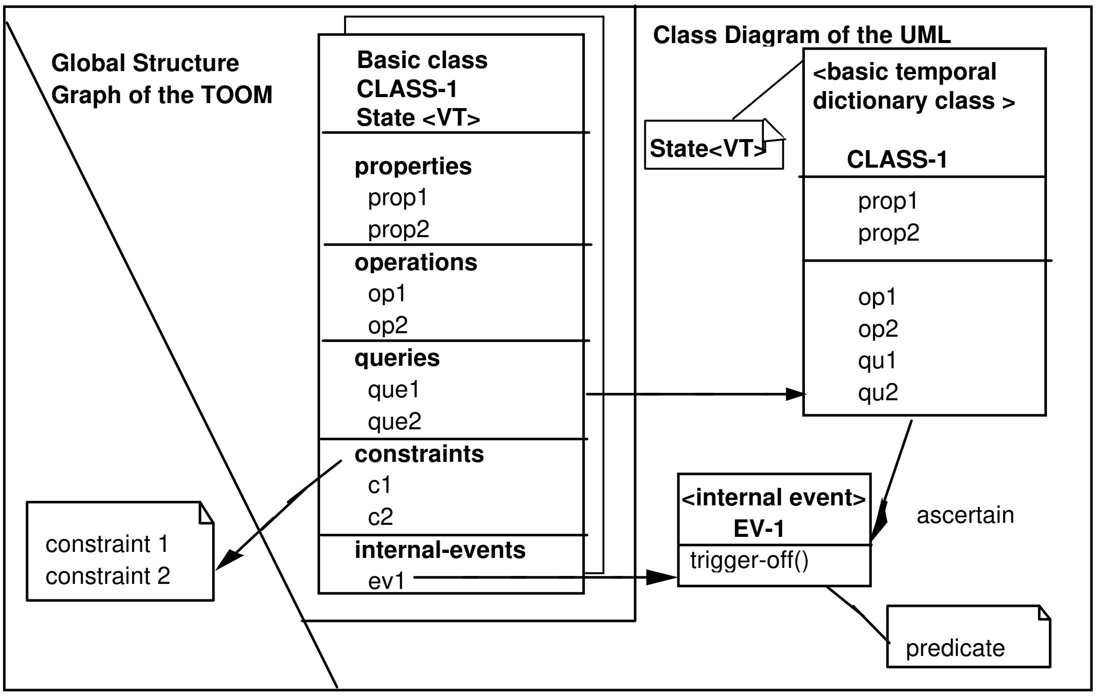


Figure 201: mapping of a temporal derived class

5.3.1.5 Temporal dictionary basic class

As the temporal classes, a TOOM temporal dictionary basic class may have different headings : State<TT> and History <TT>. These distinctions are specified in a UML note, linked to the class.

The mapping of the contents of the TOOM class in UML class follows the same principles than for the snapshot or temporal basic classes, as shown in Section 5.3.1.1.

5.3.1.6 Temporal dictionary derived class

The mapping from a temporal dictionary derived class follows the same principles than the mapping from a snapshot or temporal derived class. The *properties* will be the same in the two models and the *derivation function* of the TOOM derived class will be placed in the operations of the UML class.

A TOOM temporal dictionary derived class, as the temporal dictionary basic classes, may have two different headers that are specified in a UML note linked to the class.

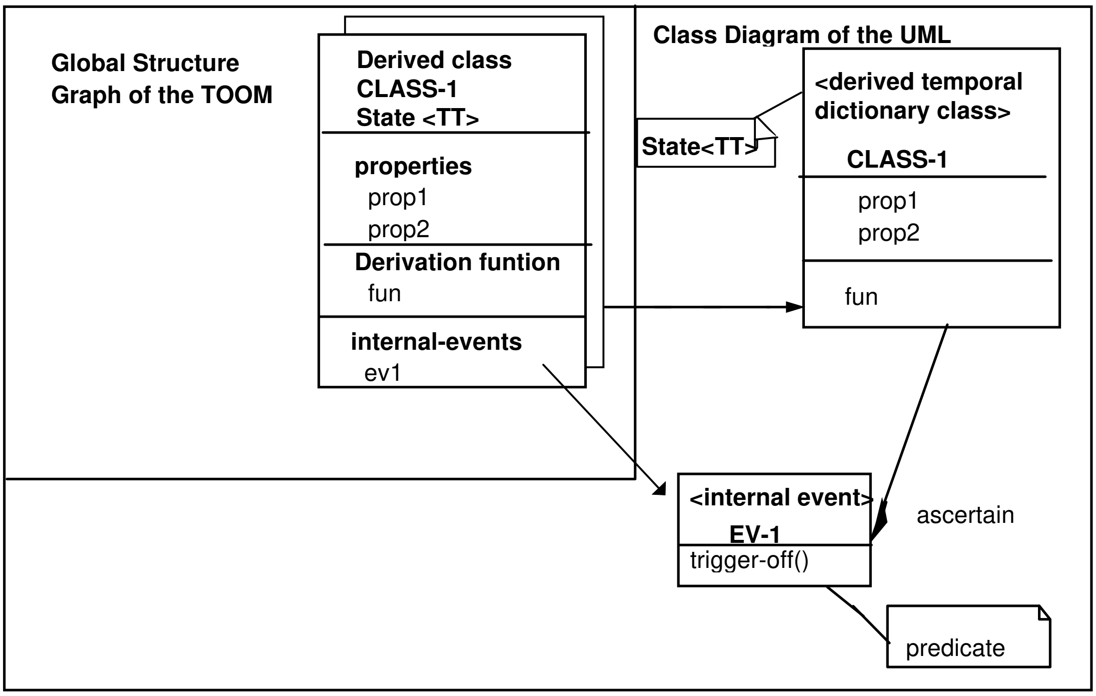


Figure 202: mapping of a temporal dictionary derived class

5.3.1.7 Actor

The concept of actor is the same in OOM and TOOM. As a consequence, the mapping to UML is the same as in the Section 5.2.1.2.

5.3.1.8 Calendar

The representation of the calendar class follow the same principles as the object and actor classes. A new stereotype <calendar> will be used to allow its notation in the class diagram.

The origin of the TOOM calendar class is represented as an attribute of the UML class. The mapping of the set of granules and of external formats follows the same principle.

The three operations of the TOOM calendar class are mapped as operations in the UML class.

The temporal events will be extracted as the other events with the stereotype <event>, linked at the calendar class with the association ascertain and with a note explaining the predicate.

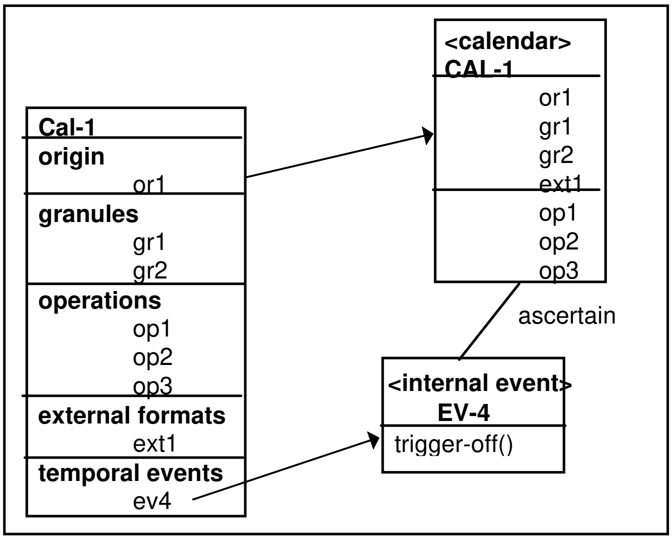


Figure 203: mapping of a calendar class

5.3.1.9 Links

There was no difficulties in the mapping of the structural links of OOM since all the links concepts exist in the notation of UML. However, some links of the TOOM does not exist in UML, it is the case for the temporal aggregation for example.

5.3.1.10 Association

The association concept is the same than for the mapping from OOM, as explained in Section 5.2.1.5.

5.3.1.11 State association

The state association is a particular concept of TOOM. A new stereotype is introduced in order to add it in the UML : *<state association>*. This class is linked to the two classes.

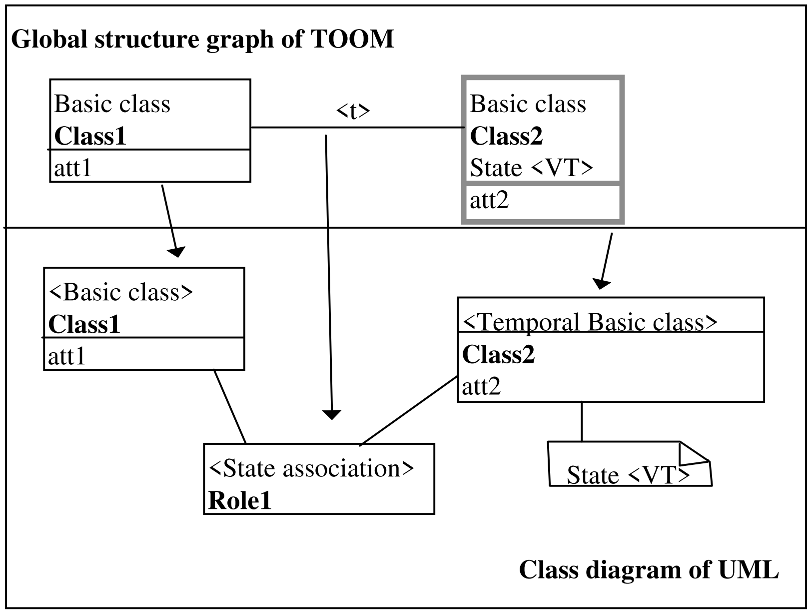


Figure 204: mapping of a state association

5.3.1.12 Aggregation

The aggregation concept is the same than for the mapping from OOM, as explained in Section.

5.3.1.13 Temporal aggregation

The temporal aggregation is a particular concept of TOOM. A new stereotype is introduced in order to add it in the UML : *<temporal aggregation>*. This new type of class is linked to the snapshot class and to the temporal class.

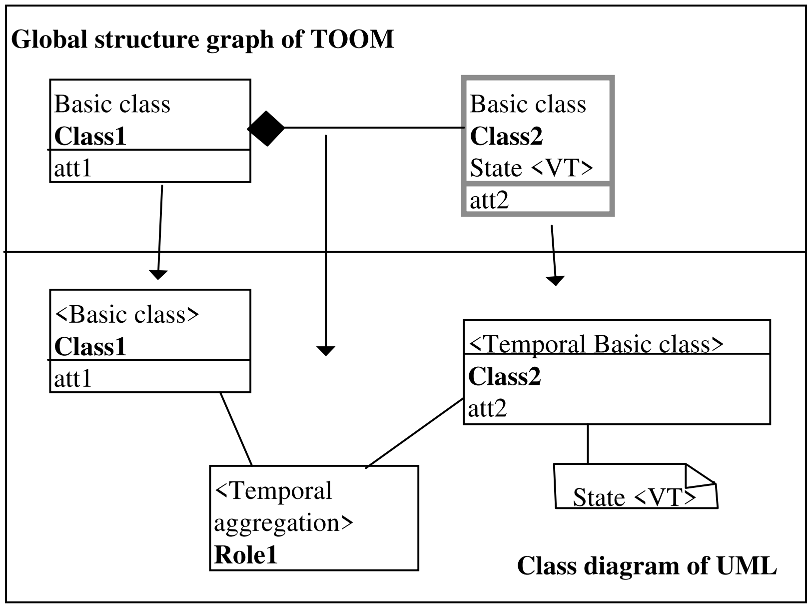


Figure 205: mapping of a temporal aggregation

5.3.1.14 Inheritance

The inheritance concept is the same than for the mapping from OOM, as explained in Section 5.2.1.7.

5.3.1.15 Summary

Global Structure Diagram of TOOM	Class Diagram of UML
Snapshot Basic class	Stereotype *<Basic object class>*
Snapshot Derived class	Stereotype *<Derived object class>*
Temporal Basic class	Stereotype *<Basic temporal class>*
Temporal Basic class History	Stereotype *<Basic temporal class>*
Temporal dictionary Basic class	Stereotype *<Basic temporal dictionary class>*
Class in State <VT>	*note* with « State <VT> » linked to the class
Class in State <TT>	*note* with « State <TT> » linked to the class
Class in History <VT>	*note* with « History <VT> » linked to the class
Class in History <TT>	*note* with « History <TT> » linked to the class
Class in History <VT,TT>	*note* with « History <VT,TT> » linked to the class
attributes of the class	attributes of the class
operation encapsulated in the basic object class	operation encapsulated in the class
derivation function encapsulated in the derived object class	derivation function encapsulated in the class as operation
queries encapsulated in the object class	queries encapsulated in the class as operations
event encapsulated in the object class	stereotype <event> class ascertain event
predicate of the internal event	note with the predicate
calendar class	stereotype <calendar>
origin of the calendar	attribute of the class
basic unit of the calendar	attribute of the class
set of granules of the calendar	attributes of the class
the three operations encapsulated in the calendar	operations encapsulated in the class
set of external formats of the calendar	attributes of the class
event encapsulated in the calendar class	stereotype <event> calendar ascertain event
predicate of the temporal event	note with the predicate
actor class	stereotype <actor>
event encapsulated in the actor class	stereotype <event> actor ascertain event
predicate of the external event	note with the predicate

5.3.2 From the Global Behavior Graph to the Class Diagram

5.3.2.1 Actor

The concept of actor in TOOM is the same as in OOM. As a consequence, the representation of an actor in UML is the same as explained in the Section 5.2.2.1.

However, the external events have a classification that does not exist in OOM. As a consequence, a note is necessary to specify which type of event is concerned.

Letter identifying the type of TOOM event	Note linked to the UML class
N	event in time
F	event a priori
P	event a posteriori

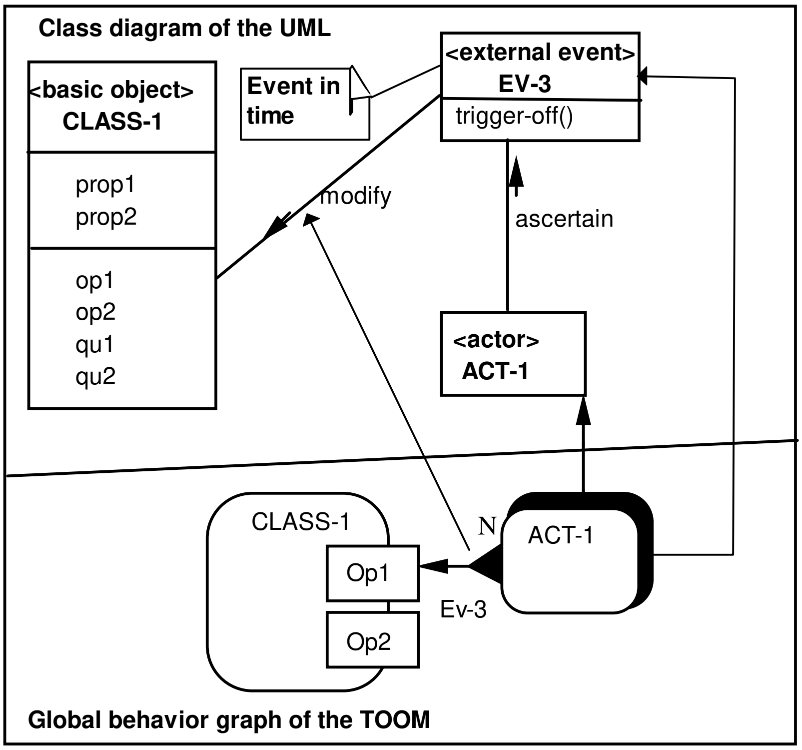


Figure 206: mapping of an actor

5.3.2.2 Calendar

The mapping of a calendar from TOOM to UML is the same as from OOM. As a consequence, the representation of an actor in UML is the same as explained in the Section 5.2.1.3. However, the temporal events have a classification that does not exist in OOM. As a consequence, As a consequence, a note is necessary to specify which type of event is concerned.

Letter identifying the type of TOOM event	Note linked to the UML class
A	event absolute
P	event periodic
R-o	event related to an object
R-e	event related to an event

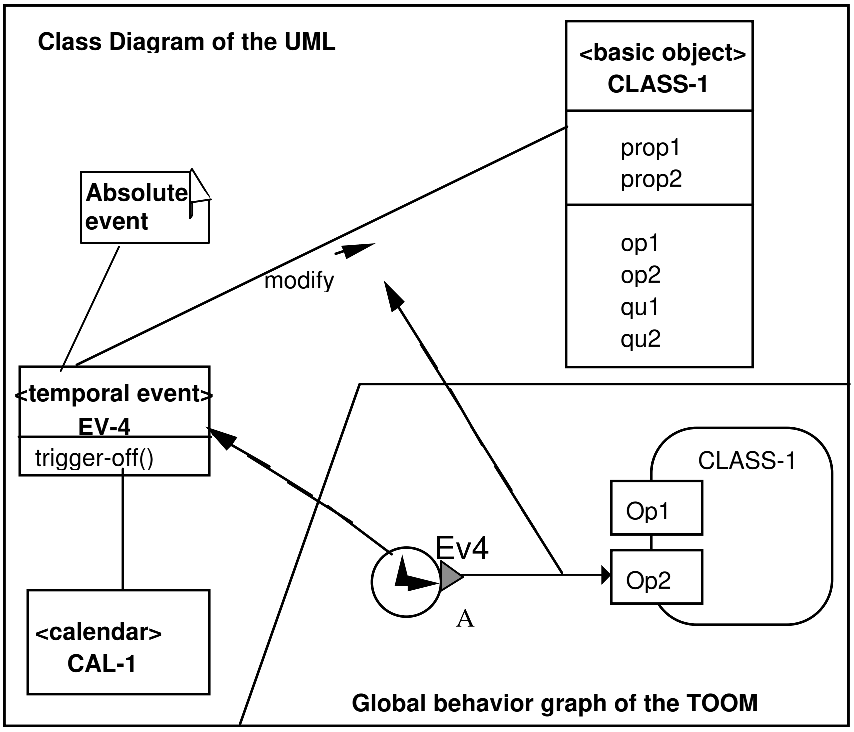


Figure 207: mapping of a calendar

5.3.2.3 Class

The global behavior graph can be useful to find the associations *modify* between the internal events and the modified classes.

The internal events have a classification that does not exist in OOM. As a consequence, As a consequence, a note is necessary to specify which type of event is concerned.

Letter identifying the type of TOOM event	Note linked to the UML class
O	event on object
H	event on history

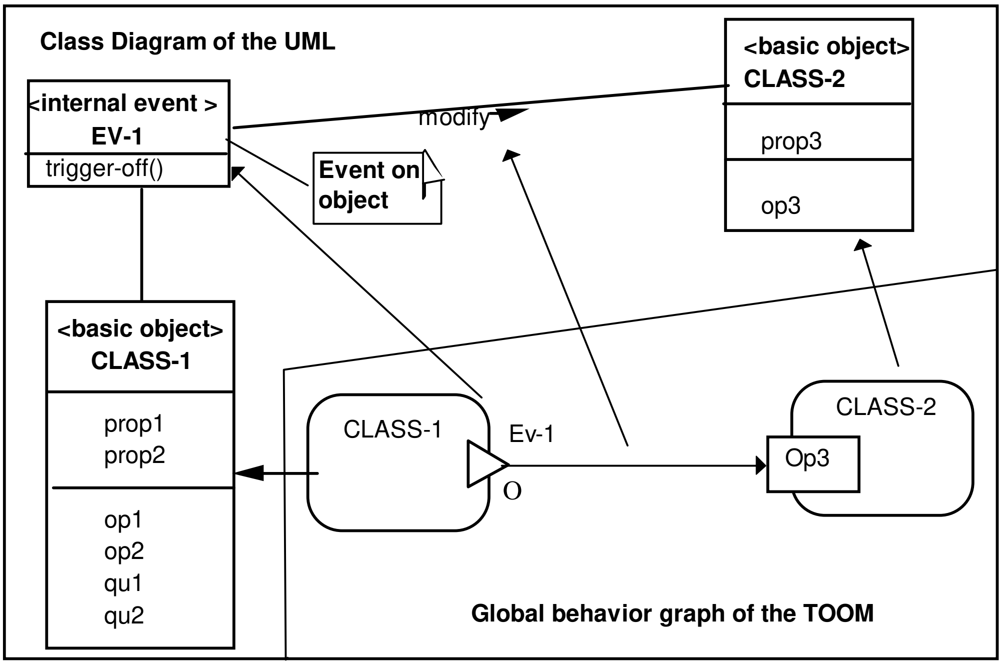


Figure 208: mapping of a class

5.3.2.4 Summary

Global Behavior Diagram of TOOM	Class Diagram of UML
actor	stereotype <actor>
calendar	stereotype <calendar>
internal event	class1 ascertain event event modify class2
internal event with the letter « O »	Stereotype <event on object>
internal event with the letter « H »	Stereotype <event on history>
external event	actor ascertain event event modify class
external event with the letter « N »	Stereotype <event in time>
external event with the letter « F »	Stereotype <event a priori>
external event with the letter « P »	Stereotype <event a posteriori>
temporal event	calendar ascertain event event modify class
temporal event with the letter « A »	Stereotype <event absolute>
temporal event with the letter « P »	Stereotype <event periodic>
temporal event with the letter « R-o »	Stereotype <event related to an object>
temporal event with the letter « R-e »	Stereotype <event related to an event>

5.3.3 From the Global Behavior Graph to the Sequence Diagram

The sequence diagrams are useful to show which operations are triggered by the events. We draw one sequence diagram for each event defined.

A sequence diagram contains at least three vertical lines: one for the release object, one for the event and one (or more) for the object(s) modified by this event.

It also contains at least two horizontal lines: one for the *trigger-off()* operation between the release object and the event and one (or more) for the operation(s) that modify the object(s) between this (these) one(s) and the event.

5.3.3.1 External event

The mapping of external event from the global behavior graph of TOOM to the sequence diagram is the same as from OOM. As a consequence, the representation of an actor in UML is the same as explained in the Section 5.2.3.1.

5.3.3.2 Temporal event

The mapping of temporal event from the global behavior graph of TOOM to the sequence diagram is the same as from OOM. As a consequence, the representation of an actor in UML is the same as explained in the Section 5.2.3.2.

5.3.3.3 Internal event

The mapping of internal event from the global behavior graph of TOOM to the sequence diagram is the same as from OOM. As a consequence, the representation of an actor in UML is the same as explained in the Section 5.2.3.3.

5.3.3.4 Control Condition

The control condition concept is the same in OOM and in TOOM. The mapping is presented in Section 5.2.3.4.

5.3.3.5 Summary

Global Behavior Diagram of TOOM	Sequence Diagram of UML
Actor	Actor on the first vertical line
Calendar	Calendar on the first vertical line
Release object	Object on the first vertical line

Event	Event on the second vertical line trigger-off() operation on the horizontal line between the first two vertical lines
Modified object(s)	Object(s) on the third vertical line, on the fourth, on the fifth ...
Operation that modify an object	Operation on the horizontal line between the event and the modified object
Condition before an operation	Condition on the horizontal line before the operation

5.3.4 From the Global Behavior Graph to the Use Case Model

As for OOM, we assume that each external and temporal event began a transaction into the application. As a consequence, each Use Case is the sequence of operations triggered by an external or a temporal event and can be named with its name. The mapping is explained in Section 5.3.4.

5.3.5 Case study

In the third part of this deliverable is exposed the case study of the loan management system in TOOM. This example is obtained from this TOOM model, following the mapping rules explained in this Appendix.

5.3.5.1 Class diagram

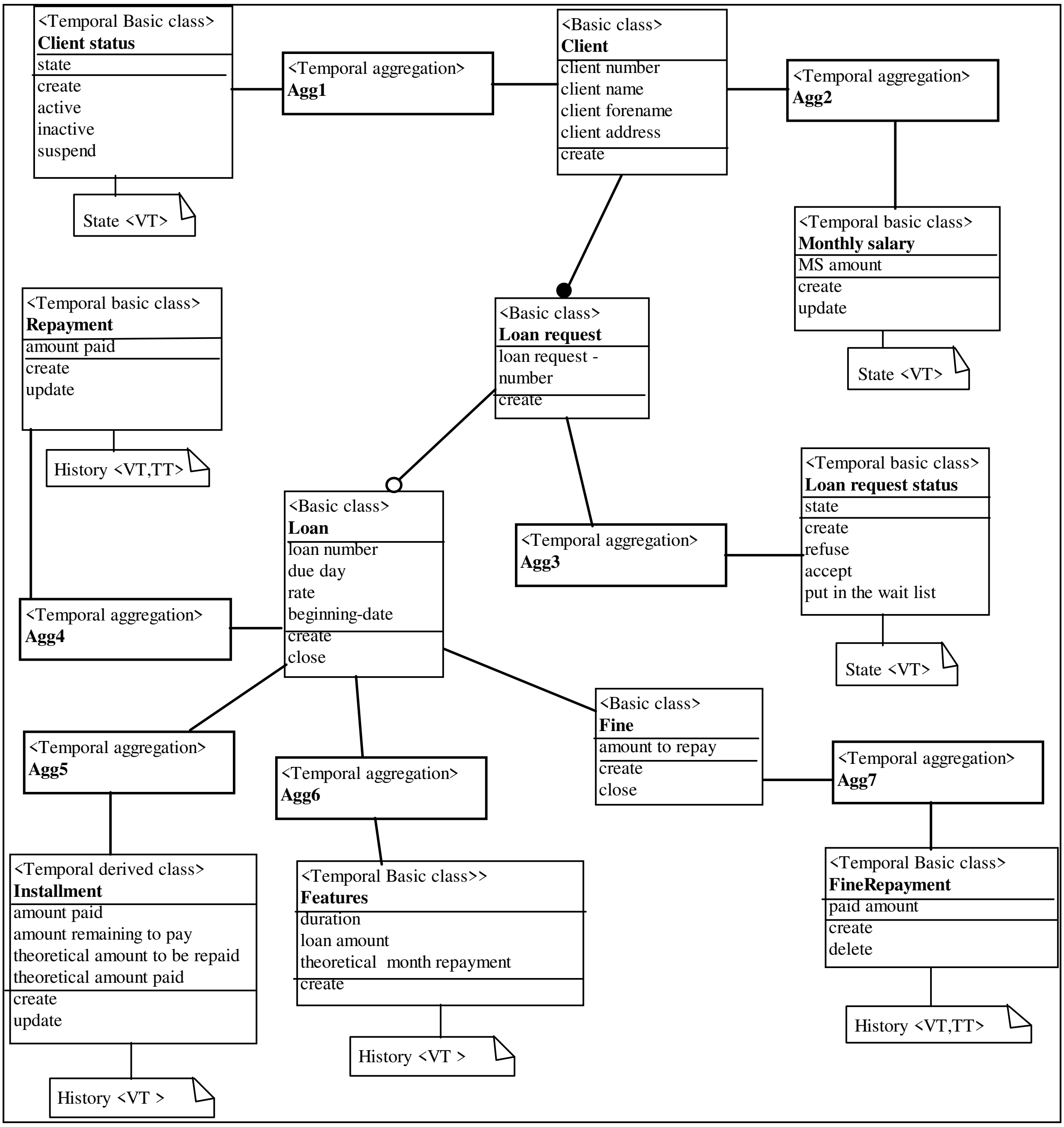


Figure 209: Class diagram (part 1)

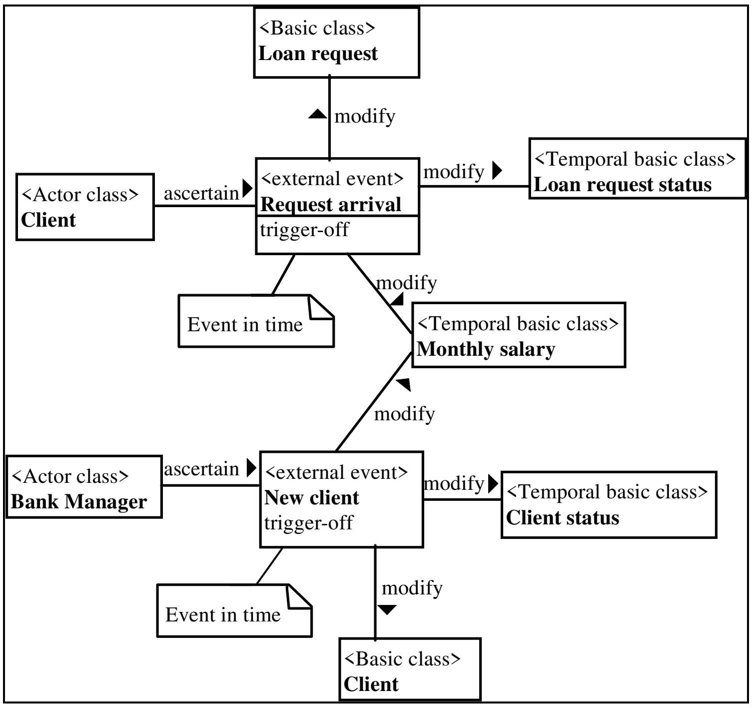


Figure 210: Class diagram (part 2)

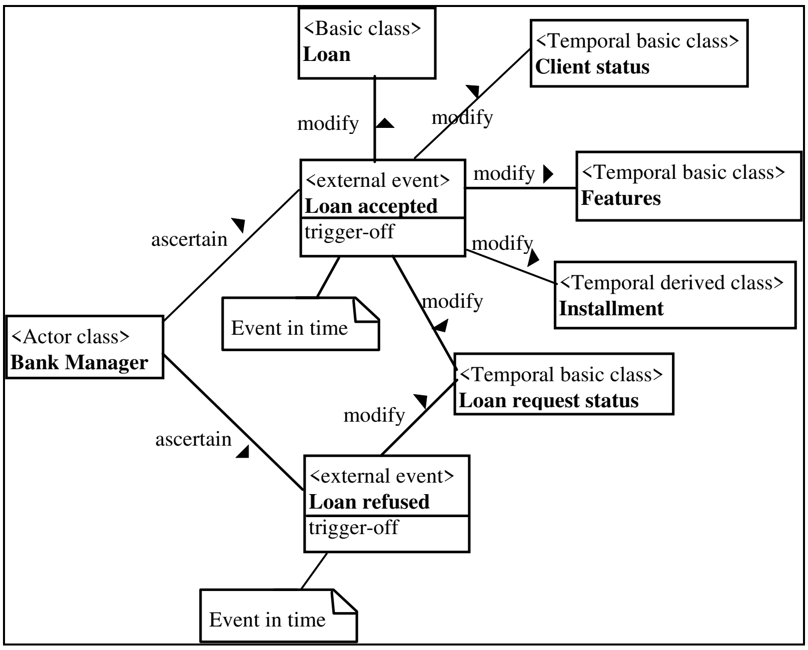


Figure 211: Class diagram (part 3)

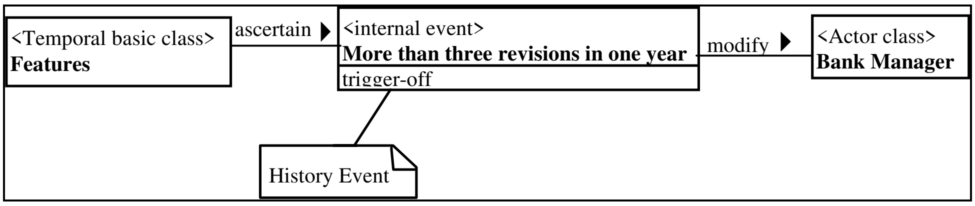


Figure 212: Class diagram (part 4)

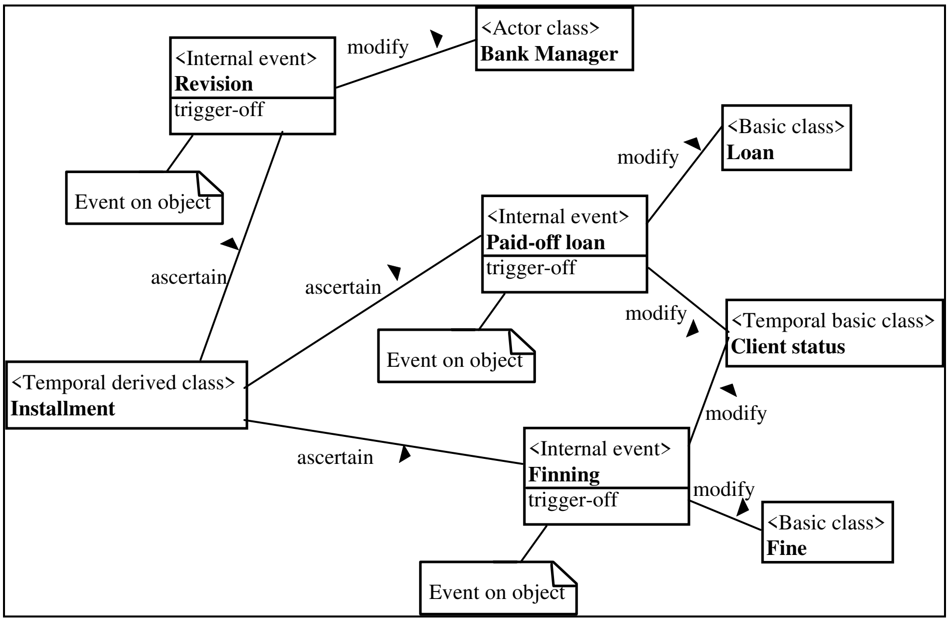


Figure 213: Class diagram (part 5)

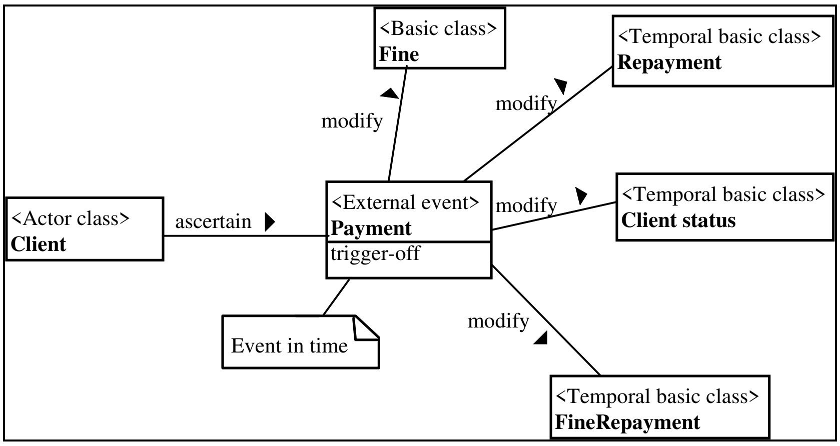


Figure 214: Class diagram (part 6)

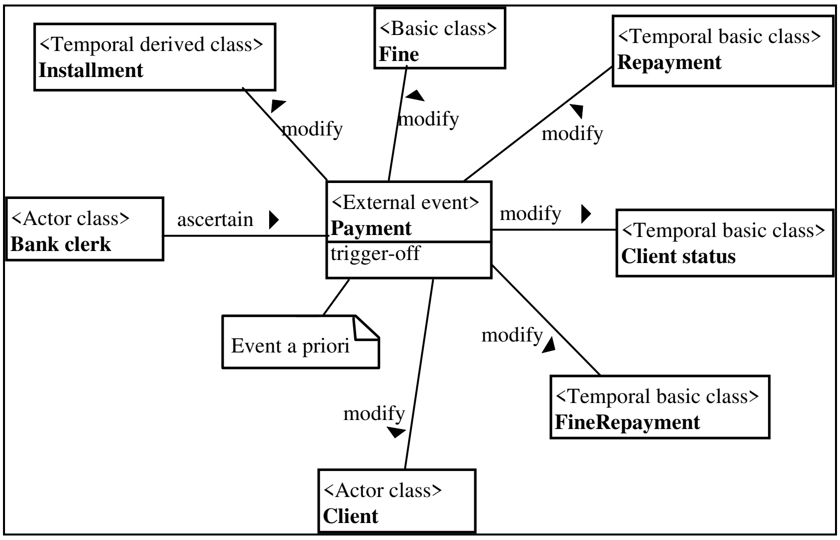


Figure 215: Class diagram (part 7)

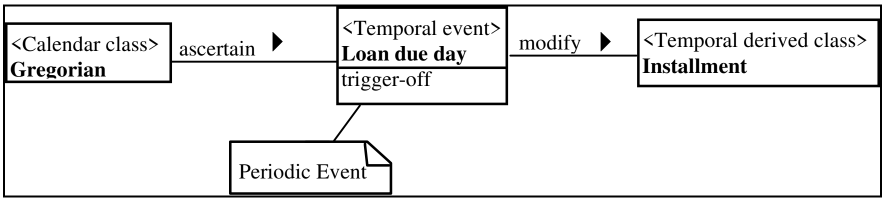


Figure 216: Class diagram (part 8)

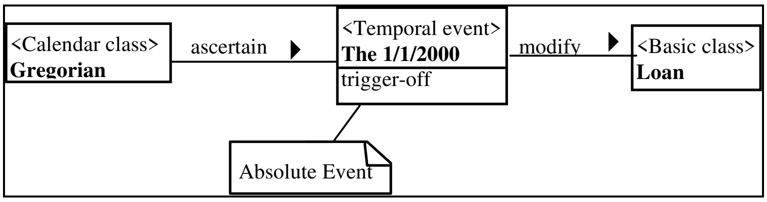


Figure 217: Class diagram (part 9)

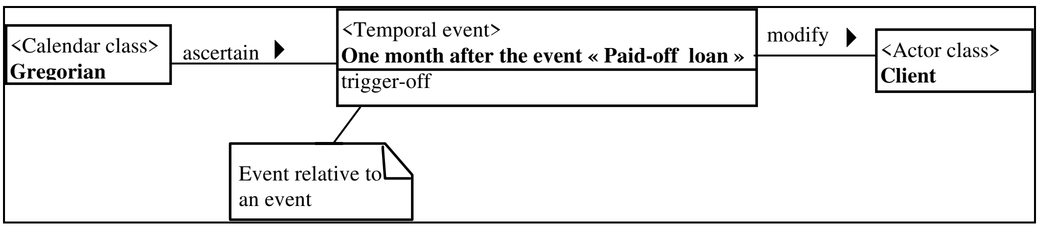


Figure 218: Class diagram (part 10)

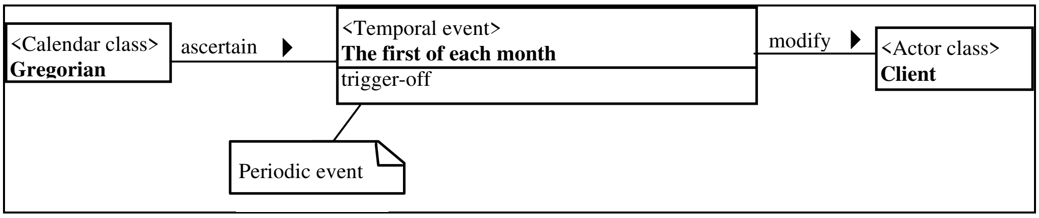


Figure 219: Class diagram (part 11)

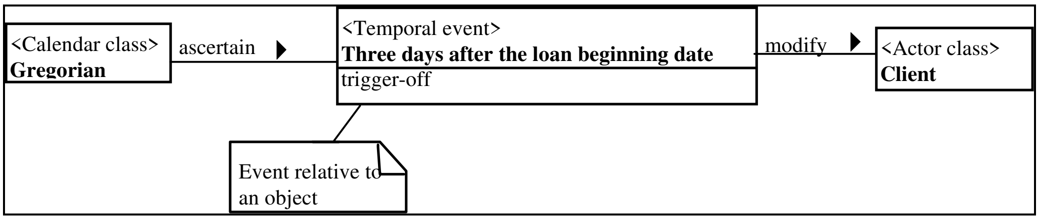


Figure 220: Class diagram (part 12)

5.3.5.2 Sequence diagrams

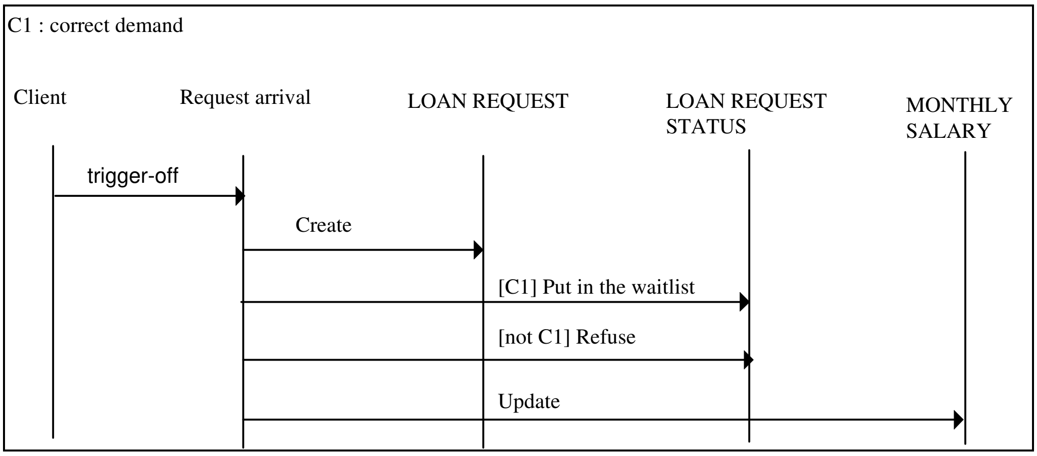


Figure 221: sequence diagram of the event *Request arrival*

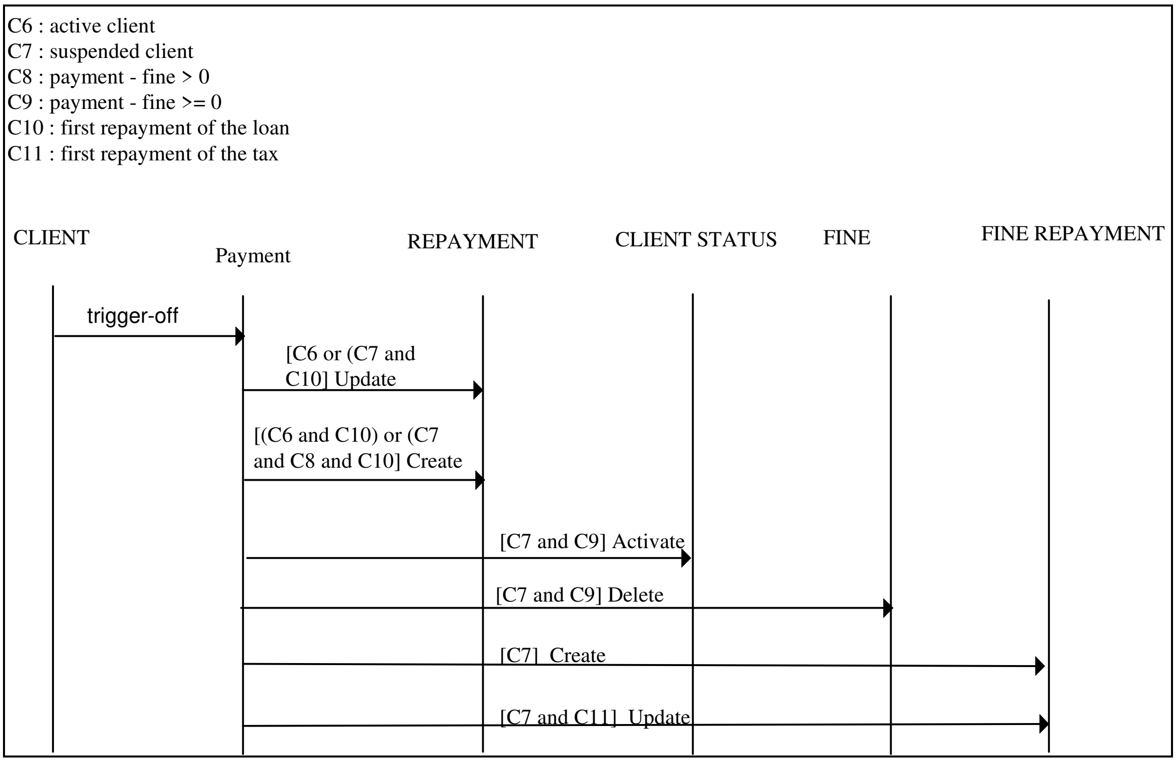


Figure 222: sequence diagram of the event *Payment*

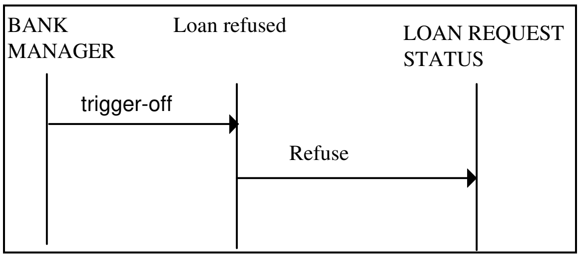


Figure 223: sequence diagram of the event *Loan refused*

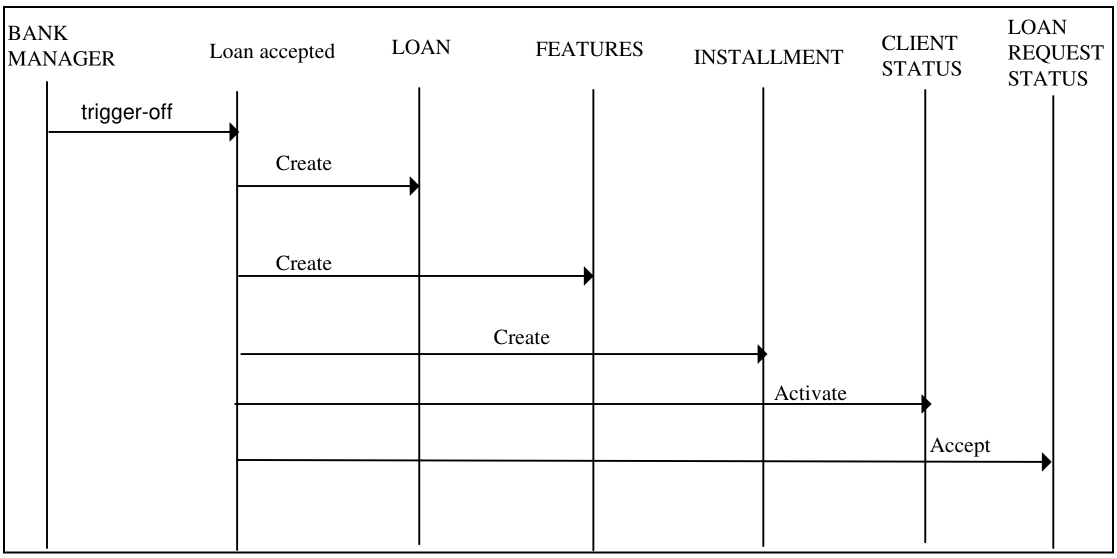


Figure 224: sequence diagram of the event *Loan accepted*

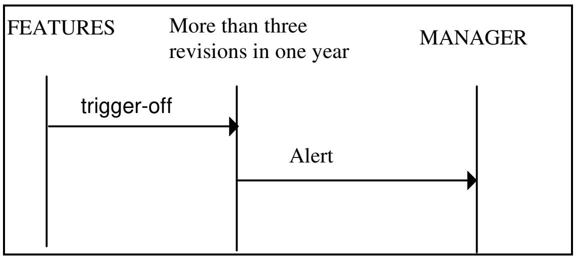


Figure 225: sequence diagram of the event *More than three revisions in one year*

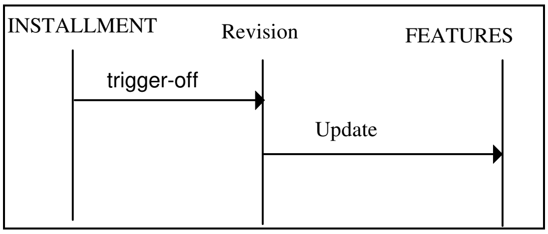


Figure 226: sequence diagram of the event *Revision*

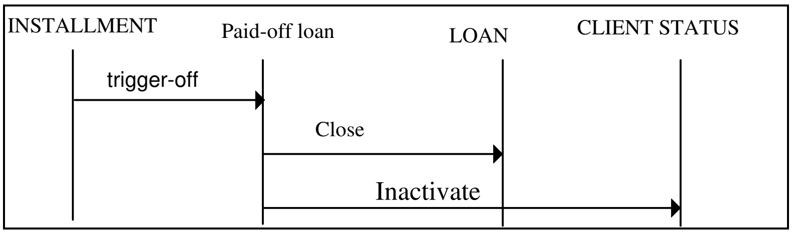


Figure 227: sequence diagram of the event *Paid-off loan*

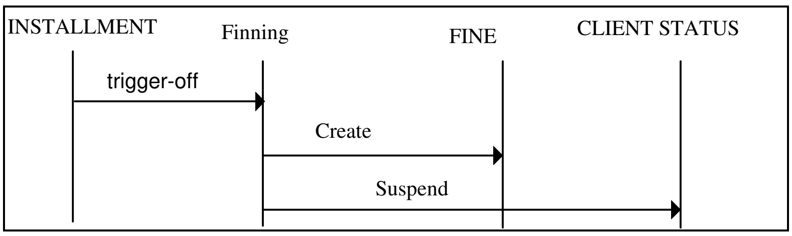


Figure 228: sequence diagram of the event *Finning*

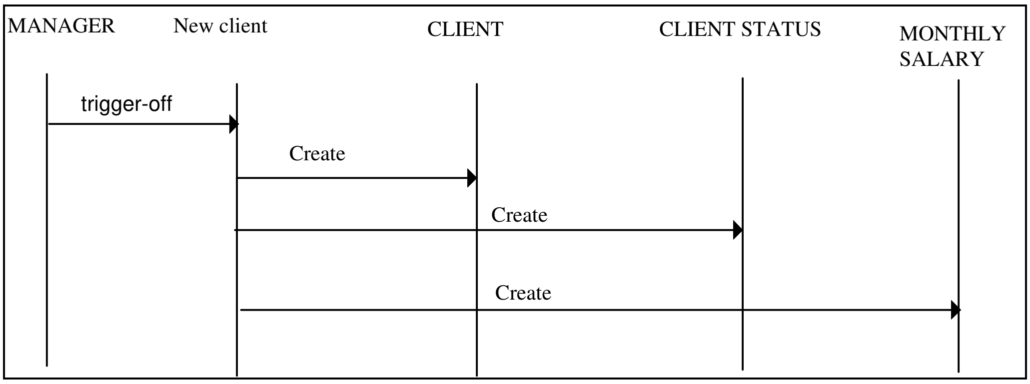


Figure 229: sequence diagram of the event *New client*

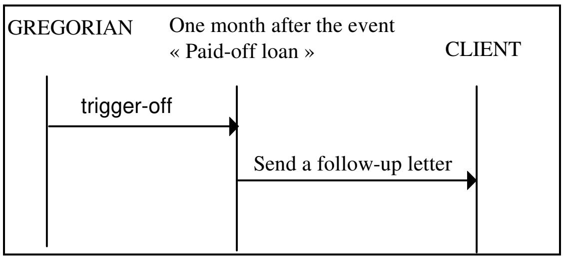


Figure 230: sequence diagram of the event *One month after the event "paid-off loan"*

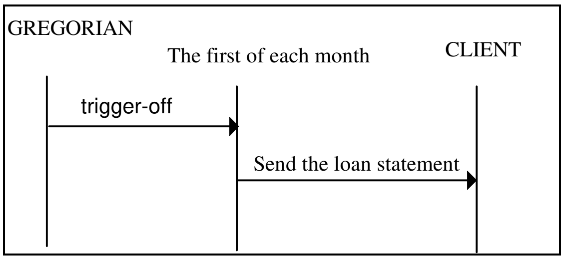


Figure 231: sequence diagram of the event *The first of each month*

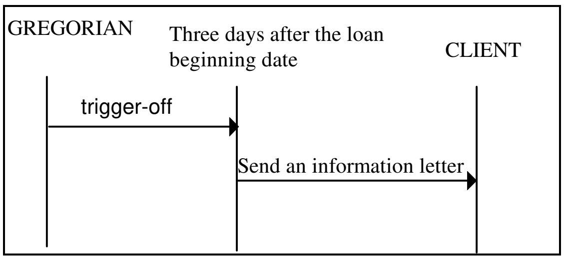


Figure 232: sequence diagram of the event *Three days after the loan beginning date*

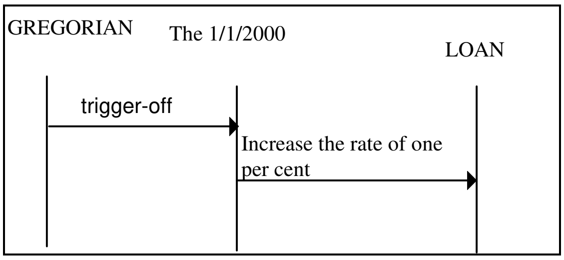


Figure 233: sequence diagram of the event *The 1/1/2000*

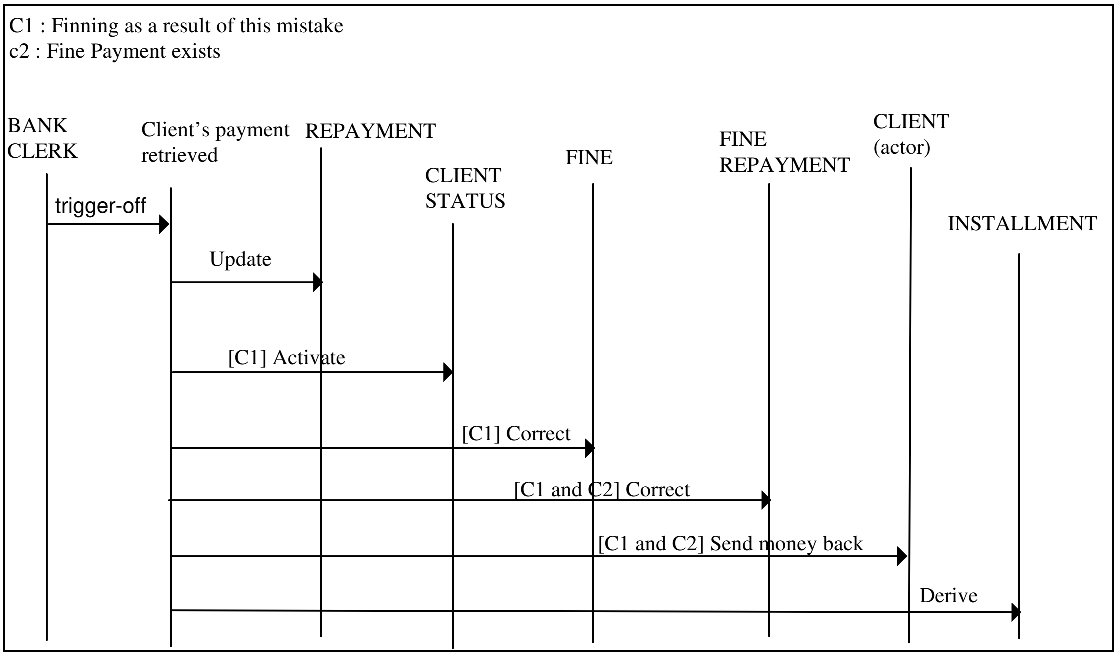


Figure 234: sequence diagram of the event *Client's payment retrieved*

5.3.5.3 Use case model

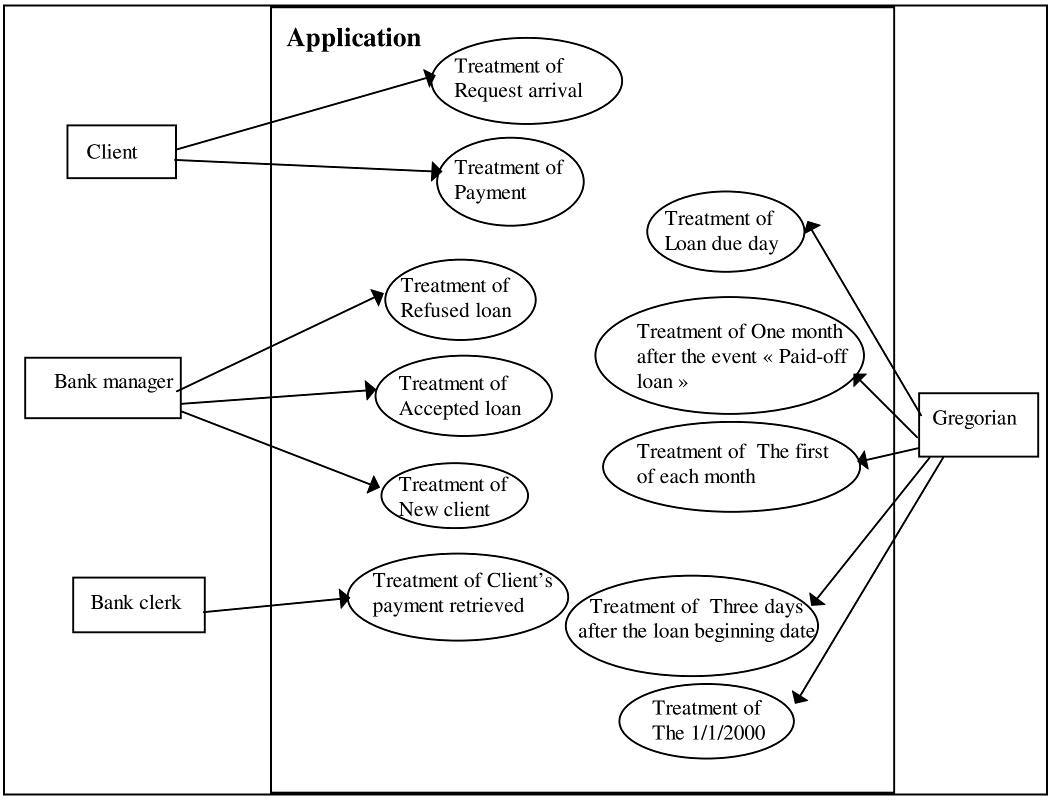


Figure 235: Use case model

5.4 References

[BOOCH95] Grady Booch, James Rumbaugh. *"Unified Method"*. Version 0.8. 1995.

[BOOCH96] Grady Booch, Ivar Jacobson, James Rumbaugh. *"The unified Modeling Language for Object-Oriented Development"*. Documentation Set, Version 0.91 Addendum, UML Update. 1996.

[SOUVEYET96a] C. Souveyet, C. Rolland, R. Deneckère, J. Ralyté. *" State of art : A comparison of six oriented object methods"*. TOOBIS deliverable T23D1.1. 1996

[SOUVEYET96b] C. Souveyet, R. Deneckère, C. Rolland. *" The object oriented methodology (OOM)"*. TOOBIS deliverable T23D1.1. 1996

[SOUVEYET96c] C. Souveyet, R. Deneckère, C. Rolland. *"The temporal extention (TOOM)"*. TOOBIS deliverable T23D1.1. 1996