Generalization and Specialization of Object Dynamics

V. Gamito Dignum

R.P van de Riet

R. Wieringa

Department of Mathematics and Computer Science
Vrije Universiteit Amsterdam
The Netherlands

November 89

ABSTRACT

This report presents a quite detailed analysis of the modeling approaches of different object-oriented database systems, namely ABSURD, OBLOG, MOKUM, TAXIS and GALILEO, with emphasis on the specialization of object dynamics, in a taxonomic structure. It is done uniformly by applying each system to the specification of an UoD example, the University world. The potentialities of the systems are compared and ambiguities discovered. An attempt to resolve some of these ambiguities is made. The issues dealing with inheritance and taxonomy of the dynamics of an UoD specification are particularly treated. The formalization of specialization issues for events and processes, which is specially cared in the systems ABSURD and OBLOG, is described in detail.

Table of Contents

1. Introduction

1.1. Motivation

Object-oriented programming began as early as 1967 with the simulation language SIMULA [Dah67], but the breakthrough is usually attributed to Smalltalk-80 [Gol83]. According to this development objects are highly *dynamic* entities. They are not just data. And they are not just software modules with an interface of named operations and a hidden local state, although this aspect applies to objects, too. But there is more. Objects are organized into *classes* which display a sophisticated *subclass* structure, together with an appropriate *inheritance* mechanism. Moreover, there is a whole world of object *dynamics*: objects can be created, destroyed, and changed, and they may have internal activity of their own. And finally, there is some mechanism of *communication* between objects, for example by means of *messages*.

In the last years, object-orientation has entered the field of databases, but here the emphasis is more on structural aspects. Object-oriented databases have attracted much interest, but there is little concern about theoretical foundations. The systems ABSURD and OBLOG, discussed in this paper, are attempts toward the definition of a coherent theory for object-oriented database systems.

Some work is being made towards a comparison of the several database systems which claim to be "object-oriented". Specially interesting are the works of Atkinson and Buneman, and Urban and Delcambre.

The work of Atkinson and Buneman [Atk87] focuses, in particular, on problems of a uniform type system and object persistency. Other important issues in their work are polymorphism, type inheritance, object identity and the choice of structures to represent sets of similar values. They do not concentrate specifically in object-oriented databases but in the larger set of database programming languages.

The work of Urban and Delcambre [Urb86] concentrates on semantic data models and on the influence of abstraction techniques in data and knowledge representation. They present a uniform analysis of structural, dynamic and temporal aspects of object-oriented models that support classification, aggregation, generalization and association abstractions.

Howevern, neither of these works give special attention to the important problem of specifying dynamics of objects and their hierarchies. With the development of more and more object-oriented database systems, it is necessary to give formal semantics to the languages. We will analyze two different approaches to the definition of formal semantics. The first is being developed by Wieringa, as part of his PhD thesis, at the Free University Amsterdam, and constitutes the system ABSURD (see, for example, [Wie89d]). It is based on algebra, more concretely on algebra of communicating processes. The other approach is the one of OBLOG. It uses algebraic categories. (see, for example, [Ehr89]). OBLOG has been developed in the scope of an ESPRIT project involving the Technical University of Lisbon and the University of Dortmund (FRG), et. al.

In section 1.2 a brief discussion of terminology and most common concepts of object-oriented databases is provided. The working example, which models an university environment, is also presented in end of section 1.2. Chapter 2 gives a detailed overview of the systems: ABSURD, OBLOG, MOKUM, TAXIS and GALILEO. In chapter 3 aspects concerning specialization and

generalization of events and processes are analyzed. Special attention is given to the formal specification, given by the systems ABSURD and OBLOG, to this problem. Some thing will be said about the logic of pre-conditions of events. Finally, chapter 4 presents our concluding remarks as well as comments on future research with respect to formal specification of dynamic inheritance.

1.2. Survey

In this section, we introduce the most common concepts one uses in the area of object-oriented databases. In the second part the two examples which will be used during the rest of the paper, when comparing the different languages, will be introduced.

1.2.1. Concepts

In the area of object-oriented database systems many concepts are used. However, the use of concepts is not always uniform in all the systems. Not always does one word refer to one concept. In the following we shall try to give a standard definition of the most important concepts.

Objects represent both the concrete and abstract entities of interest in the application domain. Objects have an **identity** which remains unchangeable during the whole life of the object. Moreover, the identity of an object makes it different from any other possible object. Objects can be grouped together into **classes**.

Atomic values in an application are viewed as the simplest form of objects. Sets of atomic values (also called **atomic objects**) are referred to as **data types**. (For example: *INTEGER*, *BOOLEAN* or *STRING*.)

An object has an internal state which can be observed through its **attributes**. For each attribute we assume a set, called the **range** of the attribute, which determines the values the attribute can have. The domain of an attribute can be an arbitrarily complex set, i.e. we also admit, object-valued and set-valued attributes, among others. All the instances of a class share the same set of attributes, called the **type** of the objects. For example the class *Person* has the attributes *name*, *address* and *age* with domains *NAME_VALUES*, *ADDRESS_VALUES* and *AGE_VALUES* and the object *john_smith* has as values for these attributes, *JOHN SMITH*, *MILLSTR. 31* and *35*, respectively The list of attribute values for each attribute of an object is called the **state vector** of the object. The **state** of the system is the union of the states of all objects.

Changes in the state of an object correspond to **events**. The effect of an event over an object is a change in the values of some of its attributes. Besides these *modification* events we can also expect *creation* and *destruction* events responsible for creating and destroying the object. The identity of the object is never changed by an event. **Processes** are possible sequences of events which define the *life* of an object. Instances of a class share the same set of possible events.

And, last but not least, objects **communicate**. Communication is the simultaneous execution of two events which need each other to be executed.

Most of these concepts are present in each of the studied systems. Table 1 in appendix A gives the terms by which these concepts are referred to in the different systems, if any.

Static aspects of an object-oriented language include **identification** mechanisms and

abstraction. Refer to Table 2 in appendix A to a summary of structural concepts.

Identification mechanisms provide a way to identify and distinguish objects. *Internal* identifiers uniquely represent the existence of an object in the system. They are also called *surrogates*. Surrogates remain equal to themselves during change. Usually surrogates are not accessible from the outside world, thus we also need means for *external* identification. *Keys* are usually used for external identification. They provide a unique identification for objects, but in contrast to surrogates, keys are not unchangeable. Moreover, one object can have more than one external key. For example, persons can be externally identified by their name, their identity card number, name and birthday, etc., depending in the organization of the database.

Abstraction is a fundamental conceptual tool used for organizing information. The following are a few aspects of abstraction which are useful to describe a conceptual model.

Aggregation. Treating a collection of concepts as a single concept. For example, person can be naively thought of as an aggregation of its *name*, *address* and *age*. **Decomposition** is the inverse of aggregation since it decomposes an entity into its constituent parts. Important aspects concerning attributes, or objects as an aggregation of attributes, are the possible values, i.e whether the value of an attribute can be a *set* of values (e.g. the attribute *children_of* of *Person*) or only one *single* value (e.g the attribute *birthday*). Or whether an attribute can be *object* valued (e.g the object *father_of*). According to the way values of an attribute are calculated, attributes of an object can be *defined*, if their actual value is stored somewhere in the database, and, they are *derived*, if their value is calculated in run-time from the values of other attributes. Another aspect is whether *restrictions* to the values of attributes can be defined.

Classification. Grouping entities (objects) that share common characteristics into a *class* over which uniform conditions hold. The inverse of classification, **Instantiation**, is used to obtain the entities which form a class. For example the class *Person* can be derived from the objects *john_smith* and *mary_brown*, which, for their part, are instances of the class *Person*. An aspect of study is the difference between a class (as a set of objects) and a type (as the structure of a set of objects).

Generalization. Extracting from one or more given entities the description of a more general entity that captures the common aspects but suppresses some of the detailed differences in the description of a given entity. The inverse of generalization, **Specialization**, creates a new entity by introducing additional detail to the description of an existing one. For example the class *Person* is a generalization of *Student* and *Employee*, which are specializations of *Person*. Generalization/Specialization structures the classes into a *taxonomy*, or the so called *is_a hierarchy*. Important aspects are the *inheritance* of attributes by the objects in the subclass and whether such inherited attributes can be subjected to stronger restrictions. Another aspect is the *multiple inheritance*, i.e. if a class can be at the some time subclass of more than one other classes, and inherit attributes from them all. For example the class *Working_Student* is at the some time a subclass of *Student* and *Employee*.

Association. Objects, possibly heterogeneous in nature, are viewed as higher-level generic objects, or *group-objects*. By heterogeneous group-object we mean a group-object which members are instances of different, and not associated by specialization, classes. One interesting aspect is the definition of attributes for sets of objects or not. For example the attribute *avg_salary* applies to a set of objects and not to a single object. Some systems consider

metaclasses which are classes in which the instances are themselves classes to view a whole class as a group-object.

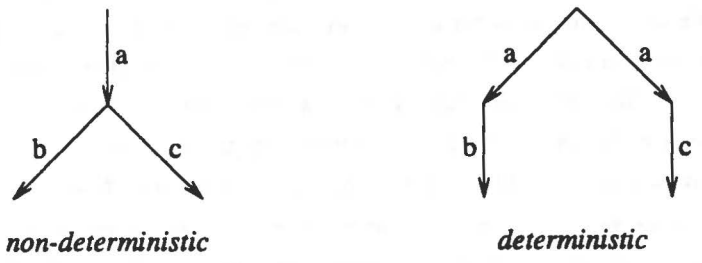
Finally, we will introduce the more important dynamic aspects of object-oriented systems. Refer to table 3 in appendix A to a summary of concepts.

Concerning the definition of **events** it is important to know whether *pre-* and *post-conditions* for the occurrence of an event can be defined. In some systems *message* events can be defined. Message events, or just messages, are events which must communicate with other events. In the case that events are inherited in a specialization relationship it is interesting to know if the effect of such an event over the inherited attributes (attributes of the superclass) can be redefined and if the definition of the event can be extended in order to incorporate the effects of the event over the attributes of the subclass.

Another interesting notion is the notion of **role playing** [Wie89e]. There are basically two ways to view the specialization relationship between classes. The first is a static definition of subclasses, i.e. objects are born members of a subclass and when they die they are still members of such a subclass (consider for example the subclasses *Oak* and *Pine* of the class *Tree*). In the other way, there is a dynamic definition of subclasses, i.e. objects can enter and leave a certain subclass during their life (consider for example the subclasses *Student* and *Employee* of *Person*). In this last case we speak of *role playing*.

Events are composed into **processes**. The most common composition operators are: *sequential*, *alternative* and **parallel** composition. With respect to the definition of **processes** we are interested in knowing whether the system provides ways to *formally* define a process, such as by using an algebraic language. Also we are expecially interested in the inheritance of processes, when that is possible.

Another aspect to be focussed in this report is **parallelism**, i.e. the way . There are two ways to model parallelism, by *partial order models* or by arbitrarily *interleaving* models [Bak89]. In partial order models, event occurrences are partially ordered, and in interleaving models they are totally ordered. In an interleaving model, parallel execution is the sequential execution of the next event in each of the parallel processes. The choice of which process will be advanced first is arbitrary.



non-deterministic *deterministic*

Fig 1.2.1.1

A system is **deterministic** if all states of the system depends completely on the past states, and it is **non-deterministic** otherwise. I.e. there is a random characteristic in non-deterministic systems. A process is non-deterministic if it contains a choice between process starting with the same event. For example, and using the syntax of ACP [Ber86], $a(b+c)$, event a followed by a choice between b and c, is nonderterministic but $ab+ac$, a followed by b or a follwed by c, is

deterministic. Figure 1.2.1.1 shows the graphical difference between these two precesses.

1.2.2. Working Example

The following example of an UoD will be used to test the potentialities of each system. We will model the example in all the studied systems and discuss several problems, araising from this experience.

Conventions:

As we have been using

Class:Words in italics and starting with an upper-case letter denote class names.

attribute:Words in italics and starting with an lower-case letter denote attribute names.

Process:Words in triumvirat and starting with an upper-case letter denote processes.

event:Words in triumvirat and starting with an lower-case letter denote events.

University environment

This example models the administration of a university concerning enrollments of students in subjects. It was adapted from [Bor84].

Relevant classes are *Student*, *Eng_Student* , *Subject* and *Enrollment*.

Eng_student is a specialization of *Student* containing enginering students. The class *Enrollment* contains all the information about enrollments of students in subjects, (it is thus dependent on the classes *Student* and *Subject*).

A student enrolls in a subject according to the following procedure: first s/he must indicate his wish to the Student Administration, then s/he will attend the class until end of term when the teacher is supposed to give him/her a grade. If that doesn't happen within a certain period of time the teacher is warned again and again until a grade is given; then the process of enrollment for the student is completed.

When considering enginering students things are a bit different: they must still follow the normal student procedure but a mid_term grade is also supposed to be given. If not, the teacher will also be warned again and again as with final grades.

So the relevant events for *Student* are: *become_student*, *enroll_subject*, *get_grade* and *drop_out*; *Eng_student* besides the events inherited from *Student* have: *get_mid_grade*. *Enrollment* have the events: *enroll*, *term*, *remind_term* and *grade* and, finally, *Enroll_Eng* (specialization of *Enrollment* concerning enginering students) also has *mid_term*, *remind_mid_term* and *mid_grade*.

2. Systems

In order to evaluate the power of the different systems there is a number of questions which should be answered for each system. Such answers will help to understand the philosophy of each system as well as its behaviour. In addition, the specification of the examples will be made using each of the different systems, which will make it easy to compare them.

The questions we will ask are:

- Which are the concepts of the language?

- How is object identification done?
  1. Internally
  2. Externally
- How are objects related?
  1. Classification
  2 Aggregation
  3. Generalization/Specialization
     i) Role playing
  4. Association
  5. Attribute sharing
- How do objects communicate?
  1. By message
  2. By sharing of 'code'
- Is there parallelism of actions?
  1. Interleaving
  2. Partial order
- Is it a non-deterministic system?

When we present the concepts of each system, we will have in mind the standard concepts introduced in section 1.2.1. Table 1 (in appendix) provides a summary of the terminology presented in section 1.2.1. and the corresponding terms used in the models studied.

Each system will be discussed in one subsection. The structure of all the subsections is:

1. Concepts of the language
2. Identification
3. Relationships between objects
4. Communication

2.1. ABSURD

ABSURD has been defined by R.Wieringa at the Free University Amsterdam. It describes a formal framework which incorporates several structures proposed in different published approaches. The system is introduced in [Wie88a, Wie89a, Wie89b].

Domain structures, both static and dynamic, must be formal. Two good reasons for formalizing are, first, formal analysis is an excellent way to eliminate obscurities and inconsistencies in any conceptual structure. Second, data models are to be implemented in a digital computer, which by nature are to be programmed in a formal language. The demand of formality can be viewed as the demand for maximal implementation-independence, for the only thing known about the implementation is that it will only accept models described in a formal language.

An important distinction to be made is between the *Universe of Discourse* (UoD) of an information system and the *conceptual model* (CM) of the UoD. The CM is an abstract structure representing the shared understanding of relevant aspects of the UoD. The CM is formed by

abstraction from the UoD and described by a specification. The CM is thus a *model* for the UoD in several meanings of the word.

The CM has two parts, the *conceptual schema* and the *conceptual database*. The former contains *universals* and the later *individuals* (abstract representation of individuals in the UoD).

2.1.1. Concepts of ABSURD

The specification of a CM is divided in four parts, in which atomic values, attributes, events and processes are specified. The specification of atomic values is done in a *value specification language* (VSL), the specification of attributes, events and processes is done in a *object specification language* (OSL). The two languages together are called L_{ABSURD}.

In the following, many definitions will be presented which have no correspondence in the standard terminology given in section 1.2.1. ABSURD gives formal definitions for many notions that are usually used quite informally.

Sets

An important notion within a CM is the notion of **set**. Traditionally a database is a set of ground atoms which express contingent facts. The set of all possible sets of ground atoms represents the set of all possible states of the database and can be treated as the intension of the database.

We introduce L as a set_theoretical language with only one predicate, the binary predicate $\in$, with the usual semantics of set-membership. L is a first-order language as it does not quantify over predicates. Other predicates can be introduced by definition, as it is common in set theory[1]. Important predicates we can define are $=$, $\subset$, $\subseteq$ and $f: x \rightarrow y$, which is a 3-ary predicate saying that f is a set of functions from set x to set y. We also assume the symbol $\varnothing$ as denoting the empty set.

The domain D into which L is interpreted consists of all the abstract objects which can be talked about in L. More elements are to be added to D in the sequel. D is required to be model for a set of axioms specified by the domain designer.

Surrogates

Surrogates represent the fact that there is a UoD state in which an entity exists, differs from other possible entities and remains equal to itself during change. I.e surrogates provide the internal identification for the entities. The minimal CM is a set of surrogates what means that "there exists a number of different things over a period of time". However, nothing is said, yet, over which state of the world they exist or in which period of time or which properties those entities have. We introduce a set of constants S of L, which denotes the set of all surrogates, and CON_S, the set of surrogate names.

The first axiom to be added to D requires:

(1) *All elements of S must be named by a unique constant of L.*

This axiom is shared by all domains and is a combination of the *unique name* and *domain closure* axioms, from relational databases [Rei84].

1- For more over definition of predicates, see for example [Tak71]

A surrogate is called a *group* if it is a set of other surrogates, otherwise it is called *primitive*.

Attributes

In ABSURD, attributes are formally defined as sets of functions which map subsets of S into subsets of S.

Def. Absurd.1 - Attribute

An **attribute** is a set of functions $[k1 \rightarrow k2]$, where $k1, k2 \in S$ and $k1$ is called the **domain** of the attribute and $k2$ its **range**. L contains a constant A which denotes a finite set of attributes and a finite set CON_A of attribute names, such that $A,S \notin CON_A$ and $CON_A \cap CON_S = \varnothing$. □

For example, if we have a set $EMP\# \subset S$ (defined as set of employee numbers) and a set of surrogates of all possible employees, $Emp \subset S$, we can define the set of functions $emp\#:[Emp \rightarrow^{1-1} EMP\#]$, such that each employee has exactly one employee number and no two employees have the same employee number. We need to generalize the functional definition of attribute so that *emp#* is the set of all 1-1 functions from *Emp* to *EMP#* in order to be able to represent the UoD entities in all their possible states, i.e. in any particular state of the UoD there is a function in $[Emp \rightarrow^{1-1} EMP\#]$ which represents the correct values of *emp#* for the existing employees.

The following axioms are to be added to D:

(2) *An attribute name denotes the same attribute whenever it occurs.*
(This is done by requiring the attributes to have global names.)

(3) *All attribute names in CON_A name attributes in A.*

(4) *All attributes in A are named by constants in CON_A.*

A surrogate is called *structured* if attributes apply to it, otherwise it is called *atomic*.

Data Types

As data types we consider sets like natural numbers, integers, booleans, strings, and so on. We assume that there are sets of surrogates like *NAME*, *EMP#*, etc, possibly with operators, which act as atomic data types.

An atomic object is identified by (s,ϵ), with ϵ representing the empty state vector. In this case s is called a (*data*) *value* and (s,ϵ) an *atomic* (or *unchangeable*) *object*. Data types, defined as subsets of S, are then sets of data values. Data types are not classified by the set of attributes applicable to them but according to the operations defined to them. So, formally, a **data type** is a set of values together with a set of applicable operations. Names of atomic data types are in uppercase letters.

Types and State Spaces

A subset of A is called a **type**. The set $T=\mathcal{F}(A)$ (set of finite subsets of A) denotes *types*.

We can now define:

Def. Absurd.2 - State Space

Consider the type $t = \{a_1,\ldots,a_n\}$. We call **state space** of t to the set:

$$space(A) = \{\{(a_1,s_1),\ldots,(a_n,s_n)\} \mid a_i \in t \text{ and } s_i \in range(a_i)\}.$$

Metavariable over state space is σ. Each member of the state space, $\{\{(a_1,s_1),\ldots,(a_n,s_n)\}$, is called a **state vector**.□

Natural kinds

In the usual way a relational database is specified, its conceptual schema states that there are certain *kinds* of facts, together with the attributes which characterize these kinds of objects.

Considering attribute applicability we can divide the set of surrogates into **natural kinds**. I.e. natural kinds are sets of surrogates to which a *fixed* set of attributes is applicable. We introduce the constants $\mathcal{K}$ and $CON_{\mathcal{K}}$ in L denoting the set of natural kinds ($\mathcal{K} \subset S$) and the set of natural kind names, respectively. As all natural kind are subsets of S they may act as domain or range of attributes.

To understand better the concept of natural kind we need first to introduce the concept of **kind** and. The set $K = \mathcal{P}(S)$ (set of subsets of S) denotes *kinds*. We define now the functions

$kind: T \rightarrow K$ and $type: K \rightarrow T$

such that *kind(t)* is the largest set of surrogates to which all the attributes in t are applicable and *type(k)* is the largest set of attributes applicable to all the surrogates in k.

Natural kind is the largest set of surrogates to which a given set of attributes is applicable. Similarly **natural type** is the largest set of attributes applicable to a given set of surrogates. It is simple to prove (see for example [Wie89a]) that natural kinds are exactly the elements of K which are closed under $kind \circ type$ and natural types are closed under $type \circ kind$.

We need to add now another axiom to D

(5) *All natural kind names are constants from* $CON_{\mathcal{K}}$.

However note that we do not require *all* natural kinds in $\mathcal{K}$ to be named (there are usually more natural kinds than we want to name).

A question which arises naturally now is whether there are other natural kinds besides attribute domains. It can be easily proved that any intersection of natural kinds is also a natural kind. So, information is needed over natural kinds which intersection is not-empty. Such information must be explicitly given , and that is done with *specialization axioms*. For example, we must say that

(6) $Emp \subset S$,
$Dept \subset S$ and
$Emp \cap Dept = \varnothing$.

It can be shown that this gives *all* natural kinds given by attribute definitions. A natural kind is thus

1. an attribute domain
2. an intersection of attribute domains.

Note that the notion of natural kind is not equivalent to our standard notion of class. Class is a set of *objects* to which a certain set of attributes is applicable, whereas natural kind is a set of *identities* (surrogates) to which a certain set of attributes is applicable. However, as we will see in the following, the ABSURD notion of class will arise from this "more simple" notion of natural kind.

Objects and Classes

We are now able to give the formal Absurd definition of **object**.

Def. Absurd.3 - Object

A pair (s,σ) where $\sigma \in space(t)$ and $t \subseteq type(\{s\})$ is called an **object** with *identity* s and *space vector* σ. Metavariable over objects is o.□

For example, consider $e \in$ Emp and *name, address, dept* $\in CON_A$, attributes applicable to employees, then

(e, (name:John, address:Amsterdam, dept:d))

represents an employee object with **internal state** *(name:John, dept:d)* and **identity** e.

The set of all possible objects, the **universe** U is:

$U = \{(s,\sigma) | s \in S, \sigma \in space(t) \text{ and } t \subseteq type(\{s\})\}$.

A subset of U is called a **class**. The **class of kind k** is:

Def. Absurd.4 - Class of kind k

The class of kind k, where $k \in K$ is the set of objects

$obj(k) = \{(s,\sigma) | s \in k \text{ and } \sigma \in space(type(\{s\}))\}$

i.e. $obj(k)$ is the set of objects with identity in k and the largest state vector (i.e. in (s,σ)).□

This notion of class of kind k is similar to our standard notion of class, i.e. a set of objects to which the same set of attributes is applicable.

σ contains all attribute:value pairs applicable to s. $id(c)$ is the set of identities of objects in c, which is called the *kind* of c.

A **world** is a set of objects with different identity. The set of all possible worlds is called PW.

Finally, an object (s,σ) is called *group, primitive, structured* or *atomic* according to whether s is a group, primitive, structured or atomic surrogate. An object is *complex* if it has at least one attribute which value is a group. For example

$(e_1,())$ is an atomic primitive object,

$(e_1, (name{:}John))$ is a structured primitive object,

$(\{e_1,e_2\},())$ is an atomic group object,

$(\{e_1,e_2\},(avg\_salary{:}1000))$ is a structured group object and

$(d_1, (emps{:}\{e_1,e_2\}))$ is a complex object.

Events

Events are the equivalent to the update procedures and transactions in conventional databases. Events map objects of a certain kind onto objects of the same type.

An event is a change in the state of an object. The identity of an object is never changed by an event.

Formally, an **event** is a function on a class of objects. An event induces a mapping $PW \rightarrow PW$ by changing the state of an object. An example is

$change\text{-}address{:}ADDRESS \rightarrow (obj(Person) \rightarrow obj(Person)) \equiv$

$change\text{-}address(a_1)(p, (name{:}n, address{:}a_0,)) = (p, (name{:}n, address{:}a_1))$.

The set A of all attributes defines state spaces for all possible objects and thus a set F of all possible events. Each natural kind has a **repertoire** $rep(obj(k))$ of logically possible events which change the state of an object in $obj(k)$ events applicable to all the objects in $obj(k)$.

Only a subset of F, called E, of events will actually be defined for each application.

Because E is only a subset of all logically possible events, only a subset of $rep(obj(\mathcal{A}))$ will actually be defined for each $\mathcal{A}$. So, for each natural kind we have

$$E_{\mathcal{A}} = E \cap rep(obj(\mathcal{A})),$$

which is called the **defined repertoire** of $\mathcal{A}$. This will be the basis for the definition of the processes executed by objects of kind $\mathcal{A}$.

Communication

Sometimes events in the life of different objects or in the life of a single object, *must* occur simultaneously. In this case we speak of **communication**. There are two kinds of communication: *global*, between different objects and *internal* in the life of a single object. An example of global communication is:

$$e_g : \text{obj}(\mathcal{A}_1) \times obj(\mathcal{A}_2) \rightarrow \text{obj}(\mathcal{A}_1) \times obj(\mathcal{A}_2) \equiv$$
$$e((s_1,\sigma_1),(s_2,\sigma_2)) = ((s_1,\sigma'_1),(s_2,\sigma'_2))$$

This communication consists of two global messages

$$\text{obj}(\mathcal{A}_1) \rightarrow \text{obj}(\mathcal{A}_1) \equiv$$
$$e(s_1,\sigma_1) = (s_1,\sigma'_1)$$

and

$$\text{obj}(\mathcal{A}_2) \rightarrow \text{obj}(\mathcal{A}_2) \equiv$$
$$e(s_2,\sigma_2) = (s_2,\sigma'_2)$$

which are events in the lives of two different objects (s_1, σ_1) and (s_2,σ_2).

Communication is thus not a single event but a composition of message events. Therefore, we assume that E contains message events but not communication itself. Events in E which can occur on their own (i.e. not messages) are called *solitary* events. All events in E are *elementary*. (In contrast, communication is not an elementary event.)

Communication is formalized as a *finite set* of elementary events which are forced to occur synchronously. Thus, communication events are elements of $\mathcal{P}(E)$.

Process

Events are composed into **processes**, which represent the life cycles of objects of a natural kind. Absurd is, together with Oblog, the only systems which provide a way to formally define processes. The specification of generic processes is done in process algebra (see for example [Ber86]) Appendix B shows the axioms for ACP_τ (Algebra of Communicating Processes with abstraction). The main operations of ACP_τ are:

- $+$ choice between processes.
- $\cdot$ (usually omitted) sequence of processes.
- $\|$ parallel execution of processes.
- ∂_H encapsulation of processes. The events in the set H must occur synchronously. Each one of them is renamed to ∂, the deadlock symbol (i.e either they occur together or, the isolated occurrence of one of them, is equivalent to a deadlock).
- $|$ communication between processes. $e_1|e_2$ is the synchronously execution of the events e_1 and e_2. To enforce this we encapsulate the process by $\partial_{\{e_1,e_2\}}$ which causes both e_1 and e_2 to be renamed to ∂.
- τ_I abstraction of processes. The events in the set I are renamed to the invisible step τ. The

problem which leads to abstraction is that when a process is placed in a context of other processes with which it may communicate, there is always the possibility of deadlock, in which case the we can not obliterate all information about events we abstract away. The event τ is then left in their place. In particular, this is the case when the abstracted event is a guard (the first event after a choice).

With each natural kind $\mathcal{K}$ we associate a *generic process* K in which the event names are those defined in the event specification. We, therefore, add a new axiom to our domain D:

(7) For each natural kind $\mathcal{K}$ there is exactly one process specification, with exactly one solution, which is the generic process K of its instances.

Each s in $\mathcal{K}$ executes an instance of the generalized process K, called the *individual process*, Λ_σ^sK. If K is the generic process of natural kind $\mathcal{K}$, then the individual process executed by $s \in \mathcal{K}$ is:

$\partial_{H_{s,\mathcal{K}}} \Lambda_\sigma^s birth.\mathsf{K}.$

birth.K is the generic process K prefixed with a generic event *birth*. Λ_σ^sK instanciates this to the object with identity s and initial state σ, and $\partial_{H_{s,\mathcal{K}}}$ enforces the local communication (cf. axioms of ∂ in appendix B). The individual processes of all objects can be composed in parallel to form the **domain process**, D. In the domain process, the operator ∂_H enforces all global communications:

$$\mathsf{D} = \partial_H(\partial_{H_{s_1,\mathcal{K}_1}} \Lambda_{\sigma_1}^{s_1} birth.\mathsf{K}_1 \parallel \partial_{H_{s_2,\mathcal{K}_2}} \Lambda_{\sigma_2}^{s_2} birth.\mathsf{K}_2 \parallel \ldots).$$

The graph of the individual process, has edges labeled by event calls and nodes labeled by objects. Each node in the individual process graph can be mapped onto a point in *space*(*type*($\mathcal{K}$)) by the mapping

$(s, (a_1{:}s_1,\ldots,a_n{:}s_n) \mapsto s_1,\ldots,s_1)$.

The individual process describes how s may move through *space*(*type*($\mathcal{K}$)).

2.1.2. Identification

In the following, the underlying formal system is different from the one used in the subsection before. It is due to the evolution of the work of R. Wieringa. Thus, some of the definitions may not coincide.

In ABSURD, the specification of object identities takes the form of a value specification (data type) although conceptually it belongs to the natural kind specification because it makes available operators for object identities. Identities specify no generator constant, so it has trivial semantics with $ID = \varnothing$.

value spec Identities
import
Booleans
sorts
ID
functions
new : ID→ ID
eq : ID × ID→ BOOL
id : ID→ ID

variables

i, i1, i2 : ID

equations

[E0] eq(i, new(i)) = false

[E1] eq(new(i), i) = false

[E2] eq(new(i1), new(i2)) = eq(i1, i2)

[E3] id(i) = i

end spec Identities

The new function generates denumerably many distinct object identities for each sort with a generator constant, and the eq function tests for syntactic equality of object identities.

2.1.3. Relationships between objects

Classification

We introduced *natural kinds* as being sets of surrogates to which a fixed set of attributes is applicable. Formally a natural kind is the carrier, $\mathcal{K}$, of an identity sort k. For each natural kind $\mathcal{K}$ a **class** is thus a set of pairs (i,σ) where $i \in \mathcal{K}$ and σ is its state vector.

In the natural kind specifications, constants will possibly be added to *ID*, such as:

p0: PERSON and

d0: DEPARTMENT,

where PERSON and DEPARTMENT are subsorts of ID. These declarations *extend* the sort ID to one having the carrier:

ID = {p0,new(p0),...,d0,new(d0),...}.

For example, the identity of an object of kind *PERSON* is a closed term generated by the constant p0 and the function new : ID → ID.

It is not necessary for every natural kind specification to contain constants of sort ID. In fact Identities and *Persons* contain no constants of sort ID. The reason has to do with the taxonomy of natural kinds and it will explained in the next subsection. Here it is enough to note that the functions new and eq, which apply to all identity sorts, generate a carrier for every identity sort which contains a constant declaration.

As an example of the syntactic specification of natural kinds, the specification of *Persons* follows:

natural kind spec *Persons*

import

Identities, Names, Integers, Addresses

identity

PERSON **specializing** ID

attributes

name : PERSON → NAME [1-1]

age : PERSON → INT

address : PERSON → ADDRESS

events

birth : PERSON × NAME → PERSON
birthday : PERSON -> PERSON
change_address : PERSON × ADDRESS → PERSON

variables
p : PERSON
n : NAME
a : ADDRESS

equations
name(*birth*(p, n)) = n
age(*birth*(p, n)) = 0
age(*birthday*(p)) = *age*(p) + 1
address(*change_address*(p, a) = a

preconditions
change_address(p, a)
when *address*(p) ≠ a

process selection
PERSON = *birth*.LIFE
LIFE = (*birthday* + *change_address*).LIFE

end spec *Persons*

Specialization

Each natural kind specification has thus one distinguished sort, called the *identity* sort, subset of ID. Each identity sort specializes at least one other identity sort, with ID at the top of the specialization hierarchy. Syntactically, specialization is just a partial ordering on identity sort names. Semantically, specialization is interpreted in the same way as the subsort relation. The only difference is that the specialization usually *extends* the superkind. For example, Persons specializes ID by extending ID. Thus in the intended algebra of Identities, the carrier of ID is empty, but in the intended algebra of Identities + Persons, we have

ID = {*p0*, new(*p0*),...} and
PERSON = {*p0*, new(*p0*),...}.

ID is again extended by Departments, and we have the intended algebra of Identities + Persons + Departments:

ID = {*p0*, new(*p0*),...,*d0*, new(*d0*),...},
PERSON = {*p0*, new(*p0*),...} and
DEPARTMENT = {*d0*, new(*d0*),...}.

The taxonomic hierarchy is thus defined by the subsort relation and the suitable placement of generating constants. If we had a total partition of *Persons*, i.e. all the interesting persons are either students or employees (or both), the generator constants would be placed in the subkinds

Students, *Employees* and *Working_Students* and not in kind *Persons*. Applying the new operator we then get:

ID = {*s0*, new(*s0*),...,*e0*, new(*e0*),..., *ws0*, new(*ws0*),...},
PERSON = {*s0*, new(*s0*),...,*e0*, new(*e0*),..., *ws0*, new(*ws0*),...},
STUDENT = {*s0*, new(*s0*),...,*ws0*, new(*ws0*),...},
EMPLOYEE = {*e0*, new(*e0*),...,*ws0*, new(*ws0*),...} and
WORKING_STUDENT = {*ws0*, new(*ws0*),...}.

Aggregation

The attributes of a specification form an aggregation graph, as show on figure 3.1.3.1. for *Person*.

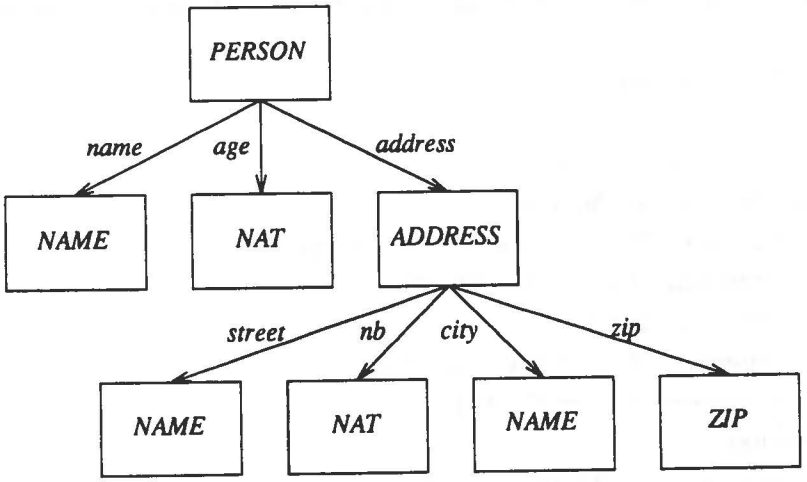


Fig. 3.1.3.1

In general the order of definition of attributes follows the reverse of the direction of the arrows in the aggregation graph, except when cycles in the graph occur. Cycles can occur in the definition of attributes. Consider, for example, that each course has a responsible teacher and a teacher can only be responsible for one course:

natural kind spec *Courses*
import
Identities, *Teachers*, ...
identity
COURSE **specializing** ID
attributes
(...)
responsible_for : TEACHER → COURSE
(...) **equations**
[EQ1] *teacher_of*(*responsible_for*(c)) = c
end spec *Courses*

This introduces the cycle in the aggregation graph as shown in fig. 3.1.3.2:

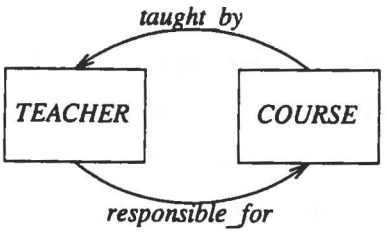


Fig. 3.1.3.2

Association

Using the parametrized specification *Sets* (cf. [Wie89d]), set classes can be formed. For example we may want to declare the kind *PersonSets* with an attribute *avg_age* which domain is a set of person and an attribute *cars_of* which value is a set of cars (owned by a person).

natural kind spec *PersonSets*
import
Persons, Integers, Cars
Sets **using** *Persons* **for** *Item*
binding [ITEM → PERSON, eq → eq],
renaming [SET → PERSONS]
Sets **using** *Cars* **for** *Item*
binding [ITEM → CAR, eq → eq],
renaming [SET → CARS]
attributes
avg_age : PERSONS → INT
cars_of : PERSON → CARS
variables
p : PERSON
pp : PERSONS
equations
[EQ1] *avg_age*(empty) = 0
[EQ1] *avg_age*(insert(p, pp)) = (*avg_age*(pp)*card(pp)+*age*(p))/(card(pp)+1)

(...)
end spec *PersonSets*

The aggregation graph of *PersonSets* is as in fig. 3.1.3.3. where an ellipse around a natural kind name represents the natural kind of finite sets of objects of that natural kind:

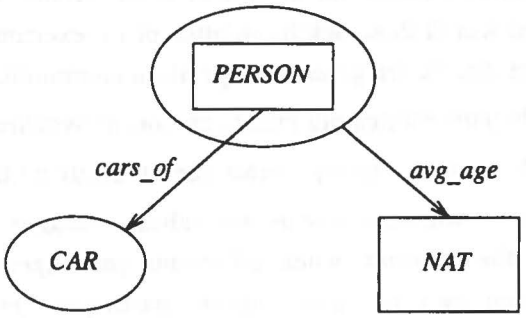


Fig. 3.1.3.3

Specification of objects with sets as identities allows us to specify objects like:

$(\{p_1,\ldots,p_n\}, (avg\_age{:}a))$

Existence

One more extension of the specification of a state of a CM is needed, the specification of the *existence* of objects. ABSURD does this by introducing a special object with identity db in the specification of a special natural kind **existence monitor**. The existence monitor is a standard part of every CM specification. In all specifications, it has identity sort DB, which is *not* a specialization of ID, and there is a single constant db in DB. In all specification db has two attributes *ext* and *old*, which hold the set of existing identities and the set of identities which have existed, respectively. The intersection of the values of these two attributes is always empty. By convention, every identity in *ext*(db) represents an existing object.

The rest of the specification of the natural kind *ExistenceMonitor* is quite large and domain dependent, but mostly standard once the other natural kind specifications are given. To show how it is defined we need the concept of *species*. The *species* of an object is the smallest natural kind to which it belongs. The existence monitor is then constructed according to the following rules:

1. For each species $\mathcal{A}$ there are implicit declarations in *ExistenceMonitor* of the attributes

   $ext\text{-}\mathcal{A}$: DB → IDS

   $old\text{-}\mathcal{A}$: DB → IDS

   $next\text{-}\mathcal{A}$: DB → NK

   These are going to have as values the current extension of $\mathcal{A}$, the set of objects which have existed, and the next identity to be created of this natural kind.

2. For each extension attribute we assume equations of the form

   x in $ext\text{-}\mathcal{A}$(db) = x in *ext*(db)

   x in $old\text{-}\mathcal{A}$(db) = x in *old*(db)

   This means that to determine if an object of species $\mathcal{A}$ exists or has existed, we determine whether it is in *ext*(db) or in *old*(db), respectively.

2.1.4. Communication

Synchronization is the simultaneous execution of two or more events. **Communication** is the synchronization of events which would deadlock in an attempt to execute them outside the synchronization. In [Wie89b] more details are given. To specify a communication,

1. it must be specified that the communicating events *can* occur synchronously, and
2. it must be specified that the communicating events *cannot* occur on their own.

Events which must communicate with other events are called *messages*. Synchronization (and therefore communication) is either *internal* when all events (messages) are executed by the same object, or *external*, when two or more objects participate in the communication. Message-sending has no direction, but it has an *audience* which is the set of participating events.

Internal communication

Internal communication in the life of objects of kind $\mathcal{A}$ is defined as follows:

1. The set $H_{\mathcal{A}}$ of *internal messages* is defined. These events can only occur synchronized with other events in the life of the same object. No other events in the life of the object can synchronize with other events.
2. For all internal messages, the precise conditions under which they must synchronize, are given in terms of the internal state of the object and the values of the actual parameters.

External communication

External communications involve at least two different objects and must be specified in the conceptual model specification. This is done as follows:

1. A set H of *external messages* is defined. These must participate in a communication in which at least two different objects participate. Globally, all possible synchronizations of events not in H in the life of the different objects are possible, except
   1.1. synchronizations in which external messages occur (these synchronizations are subject to the condition under 2), and
   1.2. synchronizations in which one object executes two communications simultaneously.
2. The precise conditions under which external messages must synchronize are given.

External communications are used to formulate global state constraints which involve more than one object. In the University example we find the synchronization events *enroll_course* from *Student* and *enroll* from *Enrollment*. The section

external messages
enroll_course(s, c)

is added to the *Student* specification. This declares the external communication *enroll_course*(s, c) to be an external message, which means that in the domain process, it must communicate with an event executed by another object. In the conceptual model specification we add the external communication in which this event must participate:

enroll_course(s, c) | *enroll*(s, c)

Both, internal and external communications can have *pre-conditions*.

2.2. OBLOG

The **OBLOG** (**OB**ject **LOG**ic) language is first introduced in [Ser87]. It is used as an experimental vehicle for the analysis of several aspects of object-oriented approach to information systems specification and design, based on an algebraic/categorical theory of abstract object specification [Ehr89, Ser89]. There is a running prototype version, *OBL-89* described in [Cos89].

In the following we will give the definitions of the underlying concepts of Oblog.

2.2.1. Concepts of OBLOG

We will give the corresponding Oblog concepts to the standard concepts of section 1.2.1. The order of introduction of the concepts will not be always the same as in section 1.2.1. in order to have a better understanding of the relations between the concepts.

Attributes

In the available literature of OBLOG, we were not able to find a formal definition of attribute. A characteristic of algebraic languages is that one can always start at a certain level of specification, i.e. one can specify a set of concepts to be the basic concepts of the language which need no definition and from which all the other concepts can be defined. That is the case in OBLOG of *attribute*, *event* and others.

An attribute is said to describe the state of an object, and it ranges over a set of values (data_type, set or object). Such set of values is called the **type** of the attribute. Moreover, OBLOG has the notion of **valuation**, which is more or less equivalent to our standard notion of **state vector**; formally:

Def. Oblog.1 - Valuation

A **valuation** over a set of attributes $A = \{a_1,\ldots, a_n\}$ is a set of attribute-value pairs $y \subseteq \{(a_1{:}d_1),\ldots, (a_n{:}d_n)\}$ where $d_i \in type(a_i)$ for $1 \leqq i \leqq n$ and $type(a_i)$ is the range of a_i. The set of valuations over A (**state space** in our terminology) is denoted by $val(A)$. □

The set of attributes of an object is called, in OBLOG, the **state** of the object.

Events

State changes correspond to the occurrence of **events**. Again, as with attributes, there is no definition of event in OBLOG. In OBLOG there are three sorts of events: **modification**, which affect the observed values of attributes, **creation** and **destruction** responsible for creating and destroying the object.

Creation and destruction events are special events in the sense that, in the life cycle of an object there occurs only one birth event and one death event, at most. Nevertheless, an object can be born and die in the diversity of admissible states. In OBLOG there are the reserved words **birth** and **death** to indicate the birth and the death event, respectively. The birth (and death) of most objects in an OBLOG society is induced by other objects. However, there are objects which birth is independent of other objects but determined by external intervention. These objects are distinguished by the reserved word **user interface object**.

Modification, or evolvement events, describe the behaviour of an object. They happen during the life of the object between its birth and its death. Such events alter the state of the objects, i.e. modify the value of state variables.

The effect of an event on attribute values is expressed by *valuation rules*, which are formulae of the form

<condition> ⇒ ([<event>]<attribute> = <term>), or

[<event>]<attribute> = <term>

meaning that, in a state satisfying <condition>, or in all states for the second case, the result of the occurrence of <event> on <attribute> is expressed by <term>. For example, if we have an object *person* with events *birth* and *birthday* and attribute *age*, we can express the valuation rules:

$[birth]age = 0$

$[birthday]age = age + 1$

Valuation rules define the effects of events over attributes. With valuation rules no conditions are imposed to the occurrence of events. Conditions are set by the so called *safety equations*. For example,

$\{age \geq 16\}\ get\_a\_job$

imposes a condition to the occurence of event *get_a_job* of persons, namely, someone can only get a job if s/he is older than 16. Safety equations will be discussed in more detail later.

Life cycles

Processes in Oblog are called **life cycles** and are defined as follows:

Def. Oblog.2 - Life cycle

If X is a set of events, $X^{\sigma} = X^{*} \cup X^{\omega}$ a set of **life cycles**. Where X^{*} is the set of finite sequences over X and X^{ω} the set of infinite sequences. Finite sequences of events are called **traces**.

Thus, OBLOG allows both finite and infinite processes.

Objects

An object has an internal **state** that can be observed. State changes correspond to the occurrence of **events** and the state can be observed through **attributes**. So, an object is given by its sets of events, attributes, life cycles and valuations. Summing up, OBLOG formally defines an object as:

Def. Oblog.3 - Object

An **object** is a tuple $<X, A, \Lambda, \alpha>$ where

X is a set of **events**

A is a set of **attributes**

$\Lambda \subset X^{\sigma}$ is a set of **life cycles**,

α: $X^{*} \rightarrow$ val(A), is a **valuation mapping** that maps each trace in X^{*} into a valuation of attributes in A. □

Such definition is said to follow a *process-oriented* model. An alternative model identifies the objects with a state machine and categories of objects and object types are defined [Ehr89]. The two models are said to be equivalent.

In Oblog the abstract syntactic specification of an individual object is illustrated by the

following example:

object *person_john*
events
birth_john;
birthday_john;
get_a_job_john
attributes
age_john: *pos_int*
life cycles
safety
{*age_john* ≧ 16} *get_a_job_john*
valuation
[*birth_john*]*age_john* = 0;
[*birthday_john*]*age_john* = *age_john*+1
end

Life cycles usually are not described in a processlike form but through **safety** conditions. These define the occurrence of an event in terms of other events and/or attribute values. Safety conditions are also called *enabling rules* and denoted by $\{c\}e$, meaning that the event e can only occur when condition c is true. We will come back later to this subject.

Valuation mappings are specified by giving the value of each attribute after the occurrence of an event and not, as the definition says, mapping traces into valuations. These terms are called *event attribute terms* and are denoted by $[e]a = x$ meaning that the value of attribute a after the occurrence of event e is x.

Classes

In OBLOG **type** is defined as being a set of "similar" objects. In our standard definition type is defined as the form of a set of objects (which we call the class). In Oblog such difference between the intension and the extension of a set of objects is not explicitly made. So, in OBLOG, type means both our standard notions of class and type.

We give now the definition of **object type** in OBLOG:

Def. Oblog.4 - Object Type

An **Object type** *ot* is a tuple $<|ot|,\omega>$, where $|ot|$ is the **surrogate space** and ω is a mapping from $|ot|$ into *OB* (category of objects), called the **template mapping**, such that:
$\forall\ u \in |ot|$ $\omega(u)$ is called the **occurrence template** of the occurrence with surrogate u. The **occurrence** corresponding to u is the object $u.\omega(u)$, where the operation "." is defined as follow:
$u.<X,A,\Lambda,\alpha>$ is the object $<u.A,u.X,u.\Lambda,u.\alpha>$ where $u.X = \{u.x: x \in X\}$ and so on.□

Given an ocurrence of the surrogate space, the correspondent object is obtained by applying the template mapping to the surrogate. To see how this definition, is applied, consider the following example of an abstract specification of homogeneous types:

homogeneous object type *Person*

surrogate
|PERSON|
template
events
birth;
birthday;
get_a_job
attributes
age: pos_int
life cycles
safety
$\{age \geqq 16\}$ *get_a_job*
valuation
$[birth]age = 0;$
$[birthday]age = age+1$
end

If we assume that *john* is a surrogate in surrogate space *|PERSON|*, the occurrence corresponding to *john* is the object, $john\omega(john) = john.<X,A,\Lambda,\alpha>$, where $X = \{birth, birthday, get\_a\_job\}$, $A = \{age\}$, $\Lambda = \{\{age \geqq 16\}\ get\_a\_job\}$ and $\alpha = \{[birth]age = 0, [birthday]age = age+1\}$. Which is the definition of an object (cf. def. Oblog.3).

Note that the template mapping is not necessarily injective. On the contrary, in many cases it is even constant. In the case that ω is constant the type is said to be **homogeneous**. Occurrences of a homogeneous type are *similar* in the sense that they are "isomorphic copies" of a unique object, called the **type template**. This notion of type template is more equivalent to our standard notion of *type*, i.e. it defines the 'form' of the objects belonging to a homogeneous type. Members of **heterogeneous** types may different templates. Heterogenous object types are a consequence of *generalization*, which will be explained later.

The surrogate mechanism of OBLOG is quite complicated and it will be discussed in subsection 2.2.2. Let us for now just assume that *|ot|* provides the means to identify objects within a type.

Liveness and safety versus process description

When introducing the definition of object, we said that in OBLOG life cycles are described by safety equations rather than by process definition. This is true in most of the examples we are aware of, but in the last publications about OBLOG (cf. [Cos89]) an alternative description, using CSP, is given.

Thus, the description of the behaviour of an object can be achieved, in OBLOG, in two different ways, which do not exclude each other:

(a) Declarative - the behaviour is described through safety and liveness requirements.

(b) Procedural - the behaviour is described through a CSP process.

Safety requirements are written in a first-order language with predicates relative to state variables and precedence of events, enriched with the temporal operators **P** (sometime in the past) and **H** (always in the past), with the following syntax:

{<expression>}*event*

Expressions can either have incidence on attributes or on events. In the case of expressions built on events there are special predicates **after** and **before** to solve the problems of precedence, mainly associated to operator **P**. In the rest of this section we will show many examples of safety rules.

Safety properties establish that no undesirable behaviour, i.e behaviour generating traces not satisfying the given property, is possible. Note that in most cases, properties of traces hold for the empty trace, i.e. a trace of all deadlocked objects because traces are closed under prefixing [Bri89]. To guarantee that something desirable will happen, i.e. to establish *liveness* properties, more is required. That is we must require that some events will occur in any possible trace. The declaration of liveness requirements in OBLOG can take the following forms:

a) the single form

e_1

which semantics are that e_1 occurs in any possible trace, which means that the object can only die after the indicated event occurred.

b) the conjunctive form

e_1 & ... & e_n

meaning that all the specified events must occur in any possible trace.

c) the disjunctive form

e_1 | ...| e_n

meaning that at least one of the specified events occurs in any possible trace.

d) the mixed form

e_{1_1} &...& e_{1_n} |...| e_{m_1} &...& e_{m_n}

meaning that at least one of the conjunctions of events must occur in any possible trace.

e) the conditional form

[<condition>]<*events*>

where <*events*> represent a set of events in one of the previous forms (single event, conjunction of events,...). The semantics of such declaration are that, the liveness requirements are valid from the moment the object moves to a state satisfying the given condition.

In the procedural case the life cycles are are defined by expressions of the language CSP[2] [Hoa85] for describing processes. In CSP events are the alphabet of the language and processes are defined using the alphabet of events and the operators:

prefixing, $(x \rightarrow P)$, meaning the process formed by event x followed by process P,

choice, $(P \mid Q)$, meaning process P or process Q,

recursion, $(P = x \rightarrow P)$, representing the repetitive behaviour of an object, and

concurrence, $(P \| Q)$, denoting the process composed by both process P and Q, which interact in some events.

2- The advantages of CSP are that it represents the regularity and cyclicity in the behaviour of objects and it represents the causal structure intrinsic to the sequence of events, i.e. the preconditions to the occurrence of an event is satisfied in previous occurrences of other events.
The problems of CSP are that processes which don't show regularity can not be represented and also the state of the described object(s) can not be represented, i.e. its state variables don't appear in the description.

more details about CSP will be introduced, by example, in the sequel.

Often, in combination with CSP, we also use safety requirements, namely in the case of a choice, developing a language of guarded events. The syntax adopted, by OBLOG, to represent procedures is:

procedure

<*birth-event*> → <Process>

where

<Process>

where <Process>, the process which is executed, contains only the operators of sequencing and choice.

Note that in the definition of a CSP procedure within a type definition, the operator of concurrence never (as far as we know) occurs. That is due to the fact that in OBLOG, interaction between different processes, objects, is described in a separate section, using a language which is not pure CSP. Interaction between object will be described in section 2.2.4.

In a more general case, a process declaration is:

procedure <process_term> [**where** <process_declaration_list>]

where <process_declaration_list> is the declaration of a list of processes. A process term can have one of the following forms:

a) an event: the clause **procedure** followed by only one event, replaces a liveness requirement for that event.

b) a process identifier: the process is declared under the clause **where**

c) a choice: the most general form of choice is

<choice> = $(\{c_1\}e_1 \rightarrow t_1) | \ldots | (\{c_n\}e_n \rightarrow t_n)$

where t_i are process terms.

To illustrate that both methods, declarative and procedural are equivalent, consider an object template with events b (birth), e_1, e_2 and e_3, and the safety constraints:

life cycles

safety

$\{c_1\}e_1$;

$\{c_2\}e_2$

This situation can be, alternatively, described in terms of processes, as follows:

life cycles

procedure

$b \rightarrow P$

where

$P = (\{c_1\}e_1 \rightarrow P) \mid (\{c_2\}e_2 \rightarrow P) \mid (e_2 \rightarrow P)$

The main difference between the declarative and the procedural methods of describing the life cycles of an object, is that, in general, the declarative way expresses a non-deterministic process. Indeed, with the process operators used by OBLOG, we can only express deterministic process, whereas the in the declarative way no sequence of processes is pre-determinated.

Taxonomy and Inheritance

OBLOG supports the notion of specialization and generalization, as well as other ways to relate

different objects, as through a link. OBLOG has rather different ideas about the formalism of such notions. This will be the subject of subsection 2.2.3.

Communication

In OBLOG, communication between objects is done by **event sharing**, which means that the life cycles of two different objects must be synchronized in one event. More details about communication will be given in subsection 2.2.4.

2.2.2. Identification

Identification of objects in OBLOG is done within object types. As we saw part of the definition of an object type corresponds to a **surrogate space**. Formally, a surrogate space is a non-empty set. In practice, though, OBLOG provides means to define a such set for each object type. Taking for instance the type *Person*, one might wish to generate a different surrogate to each string, representing the name of a person. In fact the construction of the surrogate space is done using a **key mapping** (giving the object corresponding to a surrogate) and a **key attribute** (giving the surrogate corresponding to each object). A *parameterized data type* of surrogate space constructors is introduced as follows:

parameterized data type *construct(dt1,dt2)*
  operations
    $key\_map: dt1 \times dt2 \rightarrow construct(dt1,dt2)$;
    $key\_att1: construct(dt1,dt2) \rightarrow dt1$;
    $key\_att2: construct(dt1,dt2) \rightarrow dt2$
  equations
    $key\_att1(key\_map(N_1,N_2)) = N_1$;
    $key\_att2(key\_map(N_1,N_2)) = N_2$;
    $key\_map(key\_att1(U),key\_att2(U)) = U$;
end

Variants of this parameterized data type for the cases of 1,...,n argument data types, are possible. For instance for the previous example (name of *Person*), case of one argument, it can simply be written: (more "syntactic sugar")

data type |*PERSON*|
  construct(name:string)
end

The surrogate data type |*PERSON*| is obtained by applying the (unary) parameterized data type *construct* to the argument data type *string*, *renaming the key attribute to name*. The key mapping (that maps strings into person surrogates) is *PERSON*. Thus the previous (incomplete) abstract specification of the object type *PersonR is now as follows:*

homogeneous object type *Person*
  surrogate
    construct(name:string)
  template

as before

end

2.2.3. Relationships between objects

In Oblog we have two basic ways to relate objects (or object types): through object morphism (subobjects and their generalization linked objects) and through object composition.

Subobjects

Subobjects (subtypes) in OBLOG correspond to our standard notion of specialization (eg. employee is a subtype of person). In order to allow "side effects" of events from the subobject in attributes of the parent object, one must be able to fit the subobject properties within the life cycles of the parent object. Although, specification-wise it is correct to look at the subobject as inheriting the attributes and events of the parent object, with respect to operational issues it is necessary to take the dual perspective: the parent object incorporates the new attributes and events of the subobject as an "aspect" of itself.

Def. Oblog.5 - Subobject

The object $ob_1 = \langle X_1, A_1, \Lambda_1, \alpha_1 \rangle$ is said to be a **subobject** of the object $ob_2 = \langle X_2, A_2, \Lambda_2, \alpha_2 \rangle$ iff

(a) $X_1 \subset X_2$

(b) $A_1 \subset A_2$

(c) $\Lambda_1 \leqq \Lambda_2$

(d) $\forall\, t \in X_2^*$, $\alpha_2(t)\downarrow A_1 = \alpha_1(t\downarrow X_1)$, where $\downarrow A_1$ and $\downarrow X_1$ denote the restrictions to the attributes and to the events, of ob_1, respectively. □

In more detail, life cycle inheritance expresses the fact that each (possible) life cycle of the object ob_1 (eg. an employee) must be contained, distribuitively, in some (possible) life cycle of ob_2 (eg. a person). Valuation inheritance says that any ob_2 (person) trace restricted to an ob_1 (employee) trace by hiding the additional events, gives rise to the same valuation in ob_1 and ob_2 when considering only the attributes of ob_1.

There are two possible situations concerning the birth of a subobject: either the birth event of the parent object is also the event that originates a replica in the specialization object (they are created at the same time) or the is an evolvement event in the parent object that originates a specialization objects (role playing). With respect to the destruction of a subobject there are also two possible situations: there is either a death event in the parent object which simultaneously destroys the subobject, or a death event in the subobject which destroys it (without destroying the parent object).

Considering object types, **subtype** is defined as follows:

Def. Oblog.6 - Subtype

The object type $ot_1 = \langle |ot_1|, \omega_1 \rangle$ is a **subtype** of $ot_2 = \langle |ot_2|, \omega_2 \rangle$ iff

(a) $|ot_1| \subset |ot_2|$

(b) $\omega_1(u)$ is a **subobject** of $\omega_2(u)$, $\forall u \in |ot_1|$. □

An example of a subtype is as follows:

homogeneous object type *Employee*
type view of *Person*
template
events
become_employee;
quit_work;
one_more_year(naturals);
increase_salary(naturals)
attributes
salary: naturals;
years_work: naturals
(...)
end

Linked-objects

Formally, **linked object** is a generalization of subobject. The events and attributes of the components of a link are not the "same" (in the sense of inheritance in subobjects) but "give rise to corresponding" events and attributes in the linked-object. The link is established to mappings (morphisms) of objects into objects. A special case of such mapping is inclusion (as it appears in subobjects).

For example consider students, courses and enrollments. In fact, we may expect that a student, a course and the enrollment of that student in that course will not have independent behaviours as objects. In this case we do not have an inclusion relationship (we cannot say that *Enrollment* is a *Student*, or vice versa!) bur a link relationship.

The semantics of a link between two objects, say, O_1 is linked to O_2 are: a) for any life cycle of O_2 there exists a life cycle of O_1. Any behaviour of O_1 is a possible behaviour of O_2. However, O_2 is allowed to have certin behaviours which are not posiible to O_1, and, b) to every attribute of O_1 corresponds an attribute of O_2 with the same range.

The formal definition is:

Def. Oblog.7 - Linked-object

The object $ob_1 = <X_1,A_1,\Lambda_1,\alpha_1>$ is said to be **linked** of the object $ob_2 = <X_2,A_2,\Lambda_2,\alpha_2>$ iff there are

(a) an **event mapping** $h_X : X_1 \rightarrow X_2$

(b) an **attribute mapping** $h_A : A_1 \rightarrow A_2$

(c) such that $h_X(\Lambda_1) \leqq \Lambda_2$

(d) and $\forall\, t_1 \in X_1^*$ and $\forall\, t_2 \in X_2^*$, if $h_X(t_1) = t_2 \downarrow h_X(X_1)$ then $\alpha_2(t_2) \downarrow h_X(A_1) = h_X(\alpha_1(t_1))$.

□

In the same way, we can define linked-types, T_1 is linked to T_2, with semantics: a) there exists a *surrogate mapping* between the space of identities of T_1 and the space of identities of T_2, and b) a *template mapping* that, to attributes and events of T_1, makes correspond attributes and events of T_2. Formally:

Def. Oblog.8 - Linked-types

The object type $ot_1 = <|ot_1|,\omega_1>$ is **linked** to $ot_2 = <|ot_2|,\omega_2>$ iff there are

(a) A **Surrogate mapping** $|U|: |ot_1| \rightarrow |ot_2|$ and

(b) A **instance mapping** $\underline{U}(u): u.\omega_1(u) \rightarrow |U|(u).\omega_2(|U|(u)), \forall u \in |ot_1|$. □

This is the general form for **object type morphism** from which subtype is an special case. Instances of linked-types are linked-objects.

When ot_1 and ot_2 are both homogeneous, a link between them can be defined in a simpler way: besides $|H|$ it is enough to establish $\underline{H}:ot_1 \rightarrow ot_2$. In this case H is said to be a **homogeneous type morphism**. Moreover, when $|ot_{11}| \subset |ot_2|$ and $\underline{H}$ is an object inclusion morphism we get a subtype morphism as introduced before.

We introduce the sintax of linked types, with an example:

homogeneous object type *ENROLLMENT*
  type linked to *STUDENT*
    surrogate map
      stud
    instance map
      enroll(|*COURSE*|) **to** *enroll_course*(|*COURSE*|);
      issue_grade(|*COURSE*|,int) **to** *get_grade*(|*COURSE*|,int)
  type linked to *COURSE*
    crs
  surrogate
    construct(*stud*:|*STUDENT*|,*crs*:|*COURSE*|)
  template
    events
      enroll(|*STUDENT*|,|*COURSE*|);
      issue_grade(|*STUDENT*|,|*COURSE*|,int);
      term;
      term_remind
    life cycles
      safety
        for S: |*STUDENT*|, C: |*COURSE*|, G: int;
        {P*term*}*issue_grade*(S,C,G);
        {P*term* and not(P*issue_grade*(S,C,G)}*term_remind*
end

We establish a link from *ENROLLMENT* to *STUDENT* and a link from *ENROLLMENT* to *COURSE*. We explain the link to *STUDENT*: the surrogate mapping is the key attribute *stud* and the template mapping maps the event $e.enroll(c)$ of the enrollment with surrogate e into the event $s.enroll\_course(c)$ of the student such that $s = stud(e)$, as well as the event $e.issue\_grade(c,x)$ into $s.get\_grade(c, x)$.

Composite objects

The last way to connect objects in OBLOG is the **composition**, which from two different objects gives a composite object containing both of them. The composite-object is called an **aggregation** of the two original objects. With respect to attributes and events, the resulting composite object will have all attributes and events of both parts. Moreover, each possible life cycle of the

composite object must be an interleaving of a life cycle of each of the parts. Finally, the valuation mapping of the composite object will provide the same value as the mapping of each part when only events of that part are considered.

Def. Oblog.9 - Composite-object

The **aggregation** $ob_1 \| ob_2$, of two independent objects $ob_1 = <X_1, A_1, \Lambda_1, \alpha_1>$ and $ob_2 = <X_2, A_2, \Lambda_2, \alpha_2>$ is $<X_1 \oplus X_2, A_1 \oplus A_2, \Lambda_1 \| \Lambda_2, \alpha_1 + \alpha_2>$, where

(a) $X_1 \oplus X_2$ is the disjoint union of the event sets

(b) $A_1 \oplus A_2$ is the disjoint union of the attribute sets

(c) $\Lambda_1 \| \Lambda_2$ is the interleavings of the life cycles of the parts

(d) $\alpha_1 + \alpha_2$ is the 'sum' of the components valuation mappings, i.e. $\forall\ t \in X_1 \oplus X_2$, $(\alpha_1 + \alpha_2)(t)\downarrow_{A_i} = \alpha_i(t\downarrow_{A_i})$, with i=1,2. □

The definition of **generalization-type is as follows:**

Def. Oblog.10 - Generalization-type

Considering the object types $ot_1 = <|ot_{1|}, \omega_1>$ and $ot_2 = <|ot_{2|}, \omega_2>$ the **generalization type** of ot_1 and ot_2 is $ot_1 \oplus ot_2 = <|ot_{1|} \oplus |ot_{2|},\ \omega_{ot_1 \oplus ot_2}>$. I.e the surrogate space of the generalization-type is the disjoint union of the argument spaces and its template mapping is defined as follows:

1. $\omega_{ot_1 \oplus ot_2}(\text{inj1}(u)) = \omega_1\ \forall u \in |ot_{1|}$ and
2. $\omega_{ot_1 \oplus ot_2}(\text{inj2}(v)) = \omega_2\ \forall v \in |ot_{2|}$

where inj1 and inj2 are the injections from $|ot_{1|}$ and $|ot_{2|}$ respectively, into $|ot_{1|} \oplus |ot_{2|}$. Note that every z in $|ot_{1|} \oplus |ot_{2|}$ is either in inj1($|ot_{1|}$) or in inj2($|ot_{2|}$). □

In general the generalization-type is **heterogeneous**, even if the parts are homogeneous. Generalization can be easily extended to the n-ary case. An example of a generalization type is:

heterogeneous object type *Thing*
  generalization*(Person, Car)*
end

2.2.4. Communication

As we already have said, communication is done in OBLOG by **event sharing**, mainly. Event sharing is made by introducing an auxiliary object linked to the objects which want to communicate. This new object has no attributes, it contains only events. These are the events which the objects share.

Suppose that we have to objects Person_john and Person_mary and that Person_john wants to communicate with Person_mary by sending her a letter. So there is an event *john_send_letter* from Person_john and an event *mary_receive_letter* in Person_mary which have to synchronize (if we exclude the mailing procedure...).

So we introduce an auxiliary object:

object auxiliary
  linked to Person_john
    mail_letter(L) **to** *john_send_letter*(L)

linked to Person_mary

mail_letter(L) **to** *mary_receive_letter*(L)

events

mail_letter(L)

end

The defined links impose:

∀L *john_send_letter*(L) = mary_receive_letter(L)

In general it is possible that two objects share other things than events (eg. attributes and whole objects). In any case an object is shared, like the auxiliary object above. As event sharing is so common in OBLOG it is possible to introduce it in a rather abbreviated way through **interaction equations**, which are taken purely as "syntactic sugar".

For instance, taking the given example we can say that, the specification of the previous interaction, using interaction equations, is:

interaction among *Person_john*, *Person_mary*

john_send_letter = *mary_receive_letter*

Society

Communication is expressed formally using the concept of **society**.

A society is a, typically large, collection of objects which interact in one way or another [Ehr89]. In the OBLOG model there are three ways of interaction between entities: sharing of events, sharing of attributes and sharing of (component) objects. Attribute sharing is common in databases. In fact it is the basis for the natural join operation of relational databases, which in turn is the foundation of relational database design. In object-oriented systems, event sharing is usually preferred. Attribute sharing means to share memory space which is in contraction with the locality principle of object encapsulation. Event sharing is a general way to express synchronous communication. Oblog favors a general an symmetric form of event sharing: any event can be shared between any number of objects.

OBLOG views both attribute and event sharing as special cases of object sharing[3]. The right place to study object sharing is in the greater context of a composite object where components share components. For example if the event of buying a car is shared by the car, the dealer-shop and the buyer, there should be an object like *"car_market"*, of which car, dealer-shop and buyer are components. Such objects are **societies**. An object society is an object instance constructed from object types by adding interaction information, employing the operation of parallel composition.

Consider our study case, university. When specifying the object society of an university in OBLOG, we first define the object types involved, say, *Person*, *Student*, *Teacher*, *Course*, *Enrollment*, etc. By generalization, we consider all instances of all these types assembled together in one type, say *University*. We can now introduce *global* events (i.e. events shared by different objects) as *local* events to the society object *University*.

3- Note that an event e can be viewed as an object $(\{e\},\emptyset,\{\epsilon\},\lambda x.\emptyset)$, and a single attribute a can be viewed as an object $(\emptyset,\{a\},\{\epsilon\},\alpha_\epsilon)$, where ϵ is the empty life cycle (i.e. expresses that the object remains non-existent) and α_ϵ is the null valuation mapping.

2.3. MOKUM

MOKUM (Manipulating Objects and Knowledge with Understanding in Mokum), is a knowledge base system under development at the Department of Computer Science of the Free University Amsterdam [Rie89]. (Mokum is also a nickname for a historical part of Amsterdam.) A prototype version is being developed at the moment [Dig89].

In a MOKUM system, knowledge is represented in the form of **objects** and a **conceptual model** (CM). The objects are stored in relations in a relational database system. The CM is represented in the form of Prolog rules, which are also stored in the database.

2.3.1. Concepts of MOKUM

Conceptually, objects represent entities of the UoD, they are able to send and receive messages, change their state and type, create and delete other objects, under the protection of formally specified access rules. The conceptual model defines both the **static structure** and **dynamic behaviour** of the objects. It uses inference techniques and is capable of reasoning. In the following an overview of the most important notions of MOKUM is given.

Object

An **object** is a unique 'thing' represented in the system and it represents (usually) an entity of the real world. Its internal representation has the form of a surrogate which cannot be transferred to the external world.

Class and Type

Objects may be considered instances of **types**. A **type** defines the static structure and dynamic behaviour of a group of objects. A **class** is the set of all objects of a certain type. There is a one to one correspondence between types and classes. I.e. a type defines the *intension* of a group of objects and a class reflects the *extension* of the same set. (In the following types will be referred to by singular nouns and the correspondent class by its plural.)

The static structure realizes the notion of **aggregation** of attributes. It has a 'record-like' appearance. The dynamic behaviour is defined in the form of a **script**, defining a finite state automaton. Types can be related to other types through the familiar is_a hierarchy realizing the notion of **specialization/generalization.** There is a most general class **things**, from which all other classes are a specialization. The only property of type **thing** is the identity property in the form of an internal defined surrogate, which is inherited by all types, through the is_a relationship.

Collection and class

A **collection** in MOKUM realizes the notion as *group-object*. Collections permit a set of objects to be used as a whole, i.e. as as a single entity. A collection of objects can be used as the value of an attribute. However, elements of a collection are always of the same type, and no attributes can be defined to a collection. An important aspect of collection is the concept of **collection keeper**. The collection keeper is a special object in charge of controlling what is going on in the collection. The keeper has a type, say, *keeperT*. This type has at least one attribute *coll*, the correponding collection of objects, and may have additional attributes, such as *number_of_members* or *avg_salary* (in the case that we are considering a collection of employees!). Using the collection keeper it is, thus, possible to define attributes ranging over a set of objects, such as *avg_salary*.

When a collection contains all the objects of a certain type one speaks of **class**. Moreover in general collections are subsets of classes.

Attribute

Attributes are tuples <*attribute_name,attribute_type*>. Attributes are always defined within a type definition and are applicable to all the objects of that type.

An attribute_type is one of three things:

1. a data type (integer, string, real ,etc.)
2. an object type
3 a collection

Attribute types can be **defined** or **derived**. In the former case the values are primary data such as, for the type *Person*, *name* or *address*. Such values can be subject to **restrictions**, by the definition of Prolog rules (also in the procedural part) which restrict the possible values for the attribute to a subset of the indicated data type.

In the case of derived attributes, the value of the attribute for a certain object is calculated, through a Prolog function, from, e.g., other attribute values. For example the type *Person* may have the attribute *age* which is a derived attribute from the attribute *birthday* and the current date. Derived attributes are not subjected to restrictions. I.e. restrictions which may involve derived attributes are placed in the attributes from which it is derived.

The syntax of the definition of the static part of a type is given in the following example:

```
Type personT
    has_a name:string
    has_a address:string
    has_a birthday:integer
    restriction birthday:birthday_restriction_on_person
    has_a age:integer:=compute_age(birthday)
    has_a children:collection(PersonT)
    proc
        birthday_restriction_on_person:-
            birthday > 18500000,
            birthday ≦ current_date.
        compute_age(birthday):-
            age is (current_date - birthday)/10000.
    endproc
```

Dynamics of objects

An object is created using the predicate **new** and it can later on be deleted with the predicated **destroy**. During its life an object can get new types and lose types by means of **add_type** and **remove_type**. However, the types an object can get during its life must be related with the ones an object already has by means of a specialization relationship. If eventual restrictions defined to a type are not verified by an object we are trying to add to the type, then such operation is not possible to execute and a error message occurs.

Objects are **active**. An object can **send** and **receive messages**, it can create or destroy other objects, it can change its types. At each moment, for each type it has, an object can be in one or more **states**, called the *active states*. Together, for all the types an object has, they form a **state set**. If the state set is empty the object is static, it can't do anything. Otherwise, the object is active. If an object is static, its attribute values can be changed by other objects which can access it, however.

The description of the life process of objects is called a **script**, which is coupled to a type. Thus, all objects which have a certain type, perform the same script.

By adding a type to an object, automatically a new process is started to the object, corresponding to the script of the type. The state of the object is then **initial**. The state of an object changes when a certain **condition** is verified. We say then that a state **transition** happens. Transitions are either triggered by **messages** from other objects, or by wake_up messages at preset times. Scripts are also objects, of type *Script*.

We give, in the following an example of a script for the type *Student*:

```
script
    initial state
        action: assign ∅ to courses
        next_state in_life
    state in_life
        at_trigger enroll_course(self, C)
            action: check_if_can_enroll; assign C to courses
            next_state in_life
endscript
```

2.3.2. Identification

Internally, objects are identified by means of a **surrogate**, which is a unique identifier and is not accessible from the outside. The Prolog rule,

new(X), **new**(Y).

will create new objects and associate internal identifiers to the Prolog variables X and Y. Note that in Prolog, the scope of variables is the rule within which they are used[4]. In the case of the predicate **new**, a value will remain in the internal database, the surrogate of the newly created object. Moreover, such internal values are not accessible from the outside.

In the previous example the newly created objects would, thus, remain in the system but it wouldn't be possible to access them any more after the rule has been executed! The normal MOKUM procedure is then to associate the newly created objects with a type and use a subset of the values of the attributes to identify externally the object, afterwards. For example

new(X), **add_type**(X,personT,[(name,"John"),(bjrthday,19650429)]).
new(Y), **add_type**(Y,personT,[(name,"Mary"),(birthday,19450705)]).

In MOKUM the user can define his/her own predicates to access the database. In [Dig89] a suggestion is made of a a user-defined predicate incorporating both the predicates **new** and **add_type**.

4- See for example [Clo81].

MOKUM does not have the concept of *key* as a way to identify objects externally. Any subset of attributes which apply to the object can be used as a identification for it. Two objects with exactly the same attribute values will then not be externally distinguishable, even if they have different internal surrogates. However, within a collection keys can be defined. The philosophy is that keys are properties of sets of objects (classes) and not of the structure of objects (types), and, thus, can only be defined for a set and not for a type.

2.3.3. Relationships between objects

Objects can be grouped in classes. Classes are structured into an IS_A hierarchy. Moreover, objects can dynamically enter and leave subclasses. In MOKUM specialization is not static, in a way that if an object is member of a subclass at a certain moment, it will remain member of such class through all its life, but dynamic. Such entering and leaving of classes is left to the designer. In certain cases it can be wise to avoid such property and, then it must be enforced as a constraint over the subclass. We say that MOKUM supports the notion of **role playing**.

As one would expect, members of a subclass, inherit the type and the script of the parent class(es). Moreover the constraints can be stronger for the members of a subclass. For example, if we define the subclass *studentT* of the class *personT*,

Type *studentT* **is_a** *personT*
has_a *student_number*:integer
has_a *major*:string
has_a *year*:integer
restriction *birthday*:birthday_restriction_on_student
proc
birthday_restriction:-
birthday > 19600000;
birthday ≦ current_date - 150000.
endproc

An interesting question is whether that the restriction for *studentT* is indeed stronger than the one for *personT* can be automatically checked, so that on giving a value to birthday only the *studentT* restriction needs to be checked. It must always be verified the following relationship:

birthday_restriction_on_student ⇒ birthday_restriction_on_person

A collection is also a way of relating objects. However, as we already said, members of a collection must always be of the same type and one cannot define attributes which apply to the collection as a whole. Collections are then particularly useful as values for attributes of another object. One can then have the attribute *children_of* of *Person* representing the set, possibly empty, of the children of a certain person. Properties of a collection can be calculated at run-time using, for example user-defined predicates. For example the user can define the predicate *avg_age_of_children_of* to be applyed to the current collection *children_of*.

2.3.4. Communication

As we already saw, objects in MOKUM communicate by sending each other **messages**. Messages have then a sender and a recipient. In the script of the recipient object an action must be defined to perform **at_trigger** M, being M the received message. Otherwise, nothing is done and an empty answer is sent back to the sender. Note that it is not an error but an idle step.

2.4. TAXIS

Taxis [Myl80, Bor84] is based in the notion of generalization/specialization, which is applied uniformly to the various components of a Information System, such as data classes, transactions, exceptions, and user interfaces.

2.4.1. Concepts of TAXIS

There are three types of *objects* (not in the same sense we have been using but in the sense of entities) in Taxis: **tokens**, **classes** and **metaclasses**. All the three types of objects can have **properties**.

Tokens

Tokens represent constants. This notion of **token** refers both to data values and to *objects*, or better saying, to surrogates of entities. Thus, *john_smith*, representing the person (object) called John Smith, "JOHN SMITH", representing the string (data value) JOHN SMITH, and 35 representing the natural number (data value) 35, are all tokens in Taxis.

Properties

Classes and tokens have **properties** through which they can be related to other classes and tokens. Properties are the equivalent in Taxis to our notion of attribute. Properties are triples *<subject,attribute,property_value>*. For tokens properties represent specific facts (eg. "*john_smith*'s name is 'JOHN SMITH'") while for classes properties represent abstract rules (eg. "each person has a name").

For example, the class *PERSON* has the property

<PERSON,name,PERSON_NAME>

and the token *john_smith* has the property

(john_smith,name,'JOHN SMITH').

To avoid ambiguities, no two properties can have the same subject(s)[5] and attribute. I.e. attribute (or property name) is not a unique identifier for a property but the combination of subject(s) and attribute is. Properties with more than one subject are called *complex* properties. For example, the property

<(NAME, ADDRESS), person_id, PERSON >

is a complex property with two subjects, *NAME* and *ADDRESS*, and *p*_value the class *PERSON*. For each pair of name and address it gives the correnponding person, if any.

Properties of a class, i.e. properties which subject is a class, provide information about the *structure* of the instances of the class, and are called **definitional properties**. On the other

5- In TAXIS, the subject of a property is either a type or an object.

hand, properties of tokens, i.e properties which subject is a token, specify the structure of the token itself, and are called **factual properties**. There is a strong relation between the definitional properties of a class and the definitional properties of its instances. Such relationships can be expressed by the following principle:

> **Property Induction Principle.** The definitional properties of a class induce the factual properties of its instances.

Some confusion can arise from the previous, because one single Taxis concept stands for several of our standard concepts. For example, properties represent both attributes of class and attribute values of objects. I.e. Taxis uses the same syntax to deal with semantically different concepts.

The property_value (or *p*_value) is always single valued. I.e. Taxis does not accept sets as values for properties. Objects, in our standard sense, i.e. instances of variable classes, can be the value of a property.

Taxis distinguishes three categories of properties:

1. **keys**, (to be discussed later, in the subsection related with identification of object in Taxis),
2. **characteristics**, which are time invariant properties, and
3. **attribute_properties**, which are time variant.

Classes

Classes are collections of tokens sharing common characteristics. The collection of all tokens which are instances of a class C is called **extension** of C. If a token is element of a class we say that the token is an *instance* of the class. Examples of Taxis classes are *PERSON*, which instances are tokens such as *john_smith*, representing persons, and *PERSON_NAME* which instances are strings (tokens) such as 'JOHN SMITH'. The former can be identified with our standard notion of class whereas the latter is our notion of data type.

In fact, the Taxis notion of class is wider than the one we have been considering. The reason is that Taxis tries to give a uniform treatment to different concepts. Actually, we have in Taxis, three major kinds of classes:

> **Variable classes:** Have the special feature that their extensions can be altered by insertion and removal of objects. Also they can queried (in a QUEL-like fashion) and their instances can be updated. Moreover, variable classes are the only classes which can have key attributes (see subsection 2.4.1. for details on keys). I.e variable classes behave more or less like relations in a relational model and are the closest Taxis notion to our standard notion of *class*.
>
> **Test_defined classes:** Instances of test_defined classes can be referenced but never created or destroyed. This kind of classes is similar to our notion of *data type*. Moreover, although it is never said in the available literature, we think that test-defined classes can never be constructed from variable classes, which is one more reason to find them close to our notion of data type. Within test_defined classes we have three subkinds of classes:
>
> > **1. Aggregate classes:** The extension of an aggregate class, *A*, is determined at all times by the cross product of the extensions of the classes which are the *p*_values of *A*'s characteristic properties. As far as we could see these are the only properties that an aggregate class can have. For example the aggregate class

[|*street:STREET, no.:STREET_NO, zip: ZIP, city: CITY*|]

is the cross product of the extensions of *STREET, STREET_NO, ZIP* and *CITY*.

2. Finitely defined classes: Has a finite, time invariant extension. For example

{|'AMSTERDAM','LISBON','LONDON','PARIS'|} and

{|1::500|}

are finitely defined classes which instances are the strings 'AMSTERDAM', 'LISBON', 'LONDON' and 'PARIS' and the natural numbers between 1 and 500, inclusive, respectively. Finitely defined classes are similar to Pascal's scalar types.

3. Formatted classes: Have as instances all 'strings' which are consistent with a particular pattern. For example the extension of the formatted class

{|'('|}@REPEAT(DIGIT,3)@{|')'|}@REPEAT(DIGIT,7)

contains all the strings with format '(ddd)ddddddd' (phone values).

Transaction classes: Are classes used to model transactions, i.e. they are used to describe events of the life of other classes (variable classes). We will come back to it later in this paper.

A problem arises from the Taxis approach. As classes and data types (in our standard sense) are represented here by the same concept, *class*, one could theoretically define properties for data types. I.e. data value could have a non-empty state vector. Actually, in all the examples given in the literature, one only defines attributes for variable classes, the ones we identify as *classes* in our standard sense. Test-defined classes, which we identified as data types in the standard sense, have never attributes. However, it is never stated that test-defined classes cannot have attributes.

Metaclasses

We are now able to introduce the notion of metaclass. A **metaclass** describes a collection of classes. Metaclasses realize the notion of association. With metaclasses the set of instances of a class are treated as a whole, and attributes can be defined to it. I.e. with metaclasses we can describe factual properties over classes such as:

"the average age of (known) persons is *30*"

Such a fact cannot be expressed as:

<PERSON,average_age,30>

because definitional properties represent information about the instances of the class and not the class itself. In the other hand if we wish to represent this fact as a factual property,

(PERSON,average_age,30),

to be consistent with the property induction principle, such factual property must be induced from some definitional property of an entity which have the class *PERSON* as instance. Such entity is thus a **metaclass**. We can now create the metaclass *PERSON_CLASS*, which instances are classes denoting persons (eg. *PERSON*, *STUDENT*, *EMPLOYEE*,...). The metaclass *PERSON_CLASS* will have the definitional property

<PERSON_CLASS,average_age,NAT>

which 'allows' the class PERSON to have the factual property as before. We will use the suffix "*_CLASS*" when referring to metaclasses.

In Taxis, tokens only have factual properties, classes can have both definitional and factual properties and metaclasses have only definitional properties.

Every class in Taxis can be an instance of more than one metaclass, as well as tokens can be instances of more than one class. However, as far as we could see, instances of a meta-class must be homogeneous.

An example of specification of classes and metaclasses is:

metaclass *PERSON_CLASS* **with**
attribute_properties
average_age:NAT
end

The class *PERSON* is defined as an instance of the meta_class *PERSON_CLASS* by indicating:

PERSON_CLASS PERSON **with**
keys
name
characteristics
name: PERSON_NAME
attribute_properties
age: {|*0::150*|}
address: ADDRESS_VALUE
phone#: PHONE_VALUE
civil_state: CIVIL_STATE_VALUE
end

Objects

The notion of *object* as being a dynamic entity with an identity and a state vector does not exist in Taxis. We have tokens which can be viewed as the identities of objects, and we have properties which apply to tokens. However, Taxis does not possess a concept to apply to both token and properties, viewed as a whole.

Transactions

Taxis does not possess our standard notion of ***event***. In Taxis the concept which is more equivalent to event is the **transaction**. Moreover, it is not possible to express processes in TAXIS. Usually, each event definition will correspond to a different transaction class. The only way to impose any order in the occurrence of events is through attribute values. For example, if we want to say that the event *get_grade*(s, c, g) can only occur after the event *enroll_course*(s, c), we need to have somewhere an attribute stating that student s enrolled in course c, and a prerequisite to *get_grade*(s, c, g) is that such an attribute exists.

As we said, transactions are instances of classes, called **transaction classes**. Transaction classes are instances of the special metaclass *TRANSACTION_CLASS*. Transaction classes are defined by a **parameter_list**, defining a (complex) property which subject(s) is the parameters of the transaction; local properties, **locals**, which are either parameters or local variables of the transaction; a body which is given in terms of zero or more prerequisites, actions and result properties (**prereqs**, **actions** and **result**, respectively), which *p_*values are expressions; finally, **returns**, associates with the transaction an expression to be evaluated when the execution of the

body has been completed. For example the transaction class *ENROLL_COURSE*, describing the way student enroll in courses in our University UoD, can be defined as:

TRANSACTION_CLASS ENROLL_COURSE **with**
parameter_list
enroll_course(*s*, *c*);
locals
s: *STUDENT*;
c: *COURSE*;
prereqs
available_course?: *c.start_date* < "today";
student_max?: *s.current_courses* < 5;
actions
enroll:
insert_object in *STUDENT_COURSE* **with**
student←*s*, *course*←*c*;
increment_courses: *s.current_courses*←*s.current_courses*+1;
end

It is assumed that the variable class *STUDENT_COURSE*[6] has already been defined (cf. appendix C4) and it has two characteristics *student* and *course*.

The **parameter_list** property of *ENROLL_COURSE* defines the complex property
<(*STUDENT*,*COURSE*), *enroll_course*, *ENROLL_COURSE*>
locals define the properties,
<*ENROLL_COURSE*, *s*, *STUDENT*>, and
<*ENROLL_COURSE*, *c*, *COURSE*>

A transaction class is similar to a variable class in that it has a time varying extension. The execution of a transaction begins by adding a token to the extension of the transaction class. Execution then proceeds by evaluating each prerequisite *p_*value expression to make sure that it returns the value **true**. If any of the prerequisite expressions gives a value other than **true**, the execution of the transaction is suspended and a special transaction, *exception*, can be called. Otherwise, action expressions and result expressions are evaluated. Result expressions must also return the value **true** or, otherwise, an exception is called and the execution of the transition is suspended. Prerequisite and result expressions can be thought of as pre- and post-conditions which must be satisfied if the execution of the transaction is to be meaningful. If any pre-requisite or result expression fails (i.e. is evaluated to false) and no exception is defined for that case, the transaction is suspended and an error arises.

If the *p_*value of a definitional property <*C*, *p*, *T*> is a transaction, where *C* is a variable class, the meaning of the property changes in that *T* specifies not a type for the *p_*values of the corresponding factual properties but rather an algorithm for getting them. I.e. such a property has a *derived* set of values. For example, consider that the property
<*PERSON*,*year_of_birth*,*COMPUTE_YEAR_OF_BIRTH*>

6- This class models the set-valued attribute of *STUDENT* representing the set of courses the student ever enrolled. Remember that set-valued attributes are not possible to define in TAXIS.

is added to the class *PERSON*. The definition of the class *COMPUTE_YEAR_OF_BIRTH* is

TRANSACTION_CLASS COMPUTE_YEAR_OF_BIRTH **with**
parameter_list
year_of_birth: (p);
locals
p: PERSON
y: INTEGER;
action
year_of_birth: *y*← "this_year" - *p.age*;
returns
return: y;
end

Clearly, to every person the property *year_of_birth* can be associated with, not an instance of *COMPUTE_YEAR_OF_BIRTH*, but the token returned by the *p*_value of the *return* property, i.e. the value of *y*.

Transaction classes, then, can be used as defining the process corresponding to a class, or as the procedure to use to calculate the value of a derived attribute.

Expressions

Expressions can appear in Taxis programs as *p*_values for prerequisite, action, result or return properties. Expressions are classes and can have definitional properties of their own (in order to associate exceptions with them). For example, the prerequisites properties of transaction class *ENROLL_COURSE*, define the properties

<*ENROLL_COURSE, available_course?, 'c.start_date* < "today"'>
<*ENROLL_COURSE, student_max?, 's.current_courses* < 5'>

Expressions are special kinds of classes in two respects:

1. their extension is always empty, and
2. their *IS_A* hierarchy is determined by the following rule: if $<T_1, p, E_1>$ and $<T_2, p, E_2>$ and $(T_1\ IS\_A\ T_2)$, then $(E_1\ IS\_A\ E_2)$, where T_1 and T_2 are transactions and E_1 and E_2 are expressions.

Thus there is no need to specify the *IS_A* hierarchy between expression classes, since that is determined by the *IS_A* hierarchy between the transactions to which they are associated.

Exceptions

A special situation arises when a transaction can not proceed 'normally'. For such cases Taxis provides an **exception handling** mechanism. Exceptions are also classes and are organized in an *IS_A* hierarchy.

Exceptions arise when a prerequisite or a result expression of a transaction evaluates to a value other than **true**. Several exceptions can be associated with a transaction, to deal with every possibility of failure of prerequisites and results. To specify which exception is raised one must associate an exception class with a prerequisite or result *p*_value, which is always an expression class.

We will discuss exceptions in more detail in section 3.5.

2.4.2. Identification

As we already said, the Taxis notion of token, is equivalent to our notion of internal identification of objects (surrogates). Tokens are said to be constants thus it satisfies our requirement that the identity of an object remains unchangeable.

External identification of objects is done with the help of *keys*. Consider the variable class *STUDENT_COURSE*, defined as follows:

VARIABLE_CLASS STUDENT_COURSE **with**
keys
sc: (*stud*, *course*);
characteristics
stud: *STUDENT*;
course: *COURSE*
end

the key property *sc* specifies a complex property

<(*STUDENT*,*COURSE*),*sc*,*STUDENT_COURSE*>

which, for each pair (*stud*,*course*) returns the corresponding tuple in *STUDENT_COURSE*, if any.

It has already been mentioned that no two (factual) properties can have the same subject(s) and attribute. As keys are factual properties, from this fact it follows that a key value is unique. I.e, in the previous, example, for each pair (*stud*, *course*) there is at most one corresponding tuple in *STUDENT_COURSE*, thus a student cannot enroll the same course twice.

2.4.3. Relationships between objects

Enough has already been said about the organization of objects in classes. Moreover, Taxis has the additional capacity of organizing classes into metaclasses. We will concentrate now on the capacities of specialization of Taxis.

The *IS_A* hierarchy

As we already said, Taxis provides the facility for organizing the collection of classes and metaclasses into a hierarchy (taxonomy). The *IS_A* relationship is defined over classes and metaclasses. Informally we say that (*A IS_A B*) if *A* and *B* are both classes (metaclasses) and all the instances of *A* are also instances of *B*.

The properties of the *IS_A* relationship are summarized in terms of the following postulates:

I. All classes (metaclasses) constituting a Taxis program are organized into an *IS_A* hierarchy in terms of the binary relation *IS_A* which is a partial order.

II. There is a most general (maximum) and a most specialized (minimum) class with respect to *IS_A* called, *ANY* and *NONE*, respectively. Similarly, there is a most general and a most specialized metaclass called, *ANY_CLASS* and *NO_CLASS*, respectively.

III. (Extensional *IS_A* constraint) If (*A IS_A B*) for classes (metaclasses) *A* and *B*, then every instance of *A* is also an instance of *B*.

IV. (Structural *IS_A* constraint) If (*A IS_A B*) and *B* is the subject of a definitional property <(C_1,...,*B*,...,C_n), *p*,D>, then *A* is the subject of a definitional property

$<(C_1,...,A,...,C_n),p,E>$ and moreover $(E\ IS\_A\ D)$.

The hierarchy of classes and meta-classes form a Galois connection [Bir67].

For example, the class *STUDENT* is defined as follows:

VARIABLE_CLASS STUDENT **is_a** *PERSON* **with**
keys
stud#;
characteristics
stud#: *INTEGER*;
major: *MAJOR_NAME*;
attribute_properties
age: {|16::50|}
current_courses: {|0::5|} /*currently taken courses*/
end

By definition of *IS_A* we know that *STUDENT* inherits the definitional properties of *PERSON*, for example we will have:

<STUDENT, name, NAME_VALUE>

One weak point of the definition of the *IS_A* relationship is that nothing is said about inheritance of factual properties. In fact if we have the person *john_smith* with the factual property (*john_smith*, *name*, 'JOHN SMITH') and moreover, *john_smith* is also an instance of the subclass *STUDENT*, nothing is said about the factual properties of *john_smith* as an instance of *STUDENT*. The desirable relation is that these factual properties are inherited by the instance *john_smith* of *STUDENT*.

Transaction classes can also be organized into a taxonomic hierarchy. This will be discussed in section 3.4.

2.4.4. How do objects communicate?

The notion of communication as we defined it, is not supported by Taxis. Although, we can view a transaction which involves two different classes as a way such classes have to communicate. More about transactions as a communication mechanism will be said in section 3.4.

2.5. GALILEO

Galileo is a strongly typed, interactive programming language designed specifically to support semantic data model features (classification, aggregation and specialization), as well as abstraction mechanisms (types, abstract types and modularization). Galileo is introduced in [Alb85] and [Alb86] is the reference manual for the language. An attempt to define the semantics of Galileo is done in [Car88].

2.5.1. Concepts of Galileo

Galileo has the following features:

1. Supports the following abstraction mechanisms for database modeling:
   a) *Classification*; objects sharing common properties are gathered into classes. Elements of classes are represented uniquely in the database.

b) *Aggregation*; Elements of classes are aggregates, i.e. they are composed by heterogeneous components, which may be elements of other classes. Because of the unique representation of objects, any modification of an object is reflected wherever this object appears as component.

c) *Generalization/Specialization*; Realizing the notion of subclasses, following the IS_A hierarchy. Elements of a subclass also belong to their parent class. The type of the elements of a subclass must be a subtype of the type of the elements of the parent class.

d) *Modularization*; Data and operations can be partioned into interrelated modules. A complex schema can therefore be partitioned into smaller units.

2. It is an expression language; each construct is applied to values to return a value.

3. It is an interactive language; interaction happens at the top level of execution. At the top level one can evaluate expressions or perform declarations. This feature allows the interactive use of Galileo without a separate query language.

4. It is a functional language[7] (but not an applicative language, because it has the operator *assign*); functions are thus denotable values of the language, i.e. a function can be embedded in data structures, passed as a parameter, and returned as a value.

5. Every denotable value of the language possesses a type:

   a) A type is a set of values sharing common characteristics, together with the primitive operators which can be applied to those values.

   b) Predefined types of the language are: **bool, num, string** equipped with the usual operators, and the type **null**, which is a singleton set with the element **nil** equipped with the equality operator.

   c) Type constructors available to define new types, from predefined or previously defined types, are: tuple, sequence, discriminated union (variant), function, modifiable value (reference), and abstract types.

   d) The type system supports the notion of type hierarchy; if a type t_1 is a subtype of t_2, then a value of t_1 can be used as argument for any operation defined for values of t_2, but not vice versa. The subtype relation is a partial order.

6. Every Galileo expression has a type. The meaning of "expression e has type t" is that the value of e possesses the type t. The type of an expression can be statically determined.

7. Class elements (objects) have an abstract type and are the only values which can be created and destroyed. Other operators over classes can be defined, but, creation and destruction are automatically provided.

8. A control structure is provided for failures and their handling.

Environment

An important notion in Galileo is that of **environment**, as it is used in the denotational

7- In [Alb85] it is said that Galileo is a high-order language. However as types cannot be produced as the result of an expression, Galileo cannot be considered higher order, in the sense we understand it. We should say that Galileo is rather a functional language strongly typed.

semantics description of programming languages. An *environment definition* is a map from identifiers to type definitions or values; it is used to typecheck declarations and expressions before their evaluation. Environments are particularly useful when specifying the dynamics of a system. It provide a way to define operators over elements of classes.

A *run-time environment* is a map from identifiers to denotable values of the language, obtained by evaluating an environment expression. The evaluation of any expression takes place in the context of an environment, which specifies what the identifiers in use denote. Types are not present in run-time environments since they are not denotable values, i.e. types cannot be produced as result of an expression. Environment operators will be introduced in the sequel.

The type system

Every denotable value of the language possesses a type: A type is a set of values, possibly infinite, together with the primitive operators which can be applied to those values. The predefined types of the language are: **bool**, **num** and **string** equipped with the usual operators, and the type **null**, which is a singleton set with the element **nil** equipped with the equality operator.

Type constructors which are available to define new types are: tuple, sequence, discriminated union, function, modifiable value and abstract value.

The data structure **tuple**, such as records in relational databases, consists of a finite set of pairs, <*identifier*, *denotable value*>. The order of the pairs is not important. Examples are:

john_smith :=
    (*name* := "John"
    and *surname* := "Smith"
    and *birthday* := 19650429)

cs_dept :=
    (*name* := "Computer Science"
    and *num_employees* := 17
    and *president* :=
        (*name* := "Paul"
        and *surname* := "Moore"
        and *birthday* := 19340513)

A **tuple type** consists of an unordered set of pairs, <identifier, type>. Tuple types are just labeled cartesian products. Two tuple types are equal if they have the same set of pairs. An example of a type is:

type *Person* :=
    (*name* := string
    and *surname* := string
    and *birthday* := num)

Note that := is an environment operator, which introduces a binding between the identifier, on the left_hand side, and its value or type, on the right_hand side. The environment operator, **and**, denotes a conjunction of bindings.

A **variant**, or discriminated union, type is a disjoint sum, i.e. consists of a set of alternative values. Examples are:

type *Employee* :=

<*Technician*:(*name*: string and *skill*: string)

or *Secretary*:(*name*: string and *TypingSpeed*: num)>

is a variant type of tuples, and **type** *Int_or_Bool* :=

<*a*: int **or** *b*: bool>

is a variant of integers and booleans. Correspondent variants are:

john_smith:=<*Technician*:=(*name*: "John" and *skill*: "Analyst")>

x:=<*b*:=true>

The environment operator, **or**, introduces an alternative union, i.e the value (or type) of the identifier will be one and only one of the indicated values (types).

A **sequence** is a finite ordered collection of homogeneous elements (i.e. data with the same type). Sequences differ from sets because in sequences ordering is important and multiplicity of elements can occur. For example:

[2;3;4;4*2;3]is a sequence of integers, and

[(*name*:="Jim" **and** *age*:=10); (*name*:="Mary" **and** *age*:=35)]is a sequence of tuples

A sequence type is denoted by **seq** followed by the type of the elements. For the previous examples:

seq num

seq (*name*:string **and** *age*:num)

Two sequences are equal if they have the same element types, the same cardinality and their elements are pairwise equal, in the correct order. Two sequence types are equal if they have the same element types.

Values associated with the previous types cannot be modified. To introduce "modification" in the language, for example, to modify the value of a tuple pair or to change the value associated with an identifier in the environment, a new kind of value, the **location**, is introduced. The prefix **var** in the declaration of an attribute indicates that it can be updated, otherwise attributes keep their creation time value, during all their "life". Locations reside in a time-varying structure, the store, and are associated with values of any type, including other locations since they are also denotable values.

The next type constructor is the **function**. Functional types are built with the constructor $\rightarrow$. The type ($\alpha \rightarrow \beta$) consists of all the functions that map values of type α into values of type β. The expression "**fun**(x:α):β **is** *Expression*" denotes a function with formal parameter x, of type α, and a body, *Expression*, that returns a value of type β. This function has type ($\alpha \rightarrow \beta$).

Finally, we will concentrate on **abstract types**. The types presented so far depend only on the structure of the values. That is, the type compatibility rule adopted is the so-called, *structural equivalence* rule. These types are called **concrete** in contrast with a new kind of type, called **abstract**. Abstract types are user defined. Two abstract types are always different (i.e, we adopt a *name equivalence* compatibility rule).

Abstract types are mechanisms to abstract representations of the data from their behavior. Such behavior is defined by the designer in terms of the operations that can manipulate the data. The main reason to introduce abstract types is protection, that is, to provide a mechanism to define a new type together with operations available only to values of that type. To define abstract types

Galileo provides the environment operation:

type *T* ⇔ *Type* {**assert** [with *"Name"*] *BoolExpr*}

This environment introduces two bindings. *T* is bound with a new type isomorphic with *Type*, possibly restricted by the assertions after **assert**. The values of the abstracted type *T* are mapped into the values of the representation type *Type*, and vice versa. If an **assert** clause is present, *BoolExpr* is a Boolean expression on the values of the type. This assertion imposes constraints in the values of type *T*, which are controlled at execution time, when data is created. For example, consider the definition of the abstract type *Time*:

type *Time* ⇔ (*hrs*: num **and** *min*: num)
    assert use this in *hrs* **within** (0,23) **and** *min* **within** (0,59)

or abbreviating:

type *Time* ⇔
    (*hrs*: num **this within** (0,23)
    and *min*: num **this within** (0,59))

This introduces the abstract type *Time*. All the operators defined for the representation types are translated into the abstract type, unless the contrary is specified. For example, the predefined operators on numbers are translated into the abstract type *Age*, except mod and *, in the following definition:

type *Age* ↔ num **this within** (0,150)
    drop mod, *

The only difference between the operators ⇔ and ↔ is that when using ⇔ no predefined operators for the representation type are inherited by the abstract type, except if explicitly required (with operator **import**, not discussed here), and, using ↔, all operators of the representation type are inherited except if explicitly dropped.

Classes and Subclasses

Classes are characterized by name and the type of their elements. The name of the class denotes the set of elements actually present in the database (extension) while the type gives the structure of the elements (intention). The type of the elements must be an abstract type, therefore two elements of different classes are always different, although they may be defined with the same representation. Elements of classes are **objects**.

Elements of classes are the only Galileo entities that can be created, and destroyed. Moreover, they are uniquely represented, and when updated, their modification is reflected in all other objects in which they appear as components.

Each class is either a *base* or a *subclass*. A base class is defined independently of all othe classes, while a subclass is defined in terms of one or more other classes. A base class is used to model a primitive collection of entities, while a subclass is used to model alternative ways of looking at the same entities. All the definitions concerning one UoD are collected in a **schema**:

University := (
    rec *Persons* **class**
        Person ↔
            (*Name*: string
            and *Birthday*: num
            and *Address*: **var** *Address*)

```
            key (Name)

        and Courses class
            Course ←
                (Code: string
                and start_date: Date)
            key (Code)

        and Students subset of Persons class
            Student ←
                (is Person
                and stud#: num
                and Major: string
                and courses: optional seq Course)
            key (stud#)

        and type Address :=
            (Street: string
            and Zip: string
            and City: string)
    (...)
    );
```

It is important to note that the designers have chosen to require separate names for the class, and the type of its elements. Members of a class can be explicitly created and destroyed. Creation implies insertion into the class. However, it is not possible to create two classes over the same abstract type, since reiteration of the same abstract type name or definition generates a distinguished abstract data type (i.e. abstract data types are always different). The constructor **seq of** allows subsets of a class to be represented and explicitly be managed. **var** indicates that the attribute value can be updated and **optional** indicates that a modifiable value can be left unspecified when the object is created

Modularization

One use of **environments** in Galileo is to provide means to deal with data and operations as a single unit which can be accessed by programs, i.e. the procedural knowledge, as the data, can be shared. Environment is then, used as modularization mechanism. I.e environments can be used to structure a complex schema into smaller units. For instance, a unit may model a user view or a description of the schema produced by a a stepwise refinement methodology.

Environments have also other useful applications. first it is the mechanism used by Galileo to deal with *persistence*. Second, to deal with evolving applications, the environment is used to establish explicitly the way in which new applications interact when they use common data. Finally, the environment is used to define application oriented views of data in a similar way to the view mechanism of commercial DBMS's.

To deal with persistence, a global environment is assumed in which all values are automatically maintained. Such an environment is managed by the system which supports the language. This

global environment can be extended by adding new bindings with the command **use**. The environment mechanism can be used to structure the global environment. For instance, the following is the definition at top level, of an environment University with three classes:

```
use University :=
    (
    /*definitions as before*/
    );
```

Each expression is evaluated inside an environment, called the *current environment*. Any environment that can be accessed from the global one, can become the current one using the command **enter** *Environment*. For the previous example a simple session is

```
enter University;
```

To get the names of the students majoring Computer Science and being inscribed in at least 4 courses:

```
for x in Students
    with Major of x = "Computer Science"
    and count courses of x ≧ 4
do Name of x;
```

Transactions and Exceptions

Top-level expressions are called **transactions**, in Galileo. A transaction is an *atomic* action agains the database, i.e. once invoked, it either completes all its operations or behaves as it were never invoked. Failures can happen due to hardware, software or run-time program error. The failure of a transaction causes an interruption of the normal execution, and all updates performed since the beginning of the transaction are undone.

A transaction can be *simple*, when only one top-level expression is present, or *compound*, otherwise.

The linguistic construct to handle failures has a block structure: *Expression* **if_fails** *Expression*. If the first expression fails its effects are undone, and the value of the whole construct is that of the second expression, the **exception**. In section 3.5. we will discuss this in more detail.

The definition of the dynamic behaviour of object is described within the class specification, by extending the class definition with some expressions of type function. Consider the following example:

```
University := (
    (...)
    and Students subset of Persons class
        Student →
            (is Person
            and stud#: num
            and Major: string
            and courses: optional seq Course)
        key (stud#)
    assert (use rec fun Enroll_course(s: Student, c: Course): Student:=
        if count(courses of s)<5 and start_date of c<"today"
```

```
                then c::(courses of s)
            if_fails print "cannot enroll course"
    (...)
    );
```

where the operator "::" appends an element to a set.

2.5.2. Identification

It is said in [Alb85] that objects in the database are all distinct, and there is a one to one correspondence between the objects in the database and the entities in the UoD. Thus, although no reference is explicitly made to the way internal identification of objects is performed, we find in Galileo the concept of unicity of objects.

Galileo does provide external identification, through **keys**. The key constraint asserts that elements of a class must differ in the value of the specified constant attributes. Attributes present in a key must be constant, i.e. cannot be updated. Note that if key is not specified, insertion will be made even though the values of the attributes are all equal to those of another object already present in the class. That is, elements of a class are always different.

2.5.3. How are objects related?

Subclasses and type hierarchies are the features provided by Galileo to support the abstraction mechanism of specialization/generalization (*IS_A* hierarchies).

Def. Galileo.1 - Subtype

If a type u is a **subtype** of type v, and we write $u \subseteq v$, then a value of type u can be used in any context where a value of type v is expected.□

The subtype relation is a partial order over types [Car88] and is automatically checked for concrete type, but must be explicitly declared among abstract types, i.e. if u and v are abstract types a subtype relation between them is declared by u **is** v. Subtype relations are statically checked by the typechecker.

There is a main difference between the type hierarchy in Galileo and the *IS_A* hierarchy (class hierarchy): the subtype notion in Galileo refers to a static aspect of the language, and has been introduced to establish a compatibility rule among all the possible values of a type and those of its supertypes. On the other hand, an *IS_A* hierarchy (for example, Students *IS_A* Persons) involves two different notions. First, it establishes an existence constraint between the elements of Students and Persons present in the database, i.e. the elements of Students are always a subset of the elements of Persons (extensional notion), second, it establishes a subtype hierarchy between the type of the elements of Students and Persons (intensional notion).

In Galileo, the two notions behind the IS_A hierarchy are expressed with two distinct mechanisms: the type hierarchy, to deal with the intensional aspect, and the class hierarchy, to deal with the extensional aspect. This distinction increases the modeling capacity of the language because it allows the use of the type hierarchy independently of the subclass mechanism.

There are three ways of defining subclasses: by *subset*, *partition* and *restriction*:

subset: A subset class of T contains those elements of the parent class that have been included explicitly in the subclass with a proper operator in T. For example:

```
Students subset of Persons class
    Student -- is Person
```

partition: A partition class is like a subset class, but it enforces the additional constraint that its elements are not included in another subclass of the same partition. For example:

```
Propaedeutical_Students partition of Students class
    with Master_Students
    Propaedeutical_Student -- is Student

Master_Students partition of Students class
    with Propaedeutical_Students
    Master_Student -- is Student
```

restriction: A restriction class contains all the elements of the parent class that satisfy some predicate, which is evaluated at the time of element construction. For example:

```
Out_of_town_Students restriction of Students
    with Address: "out_of_town"
    class
        Out_of_town_Student -- is Student
```

Galileo provides the predicate **alsoin** to check whether an element of a class also belongs to a subclass. Moreover, one can operate on an object of a parent class as if it were an element of a subclass, provided that it is retyped using the predicate **likein**. This operator is needed due to the static type checking discipline.

Galileo accepts multiple inheritance, providing that the type of the elements of the subclass is a subtype of the element type of each parent class. An element of a subclass is always an element of all its parent classes.

One specific aspect of Galileo is that it does not provide *role playing*, i.e. the elements of the subclass remain there for their whole life. When an element is removed from a subclass it is also removed from the parent classes. This is clearly contrary to the other systems.

2.5.4. How do objects communicate?

Objects in Galileo do not communicate in the way we understand it. However, changes in one object may effect others to which it is related by specialization or by attribute value.

Depending in the way the dynamic part of a class is specified, and here the user is quite free to specify whatever s/he wants, it can involve other objects from the same class or other ones. Nevertheless, the concept of communication does not exist.

3. Specialization and Inheritance of Dynamics

One aspect usually specified very little, in object-oriented database systems, is the inheritance of behaviour in a taxonomic hierarchy. While it is commonly accepted and understood that if a class A is a subclass of class B, then A will inherit all the attributes of B (static definition of B), with respect to the relationship between the dynamics of A and B very little is said.

Adding attributes to a set does not create an inconsistent set (attributes are independent of each other except when related by constraints), but adding more events to a set, or even, modifying

the effects of some events, is a more complex "operation", because the effects of an event can affect the whole life of the object, (i.e. effects of events are not independent of each other).

Better said, one must take care that conditions to the occurrence of events in the superclass must still hold in the subclass. We will come back to this problem in section 3.4.1.

We can see attributes as being dimensions of a space and events as being translations in such space. Then generalization of classes will produce a space with less dimensions, when we consider the attributes of the class. Generalization of events will correspond to a projection of the translation in the larger space into a translation in the smaller space.

In the studied system we find mainly three different approaches for the problem of formalize specialization and generalization of dynamic aspects:

1. ABSURD - Abstraction
2. OBLOG - Upward and downward inheritance
3. MOKUM, TAXIS and GALILEO - Transaction mechanism and exceptions

Although MOKUM does consider specialization of the dynamics of objects, such aspects are not very much detailed. Thus, we will not in the following, consider the MOKUM system. Also, the available literature of GALILEO does not refer in detail the specialization mechanisms. However, both systems presented some similarities to TAXIS, thus we will treat them globally, keeping the emphasis on TAXIS which is, from the three, the system presenting a more detailed view of the specialization mechanisms.

The rest of this chapter will be structured as follows: we start by giving an explanation of the differences between specialization and inheritance. In section 3.2. aspects concerning attribute specialization in the different systems will be presented. Specialization of constraints is the subject of section 3.3, where we concentrate especially on inheritance of preconditions of events. In section 3.4. the specialization of events is discussed, subsection 3.4.1, focuses invariance of constraints and in subsection 3.4.2 the differences between monotonic and non-monotonic specialization are presented and the main problems of each approach discussed. In section 3.4.3. the way each system does event specialization is presented. The special case of exception handling as a specialization mechanism is treated in section 3.5. Finally, section 3.6 discusses the specialization of processes.

3.1. Inheritance versus specialization

The concept of inheritance was already present in the first object-oriented languages, like Smalltalk-80 and Simula. The basic idea is that when defining a new class it is often very convenient to start with all the structure of an existing class and to add some more attributes and/or events in order to get the desired class. The new class, B, is said to **inherit** the structure of the old one, A. This inheritance mechanism constitutes a very successful way of incorporating facilities for *code sharing* in a programming language.

Moreover, it is clear that, in the situation described above, class B has all the attributes and events of class A, i.e at any point where an object of class A is expected an object of class B will satisfy the conditions because it can accept all the messages an object of class A can accept. Thus, the objects in class B are considered **specialized** versions of objects of class A, or B is said to be a **subclass** of A, or conversely A is said to be a **superclass** of B.

It is essential to realize that the inheritance structure, used for code sharing, and the conceptual specialization mechanism are not the same thing. They lie on different levels of abstraction in the system, i.e., inheritance is concerned with the *implementation* of a description (i.e. its storage and manipulation), while specialization is based in the *behaviour* of the objects (as they are seen from the outside, by other objects).

Inheritance is a *syntactic* mechanism; it has to do with the structure of a piece of text. For the sake of understandability of the text, ease of writing, and reduction of specification effort pieces of text can be shared by the different modules that need them. By inheritance we mean *code sharing* in the linguistic sense.

On the other hand, specialization is a *conceptual* mechanism; it is concerned with the classes of objects in the conceptual schema (CM) and with their taxonomic relations. These taxonomies represent qualitative knowledge of the UoD. Inheritance in the specification is independent of the taxonomy in the CM.

For simple record-like objects the difference may not be very large. A record has the same structure as its descriptor (a tuple of values -the record- is equivalent to a tuple of constants -its descriptor) thus specialization is the same as inheritance. For more complicated objects, as UoD entities, the distinction between their interface with the external world and their internal implementation is much more important, and, therefore, the isomorphism between specialization and inheritance does not exist. This observation is not new, for example it was also presented in [Ame87].

3.1.1. Inheritance

Lets us first concentrate on **inheritance**. We consider inheritance only as a syntactic mechanism, or code sharing. This is very close to the meaning of inheritance in object-oriented programming languages. Flavors, [Moo86], for example introduces the concept of **generic operations**, which can be performed by objects of different types. This is only a way to avoid duplication of code, which, without inheritance, would appear in several places of a program. Flavors is thus concerned with the implementation of descriptions. The several objects which need such piece of code, call it by *sending a message*, which is equivalent to invoke a function in traditional programming languages. Following our views, such objects are said to inherit those pieces of code.

Also in the studied systems inheritance appears. It is mainly a **syntactic** mechanism. I.e. in a subclass hierarchy, common pieces of code are not rewritten. For example, if the class *Student* is a specialization of *Person*, in the description of *Student* we do not rewrite properties that *Student* inherits from *Person*, as *name* and *birthday*.

It should be noted that most systems do not consider such difference between specialization and inheritance. Both concepts are used as synonyms.

3.1.2. Specialization

More interesting than inheritance is the conceptual mechanism of **specialization/generalization**. It may be convenient now to define what we mean exactly by specialization.

Def. Specialization is:

1. To view an attribute ranging over a narrower set of values
2. To view more attributes, i.e to view an object as located in a higher dimensional space
3. To see more events
4. To see more of the changes done by an event
5. To impose stronger necessary preconditions on events.
6. To view a more complex process

Some examples of the previous points are (without considering any special syntax):

1. *person_age*$\in$[0,150] and *student_age*$\in$[16,50], thus *student_age* specializes *person_age* by ranging over a subset of its values.
2. *Person* = {*name*, *age*, *address*} and *Student* = *Person* $\cup$ {*st_number*, *major*, *courses_taken*}, *thus Student* specializes *Person* because the set of attributes of *Person* is a subset of the set of attributes of *Student*.
3. The event *enroll_course*(Student, Course) $\equiv$ Student.*courses_taken* $\leftarrow$ Student.*courses_taken* $\cup$ {Course}, does not occur for *Person*, as it only affects attributes of *Student* which are not attributes of *Person*.
4. If we consider the, not very likely, situation that when a student changes its address then it will move to the university closer to the new town, and therefore its *st_number* will change, then we have that the event *change_address* effects more changes for *Student* than for *Person*: *change_address*(Person, New_address) $\equiv$ Person.*address* = New_address, and *change_address*(Student, New_address, New_st_number) $\equiv$ Student.*address* = New_address **and** Student.*st_number* = New_st_number.
5. If a person can move to another town then s/he has a permit to live there, i.e. If the event *change_address* occurs (for *Person*) then the precondition *has_permit* holds.
   If a student can move to another town then s/he has a permit to live there and s/he was accept in the town university. I.e if the event *change_address* occurs (for *Student*) then the preconditions *has_permit* and *accept_by_town_university* hold. Thus, we impose a stronger set of necessary preconditions for the occurrence of the event for members of the subclass.
6. The life cycle of *Person* is (using ACP):
   PERSON = *birth*.PERSON_LIFE + *death*
   PERSON_LIFE = (*birthday* + *change-address*).PERSON_LIFE
   and the life cycle of *Student* is more complex:
   STUDENT = *birth*.STUDENT_LIFE + *death*
   STUDENT_LIFE = PERSON_LIFE + *enroll_course*.STUDENT_LIFE

In the following we will concentrate specially in the points 3, 4, 5 and 6, which deal more directly with dynamic aspects of objects. Nevertheless, the next section will concentrate briefly in the points 1 and 2, in order to take some similarities, from specialization of attributes to the more complex problem of specialization of dynamics.

3.2. Attribute Specialization

Attribute generalization and specialization have been discussed and specified for a long time. It is not the subject of this work so we will look to it informally in order to gain some intuition about how to describe dynamic specialization.

We will follow the ideas of [Wie89b] and view objects in the CM as observers of entities in the UoD, which merely follow every state change in the observed entities. Lets consider a UoD with persons, students and employees. The same attribute can have different value ranges for different classes. We will write a_C to indicate the restriction of the values of attribute a to the class C. Then

$$age_{Person} \in [0, 150]$$
$$age_{Student} \in [16, 50]$$
$$age_{Employee} \in [14, 65]$$

A person observer knows that persons have an age which is a natural number, ranging from 0 to 150, but does not know that there are students and employees which have different values for age. A student observer knows that there are students but does not know that there are employees, and knows that ages of students range from 16 to 50.

To become more specific about the objects we are talking about is thus to become more specific about possible attribute values, which is to increase our knowledge about the attribute. On the other hand to become more general about the object we are talking about is to lose knowledge about the kind of object we are talking about.

Four of the studied systems offer the possibility of defining smaller ranges for the attributes of a specialized class. In the available literature of OBLOG nothing is said about this problem. However, we think that it must be possible to introduce in the subclass stronger constraints, to restrain the values of an inherited attribute.

In Taxis it is required that if class A is_a B and A inherits the attribute a from B, then there must exist a relationship between a_A and a_B, more precisely,

$$range\_of\_a_A \text{ IS\_A } range\_of\_a_B$$

Galileo uses the same mechanism of redefining ranges of inherited attributes. And, although nothing is explicitly said, we suppose that an equivalent relationship must hold.

In MOKUM, ranges of attributes are usually defined by restrictions, and it is required that, if A is_a B and share attribute a, then

$$\text{attr\_}a\text{\_restriction\_on\_}A \Rightarrow \text{attr\_}a\text{\_restriction\_on\_}B$$

Subclasses can have more attributes than their parent classes, inheriting, however, all the attributes from their parents. Classes form thus a hierarchy. This is the main concept people have of specialization and is thus offered by all the systems studied. The knowledge associated with such hierarchy of classes is that to become more specific about the objects we are talking about is to become more specific about applicable attributes, i.e. it is to increase our knowledge about possible states of the objects. To become more general about the objects we are talking about is to lose knowledge about the kind of object we are talking about.

Here, all the systems studied use syntactic inheritance, in the sense we introduced before. I.e in

the specification of a specialized class it is indicated: a) the corresponding superclass(es), b) the new attributes of the class not shared with the parent class(es), and possibly, c) extra restrictions on the values of inherited attributes.

Multiple inheritance is possible in all the systems studied.

3.3. Specialization of Preconditions

One important aspect of the dynamic part of a system is the definition of preconditions of an event. In the definition of specialization given at the beginning of this chapter we said that one way to specialize was to impose stronger necessary preconditions.

Just to clarify ideas, a precondition C, of event E is **necessary** if we have

if E can be executed then C holds

and is **sufficient** if

if C holds then E can be executed

Note that, in the above definition, we do not mean $E \Rightarrow C$ (or $C \Rightarrow E$ for sufficient preconditions) but occurrence_of_$E \Rightarrow$ C_holds (C_holds$\Rightarrow$occurrence_of_E, respectively).

One problem with most systems, (eg. TAXIS) is that the preconditions present are always necessary but are treated as sufficient. And when it concerns specialization such a mistake is very important. Consider the following example: we have the specializations *Child* and *Portuguese_Citizen* of *Person* and the specialization *Out_of_EEC_flight* of *Flight* and we are concerned with the event *reserve_seat*(person, flight). In the usual way, we define conditions it is:

> if *there is an available seat on flight then a person can reserve it*
>
> if *there is an available seat and one of his/her parents is also flying then a child can reserve a seat*
>
> if *there is an available seat and s/he has a valid passport then a Portuguese citizen can reserve a seat in a flight to outside the EEC countries*

Thus, we are imposing sufficient preconditions on the event and such preconditions get stronger for the specialized classes. The problem with this is that the following line of thought (A) is correct:

(A) *Manuel is a Portuguese citizen, he is 5 years-old and wants to fly from Lisbon to New York. Nevertheless, Manuel is a person thus it is sufficient that there is an available seat in the plane!*

When modeling preconditions as sufficient and imposing stronger preconditions on the specialized classes, we are viewing the superclass not as containing its superclasses but as containing only the objects which are not in its subclasses. I.e. in this way *Person* represents only the persons which are not in children and not Portuguese-citizens. We should then consider an "exception" subclass of *Person*, the class *Not_child_and_not_Portuguese_citizen* and place the conditions we gave to class *Person* in this subclass.

When defining conditions as being sufficient, one must correctly place all the events at the bottom level of the hierarchy graph, and consider the subclass *Not_yet_in_another_subclass*. A the upper levels one has to consider the conjunction of conditions of its subclasses. An observer of class *Person* does not know if a particular person is also a child or a Portuguese citizen, thus

at this level one cannot be too permissive when selling flying tickets and thus all the preconditions of all the subclasses must be verified. I.e, the sufficient preconditions for a subclass are also sufficient preconditions for the superclass, so sufficient preconditions inherit upwards. So, if we are considering sufficient preconditions the following must hold:

> *if there is an available seat and one of the parents is also flying and s/he has a valid passport then a person can reserve a seat*

Naturally we do not really want such a situation! If someone is not a child nor a Portuguese citizen, s/he does not want to be constrained to fly always with his/her parents and a valid passport!

We consider now necessary preconditions. Again taking the reserve seat example we now say:

> if *a person can reserve a seat then there is an available one on flight*
>
> if *a child can reserve a seat then there is an available one and one of his/her parents is also flying*
>
> if *a Portuguese citizen can reserve a seat in a flight to outside the EEC then there is an available one and s/he has a valid passport*

Now, before the event is executed, the person is already placed in the correct subclass, and thus the preconditions to be available are determined. A thought like (A) cannot be verified.

Moreover, the conditions on the superclass are the intersection of the conditions in its subclasses. We do not need any more to consider an exception subclass, *Not_yet_in_another_subclass* to, correctly, model the problem. Necessary preconditions are stronger in the specialized classes. I.e. necessary preconditions inherit downwards. Most of the previous observations are based on the work presented in [Wie89c].

The problem with most systems is that conditions are defined as sufficient and their specializations are treated as necessary.

3.4. Event Specialization

An event is a function on a class of objects, which changes the state of the object and keeps its identity. Formally,

> **Def.** - Event
> Let C be a class with attribute set $A = \{a_1,...,a_n\}$. An event e, over objects of class C is defined as follows:
>
> $$e{:}obj(C)\times D_1\times...\times D_j\rightarrow obj(C) \equiv$$
> $$e((o,(a_1{:}u_1,...,a_n{:}u_n)),x_1,...,x_j) = (o,(a_1{:}v_1,...,a_k{:}v_k,a_{k+1}{:}u_{k+1},...,a_n{:}u_n))$$
>
> where, $v_{a_i}\in r_{a_i}$ (range of a_i), is a transformation of the previous value of a_i, using $x_1,...$ x_j, and $a_i\in A$, for i = 1,...,j.□

For example consider the event *change_address* for objects of class *Person* with attributes *name* and *address*:

$$change\_address{:}obj(Person)\times ADDRESS\rightarrow obj(Person) \equiv$$
$$change\_address((o,(name{:}n_0,address{:}a_0),a_1) = (o,(name{:}n_0,address{:}a_1))$$

We may want to specialize an event of, say, *Person* to *Student* in quite complicated ways. For example, as before, we may want to add to the event *change_address* of person, the change of university and thus of *student_number*, when applying such an event to students. Or, if the person to whom we apply it is an employee, then we may wish that the event reflects that when s/he is moving to a place further from the place where s/he works, s/he will get an allowance for transportation, which is reflect in a raise of salary. I.e., to become more specific about the objects we are talking about is to become more specific about the effect of the events on attribute values. On the other hand, to become more general about the objects we are talking about is to lose knowledge about the effects of an event which are specific for different subobjects.

If *change_address* is an event of *Person*, it is inherited by *Student*, a subclass of *Person*. A *Person* object observes an arbitrary person and does not know anything about students or employees nor about the specific attributes of these, so it can not know anything about possible effects of *Person* events over such attributes either. A *Student* object observes a student and does know that persons exist and, that students are specializations of persons. Moreover, a student observer knows nothing about employees.

Within the subclass *Student* we can define other (new) events which an observer of *Person* does not see. For example the event *enroll_course* is defined for students and is not observable by *Person*. I.e. the occurrence of such event in the life of a student, has no correspondence with any event on the life of the corresponding person. We say then that when an observer of *Student* sees the occurrence of *enroll_course*, the observer of the corresponding person "sees" an **invisible event**, which we represent by τ.

This vision of the problem is formally defined in ABSURD. OBLOG, has an opposite approach to this problem: i.e. in this case a *Person* object would 'inherit' the specific information of its subclasses. TAXIS and GALILEO use the concept of *exception* to model specialization of events. MOKUM does not describe how specialization of dynamics is done although it is possible to define a script to the specialized class, but, as far as we know, without taking in account the script(s) of the parent class(es).

3.4.1. Invariance

In the beginning of this chapter, we referred to the fact that event specialization is quite more complicated than attribute specialization, because events are not independent of each other. We mean, by not independent of each other, that the addition of a new event must take into account the state of the environment in which the event is to be inserted. Specially, we must be careful in checking that the effects of the event verify possible constraints over the modified attributes.

Constraint manipulation is, thus, a critical aspect of dynamic systems. That is to say, the occurrence of any event in the system must be such that, if a constraint is verified in the moment prior to the occurrence of the event, then it must still hold after the occurrence of the event. Formally, we say that constraints are **invariant** to the occurrence of events.

Definition - Constraint Invariance

Consider the object $o = (\sigma, (a_1,...,a_n))$, with identity σ and attributes $\{a_1,...,a_n\}$, and an event e over o. If $C(\underline{a})$ is a constraint over a subset of attributes of o, $\underline{A}$, then we say that $C(\underline{a})$ is *invariant* to e iff

$$\{C(\underline{a})\}\ e\ \{C(\underline{a})\}$$

i.e. if C holds before the occurrence of e then C is still verified after e occurs.□

Consider, for example, the attribute *price* of the class *Product*, subject to the constraint

C: $\forall o$ in *Product*, $o.price \geqq 0$

i.e. prices are always positive. Now, consider the event *change_price* over *Product*, defined as follows:

change_price: $Product \times NAT \rightarrow Product \equiv$

change_price$((o,(price:x,...)),y) = (o, (price:y,...))$

then the constraint C is invariant to event *change_price*, because $\forall\ y \in NAT,\ y \geqq 0$.

When defining specializations of classes, constraints over attributes of the superclass, must also be invariant for events of the subclass which modify values of attributes of the superclass. Continuing our example, consider now the subclass of *Product*, *Perishable*, with attribute *validity* (representing the number of days the product can still be sold), and imagine that we want that if the product has only one month to live then we decrease its price in y units, i.e. we have, in *Perishable*, the event

change_price: $Perishable \times NAT \rightarrow Perisable \equiv$

change_price$((o,(price:x,validity:30,...)),y) = (o, (price:x-y,validity:30,...))$

In this case, the constraint C over price is not always invariant. Consider the case of $y > x$.

We identified, a first problem of event specialization. Such a problem can be solved by checking constraint invariance not only for constraints defined in the class but for all the constraints defined within its superclasses. We can then speak of *constraint inheritance*.

3.4.2. Monotonic versus Non-monotonic Specialization

In the following, we will distinguish, in a subclass, inherited events, i.e. events introduced within the specification of its superclass(es), and the events introduced within the definition of the subclass itself. We suppose that events are identified by their names, meaning that events with the same name are the same event, i.e events must have global names. Moreover, we consider that one and the same event can have different effects over the state of an object, depending on whether we are considering a class or one of its specializations, that is to say that the definition of an event can be modified in the subclass specification, though, it is still considered the same event.

An event defined within a subclass specification can either be a **new event**, i.e. an event identified by a new name, or a **extended inherited event**, i.e. an event with te same name as an inherited event, but with different effects over attribute values.

This may look complicated, but is actually, very common and possible in all the systems, even if not always all the consequences are analyzed. Consider the following example from the business world: we have a company with departments and employees, some of which are managers. All employees can move from one department to another, and get changes in their salaries, and managers can hire and fire other employees.

Employee with

attributes

name: NAME, dept: Department, salary: NAT

events

change_dept:$Employee \times Department \rightarrow Employee \equiv$

change_dept((*e*,(*name*:n,*dept*:d0,*salary*:s)),d1) =
(*e*, (*name*:n,*dept*:d1,*salary*:s))
change_salary:*Employee*×*NAT*→*Employee*≡
change_salary((*e*,(*name*:n,*dept*:d,*salary*:s0)),s1) =
(*e*, (*name*:n,*dept*:d,*salary*:s1))

Manager specialization of *Employee* with
attributes
employees: set_of(*Employee*)
events
change_dept:*Manager*×*Department*→*Manager*≡
change_dept((*m*,(*name*:n,*dept*:d0,*salary*:s,*employees*:e1)),d1) =
(*m*, (*name*:n,*dept*:d1,*salary*:1.1s,*employees*:e2))
hire_emp:*Manager*×*Employee*→*Manager*≡
hire_emp((*m*,(*name*:n,*dept*:d,*salary*:s,*employees*:s)),*e*)=
(*m*, (*name*:n,*dept*:d,*salary*:s,*employees*:s∪{*e*})
fire_emp:*Manager*×*Employee*→*Manager*≡
fire_emp((*m*,(*name*:n,*dept*:d,*salary*:s,*employees*:s)),*e*)=
(*m*, (*name*:n,*dept*:d,*salary*:s,*employees*:s-{*e*})

Looking to the events defined within *Manager*, we have then an extended inherited event, *change_dept*, which states that when a manager changes department, s/he gets a raise of salary of 10%, and two new events, *hire_emp* and *fire_emp*.

Considering events introduced in a subclass definition, we are specially concerned about their effects over inherited attributes. According to such effects we say that class *A* is a **monotonic** specialization of class *B* if,

Definition - Monotonic Specialization

We say that the subclass *A* of *B* is a **monotonic** specialization of *B* if for any event *e* introduced in the definition of *A*, we have that, either,

1) *e*, is a new event which modifies only attributes of *A* not inherited from *B*, or
2) *e* is an extended inherited event, which keeps the effects of the correspondent event in *B* for inherited attributes. □

A subclass *A* is a **non-monotonic** specialization of class *B*, if it is not a monotonic specialization. It is enough only one event event introduced within *A* to be non-monotonic, for *A* to be a non-monotonic specialization.

Definition - Non-monotonic Specialization

We say that the subclass *A* of *B* is a **non-monotonic** specialization of *B* if there is one event *e* introduced in the definition of *A*, such that

1) *e*, is a new event which modifies inherited attributes of *A*
2) *e* is a extended inherited event, which changes the effects of the correspondent event in *B* over inherited attributes. □

Monotonic specialization is easier to study and formalize, and in this paper we will mainly, refer to it. There is hardly any literature, over non-monotonic specialization, and its semantics

are quite difficult to formalize. There are however, simple cases of non-monotonic specialization which can be reduced to monotonic and which will be referred to later in this subsection.

An ideal system should allow only monotonic specialization. The reason is that we want specialization to be transparent for an observer of the superclass, i.e. such an observer should be unaware of any event occurring in the subclass. In the first case of monotony, only not-inherited attributes are modified. As an observer of the superclass does not see such attributes, an event modifying their values is transparent to him. If a newly introduced event would modify attributes of the superclass, as in the first case of non-monotony, such modification would appear to an observer of the superclass as a random modification of the value of one of its attributes, because he was not aware that the event occurred in the subclass.

In the case of non-monotony, we extend an inherited event by adding the effect of such event over the not-inherited attributes of the subclass, but keeping the effects over the inherited attributes. If we would change the effects of an inherited event over the inherited attributes, as in the second case of non-monotony, an observer of the superclass would see that such an event has occurred (because it is also an event of the superclass) but he would not "recognize" the effects of the event, as they would be different from the definition of the event in the superclass.

The previous example of employees and managers can easily be proved to be non-monotonic. Unfortunately, in the real world we have several other examples of UoD's with non-monotonic specializations. Most systems not even consider the differences between monotonic and non-monotonic specialization, allowing both without being concerned about their formal semantics. Indeed, most systems, eg. Taxis or Mokum, do not define the semantics for the dynamic part of their specifications, thus one can deduce that everything is allowed. Also in OBLOG is still missing a good semantics for its dynamic part. ABSURD dedicates much work to the problem of giving a semantics to the dynamic part of a specification, namely for the specialization of dynamics, but the problem does not have, by no means, a easy solution. Nevertheless, monotonic specialization "behaves" better, thus we will give special attention to it in the rest of this paper.

Renaming of events

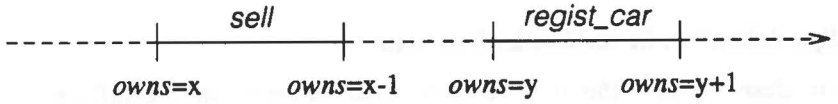
We can now consider a special case of a non-monotonic specification, which can be approached by a monotonic one. Consider the following specification of UoD:

Owner with
  attributes *owns*: set_of(*Car*)
  events *buy*:*Owner*×*Car*→*Owner*≡
    buy((*o*,(*owns*:s)),c) = (*o*, (*owns*:s∪{c}))
    sell:*Owner*×*Car*→*Owner*≡
    sell((*o*,(*owns*:s)),c) = (*o*, (*owns*:s-{c}))
  life OWNER = (*buy* + *sell*).OWNER
Manufacturer specialization of *Owner* with
  events *regist_car*:*Manufacturer*×*Car*→*Manufacturer*≡
    regist_car((*m*,(*owns*:s)),c) = (*m*, (*owns*:s∪{c})
  life MANUFACTURER = (*regist_car* + *sell*).MANUFACTURER

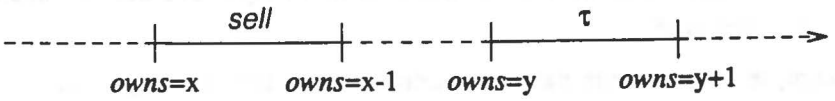
In this universe we have, thus, owners of cars which can buy or sell cars, and a specialization of

owners, manufacturers, which can sell cars but which happen to never sell them. Moreover manufactures can register the cars they build.

This system is, clearly, non-monotonic once the event *regist_car* newly introduced in the definition of the subclass *Manufacturers*, modifies the value of the inherited attribute, *owns*. Graphically, the life of an object of class *Manufacturer* can be viewed as:

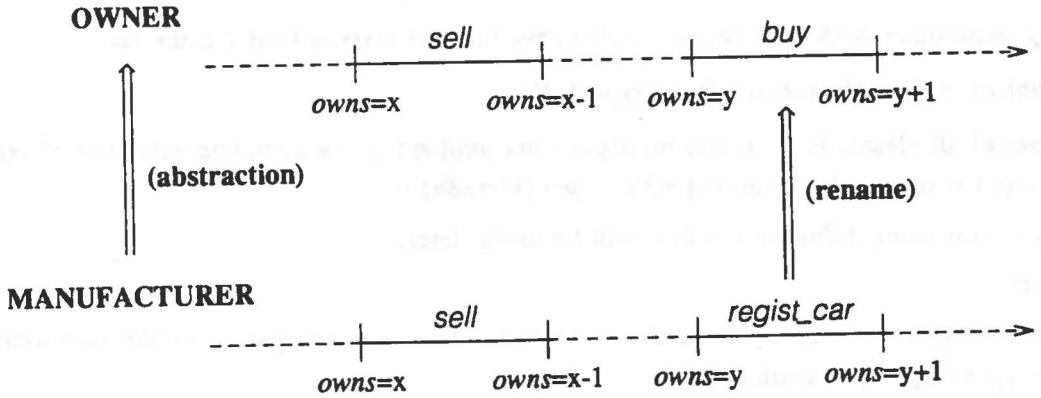


As we discussed before, in a specialization the life of an object while member of the superclass must be possible to deduce from the life of the object while element of the subclass, by abstracting the events introduced in the subclass. Thus, if we apply abstraction to MANUFACTURER, we obtain:



As *regist_car* is an event introduced in *Manufacturer*, it must be renamed to the invisible event τ, in the abstracted process. Nevertheless, the effects of *regist_car* over the attribute *owns* remain, thus, for an observer of *Owner*, it looks like, although no event has occurred, the value of attribute *owns* changed. It is thus a non-monotonic behaviour.

Now, if we look for the definition of the events *buy* of *Owner* and *regist_car* of *Manufacturer* we see that their effects over the variable *owns* are exactly the same, namely inserting a new car in the set. Such behaviour suggests that we would do better if, instead of renaming *regist_car* to τ, we rename it to *buy* as their effect of *owns* is the same, and *Owner* do recognize the event *buy*. Graphically we get,



Recall again the example of the previous subsection, about employees and managers, where the non-monotony was due to the fact that the extendend inherited event *change_dept* in *Manager* also modified the value of attribute *salary* of *Employee*:

$$change\_dept{:}Manager \times Department \rightarrow Manager \equiv$$
$$change\_dept((m,(name{:}n,dept{:}d0,salary{:}s,employees{:}e1)),d1) =$$

$(m, (name{:}n, dept{:}d1, salary{:}1.1s, employees{:}e2))$

In this case we could also avoid non-monotony by renaming this event to the composition of the events *change_dept* and *change_salary* of *Employee*. I.e, what an employee observer would see when the event *change_dept* occurred for a manager was:

change_dept((*m*,(*name*:n,*dept*:d0,*salary*:s)),d1).
change_salary((*m*,(*name*:n,*dept*:d,*salary*:s)),1.1s).

3.4.3. Event Specialization in the Studied Systems

It is now time to describe how the studied systems describe event specialization. We will concentrate on monotonic specialization.

Event Specialization in ABSURD

We will first concentrate in ABSURD. Informally, we can say that in ABSURD events occurring in the objects of a subkind must include those occurring in the life of a superkind.

As defined before, a *natural kind* is a subset $\mathcal{k}$ of S (the set of surrogates), such that $kind(type(\mathcal{k})) = \mathcal{k}$. $\mathcal{K}$ is the set of all natural kinds. *Natural types* are defined analogously and the set of all natural types is $\mathcal{T}$.

Now, $\mathcal{P}(S)$ is a complete lattice[8] with partial ordering $\subseteq$, and *lub* $\cup$ and *glb* $\cap$.

It is not very difficult to prove that $\mathcal{K}$ is a complete lattice under set inclusion. (See [Wie88b].) However, the *lub* and *glb* of a collection of sets in $\mathcal{K}$ is not the same as in $\mathcal{P}(S)$. Using $\bigsqcup$ for the *lub* and $\bigsqcap$ for the *glb* of two sets in $\mathcal{K}$, we have:

$$\mathcal{k}_1 \bigsqcap \mathcal{k}_2 = \mathcal{k}_1 \cap \mathcal{k}_2$$
$$\mathcal{k}_1 \bigsqcup \mathcal{k}_2 = \bigsqcap \{\mathcal{k} \mid \mathcal{k}_1, \mathcal{k}_2 \subseteq \mathcal{k}\}$$

I.e. the *lub* of two natural kinds is the smallest natural kind containing both of them, and their *glb* is simply their intersection. For example if *Student* and *Employee* are natural kinds then *Student*$\cap$*Employee* is also a natural kind but *Student*$\cup$*Employee* is not necessarily one. If we consider *Person* = *Student*$\cup$*Employee*$\cup$*Child*, then *Person* is smallest natural kind containing *Student*$\cup$*Employee*, i.e. *Person* is the *lub* of *Student* and *Employee*.

Recalling some other ABSURD definitions we have that the *class of kind* $\mathcal{k}$ is the set

$$obj(\mathcal{k}) = \{(s,\sigma) \mid s \in \mathcal{k}, \sigma \in space(type(\{s\}))\}.$$

and the set of all classes is U. It can be proved that $obj[\mathcal{P}(S)]$ is a complete sublattice of $\mathcal{P}(U)$ and that $\mathcal{P}(S)$ is isomorphic with $obj[\mathcal{P}(S)]$. (See [Wie88b].)

We present now some definitions, which will be useful later.

Definition

1. For any type $t \subseteq A$, $\pi_t(\sigma)$ is the projection of σ (a state space) on the components corresponding to the attributes in t
2. For any $o=(s,\sigma) \in U$, $\pi_t(s,\sigma)=(s, \pi_t(\sigma)))$.

8- A set S is a lattice if there is a partial ordering of its members and for each pair $a_1, a_2 \in S$, there exists $lub(a_1, a_2)$ and $glb(a_1, a_2)$. *lub* and *glb* stands for *lowest upper bound* and *greatest lower bound*, respectively.

3. Let $t \subseteq A$ and $\mathcal{k} \in \mathcal{K}$. The t-objects of kind $\mathcal{k}$ form the set

$$obj_t(\mathcal{k}) = \pi_t[obj(\mathcal{k})].$$

We write $obj_A(\mathcal{k}) = obj(\mathcal{k})$.

We are now able to give the formal definition of an event as it appears in ABSURD:

Definition

Let $k_1,\ldots,k_n \subseteq S$, $t_1,\ldots,t_m \subseteq A$ and $\mathcal{k}_1,\ldots,\mathcal{k}_m \subseteq \mathcal{K}$. An **event** on $\mathcal{k}_1,\ldots,\mathcal{k}_m$ is a function

$$e\colon k_1 \times \ldots \times k_n \rightarrow (obj_{t_1}(\mathcal{k}_1) \times \ldots \times obj_{t_m}(\mathcal{k}_m)) \rightarrow (obj_{t_1}(\mathcal{k}_1) \times \ldots \times obj_{t_m}(\mathcal{k}_m))$$

such that if $(o'_1,\ldots,o'_m) = e(s_1,\ldots,s_n)(o_1,\ldots,o_m)$ then $id(o_i) = id(o'_i)$, $i = 1,\ldots,m$.□

An event is just a function on objects which keeps their identities invariant. If $n = 0$, the event is called *parameterless*, otherwise is called *parameterized*. If $m = 1$, the event is called *solitary*, otherwise is called a *synchronization* event. An event application is also called an *individual event* and the event itself is also called a *generic event*.

An event e is considered to have a *domain*, which we write $dom(e) = obj_{t_1}(\mathcal{k}_1) \times \ldots \times obj_{t_m}(\mathcal{k}_m)$.. We say that an event is *aplicable* to objects in its event domain. If $dom(e) = obj(\mathcal{k}_1) \times \ldots \times obj(\mathcal{k}_m)$ then e is *completely specified*. The set of all events completely specified is called E. The set of possible events is called E^*.

An application of a solitary event $e\colon k_1 \times \ldots \times k_n \rightarrow (obj(\mathcal{k}_0) \rightarrow obj(\mathcal{k}_0))$ can be viewed in a traditional object-oriented fashion as a *message* e sent to an object with *address* $s_0 \in \mathcal{k}_0$ and with *parameters* $s_i \in k_i$. The *method* to be executed is defined in the body of function e.

Now, how is event inheritance constructed in ABSURD? Consider again the situation *Person* $\supset$ *Student* $\cup$ *Employee* and *Work_Student* = *Student* $\cap$ *Employee* $\neq \emptyset$. If e is an event

$$e\colon obj(Person) \rightarrow obj(Person)$$

then e is 'inherited' by the natural kinds *Student*, *Employee* and *Work_Student*. At implementation level we can say that an object in, say, *Work_Student delegates* the responsibility of executing e to an object in *obj*(*Person*), or that it *shares* code with such an object ([Ame87, Ste87]). But at the domain level of abstraction, we cannot say this, since an object in *Work_Student* is a single individual object. There is no 'part' of the object which is 'stored' elsewhere and to which responsibility can be delegated or with which code can be shared. What we do have, at this level of abstraction, is that an object in *Work_Student* can be alternatively *viewed* as a person, a student, a employee or a working-student.

Consider the case of *observers*. To say that *Student* is a specialization of *Person* is to say that for any $o \in obj(Student) \subseteq obj(Person)$ there is an observer $O_{Student}$ which sees the object $\pi_{type(Student)}(o)$ and an observer O_{Person} which sees the object $\pi_{type(Person)}(o)$. O_{Person} sees an object $\pi_{type(Person)}(o) \in obj_{type(Person)}(Person)$ and does not know that it is also a student. On the other hand, $O_{Student}$ sees the object as student and knows that it is also a person.

ABSURD constructs event inheritance at the domain level of abstraction as a relation between more and less general observations of an object. This connects the idea of *view* from databases with the idea of *abstraction* from process algebra. A view on a database is a set of information items derivable from the database. In the domain level, where we abstract from the distinction between derived and stored information, a view of an object is simply a projection of that object

on a subset of its attributes. We may construe this as a *renaming* of pairs attribute/value to an invisible event. Using the abstraction mechanisms of process algebra [Ber86] we demand that the specification of a process is the abstraction of its implementation, which means that we rename implementation events to the invisible event τ.

Event inheritance is constructed in the same way as a projectability constraint on observation. For every event $e{:}obj(Person) \rightarrow obj(Person)$ we will require that there are observers O_{Person} and $O_{Student}$ which see the events

$$e^{\text{Person}}: obj_{type(Person)}(Person) \rightarrow obj_{type(Person)}(Person) \text{ and}$$
$$e^{\text{Student}}: obj_{type(Student)}(Student) \rightarrow obj_{type(Student)}(Student)$$

Each one of the observers sees only the changes in the attributes in $type(Person)$ and $type(Student)$, respectively. There is a relation between the events e^{Person} and e^{Student}, namely

$$e^{\text{Person}} \circ \pi_{type(Person)} = \pi_{type(Person)} \circ e^{\text{Student}}$$

From these elementary projectability relations between the applications of an event, we can define the concept of event observation.

Definition

Let $\mathcal{A} \subseteqq \mathcal{A}_0$ and $type(\mathcal{A}) = t$. A *$\mathcal{A}$-observer* of the event $e{:}k_1 \times \ldots \times k_n \rightarrow (obj(\mathcal{A}_0) \rightarrow obj(\mathcal{A}_0))$ is an event

$$e^{\mathcal{A}}{:}k_1 \times \ldots \times k_n \rightarrow (obj_t(\mathcal{A}) \rightarrow obj_t(\mathcal{A}))$$

such that for every $o \in obj(\mathcal{A})$,

$$e^{\mathcal{A}} \circ \pi_t(o) = \pi_t \circ e(o).$$

If there exists a *$\mathcal{A}$-observer* of e for each $\mathcal{A} \subseteqq \mathcal{A}_0$ we say that e is *observable*. The set of observations of e is

$$[e] = \{e^{\mathcal{A}} \mid \mathcal{A} \subseteqq \mathcal{A}_0\}.\square$$

We have identified a *$\mathcal{A}$-observer* with the description of the events it observes. The **observability requirement** for events is that this definition is well-formed, i.e. that there is a unique event $e^{\mathcal{A}}$ satisfying the equation $e^{\mathcal{A}} \circ \pi_t = \pi_t \circ e$. It is best to think of $e^{\mathcal{A}}$ as a translation in $space(t)$. It is trivial to prove that if there exists a *$\mathcal{A}$-observer* of e, then it is unique. Nevertheless, such a $\mathcal{A}$ does not always exist. For example, suppose the event *swap* defined as follows:

$$swap(s,(a_0{:}v_0,a_1{:}v_1)) = (s,(a_0{:}v_1,a_1{:}v_0))$$

Then a *$\mathcal{A}_0$-observer* of *swap*, if it exists, would see

$$swap^{\mathcal{A}_0}(s,(a_0{:}v_0)) = (s,(a_0{:}v_1)),$$

which would look as a random assignment to a_0, and not as a function, as it is the case. I.e. we have no way of defining systematically v_1 in terms of v_0. So there is no *$\mathcal{A}_0$-observer* of *swap* and *swap* is not observable.

If $\mathcal{A}_1$ and $\mathcal{A}_2 \in [e]$ for an event e, with $\mathcal{A}_2 \subseteqq \mathcal{A}_1$ then, in some sense, $\mathcal{A}_1$ is an *abstraction* of $\mathcal{A}_2$. This leads to the following definition:

Definition

Consider the event $e{:}k_1 \times \ldots \times k_n \rightarrow (obj(\mathcal{A}_0) \rightarrow obj(\mathcal{A}_0))$ and the observers of e, $e^{\mathcal{A}_1}$ and $e^{\mathcal{A}_2}$ then we say that $e^{\mathcal{A}_1}$ is an *abstraction* of $e^{\mathcal{A}_2}$ if and only if $\mathcal{A}_2 \subseteq \mathcal{A}_1$, and we write $e^{\mathcal{A}_1} \leqq e^{\mathcal{A}_2}$.□

Note that we do not insist that e must be observable, because by definition of observability there must be a $\mathcal{A}$-observer for every $\mathcal{A} \subseteq \mathcal{A}_0$. If there would exist a $\mathcal{A}_3 \subseteq \mathcal{A}_0$ for which no $\mathcal{A}$-observer exists, e wasn't observable but still $e^{\mathcal{A}_1}$ would be an abstraction of $e^{\mathcal{A}_2}$.

From the definition of abstraction we can deduce that $e^{\mathcal{A}_1} \leqq e^{\mathcal{A}_2}$ iff $t_1 \subseteq t_2$, i.e. iff $space(t_1)$ is a projection of $space(t_2)$. If we see $e^{\mathcal{A}_1}$ as a translation in $space(t_1)$, $e^{\mathcal{A}_1}$ is an abstraction of $e^{\mathcal{A}_2}$ if its translations are projections of the translations defined by $e^{\mathcal{A}_2}$. An event gets larger in $\leqq$ if it gets more dimensions, i.e. can change more attribute values, and gets smaller (more abstract) if its effect on some attribute values are omited from its specification.

We can then conclude that if $\leqq$ is defined then it is a partial ordering on $[e]$. Moreover, if $e^{\mathcal{A}_1} \leqq e^{\mathcal{A}_2}$ then on obj_{t_2} we have

$$e^{\mathcal{A}_1} \circ \pi_{t_1} = \pi_{t_1} \circ e^{\mathcal{A}_2}.$$

Specialization of events in OBLOG

OBLOG, follows a completely different approach from ABSURD. In OBLOG events defined in the subobject must "fit" in the life of the superobject. OBLOG distinguishes semantic from syntactic specialization. Syntactic specialization is treated as expected, the main differences between OBLOG and ABSURD are in the semantics of specialization.

If *STUDENT* is a specialization of *PERSON*, then, semantically the attributes and events of *PERSON* are also attributes and events of *STUDENT*, but, syntactically, we do not specify in *STUDENT* the attributes and events of *PERSON*:

object *PERSON*
  surrogate
    construct(*name*: string)
  template
    state
      attributes
        birthday: nat
        address: string
  events
    birth *creation*;
      another_year(NAT);
  life cycles
    (...)
end

object *STUDENT*
view (*PERSON*) **with**
  state
    attributes

stud_number: nat;
major: string;
courses: set(|*COURSES*|);
events
enroll_course(|*COURSE*|);
(...)
end

However, in the previous example, the object *PERSON* would also "inherit" the events and attributes specified in its specialization *STUDENT*. I.e. the semantics of specialization determine that in subobject/superobject relationship the inheritance goes from parent to child, as usual, and, also, from children to parent(s). Consider that we add the following object to the previous example:

object *MASTER_STUDENT*
view (*STUDENT*) **with**
state
attributes
study_area: string;
(...)
end

We must recall, that OBLOG calls to the semantics of an entity an object, and to the syntactic structure of such entity a template. We will represent the template of object *OBJ* by *OBJ*. In the above example we have the following syntactic situation:

MASTER_STUDENT → *STUDENT* → *PERSON*

In this situation, the object *PERSON* is the colimit of the diagram. However, from the point of view of the specified templates, we want the object *STUDENT* to inherit as much from the template *PERSON* as from *MASTER_STUDENT*. This fact suggest that we should associate with each object three templates, namely:

(a) the template **UP**: syntactic inheritance from its ancestors

(b) the template **SELF**: template specified when the object is introduced.

(c) the template **DOWN**: syntactic inheritance from its descendants.

The object, the semantics thus, is recovered from the union of the three templates. If we interprete the previous schema in terms of the objects we have that the three nodes in the diagram represent the same object. Each knot shows one part of the object: the correspondent to the specified syntactic template (SELF). In the previous example, before introducing the object view *MASTER_STUDENT* we have:

UP(*PERSON*) = empty
SELF(*PERSON*) = {*name, birthday, address, birth, another_year* }
DOWN(*PERSON*) = {*stud_number, major, courses, enroll_course* }
and
UP(*STUDENT*) = {*name, birthday, address, birth, another_year* }
SELF(*STUDENT*) = {*stud_number, major, courses, enroll_course* }
DOWN(*STUDENT*) = empty

After we introduce the object view *MASTER_STUDENT*, then we get:

UP(*PERSON*) = empty
SELF(*PERSON*) = {*name, birthday, address, birth, another_year* }
DOWN(*PERSON*) = {*stud_number, major, courses, stud_area, enroll_course*}
and
UP(*STUDENT*) = {*name, birthday, address, birth, another_year* }
SELF(*STUDENT*) = {*stud_number, major, courses, enroll_course* }
DOWN(*STUDENT*) = {*study_area*}
and
UP(*MASTER_STUDENT*) = {*name, birthday, address, stud_number, major, courses, birth, another_year, enroll_course*}
SELF(*MASTER_STUDENT*) = {*study_area*}
DOWN(*MASTER_STUDENT*) = empty

In conclusion, we can say that the introduction of a view of person changes the global template (UP+SELF+DOWN) of person. There is thus, a distinction between the view (described by template SELF) and the corresponding subobject (which corresponds to the union of all three templates).

Consider the more complex situation of fig. 4.4.3.1., were the arrows represent views, i.e, if *OBJi* → *OBJj* then we mean that *OBJi* is a view of *OBJj*.

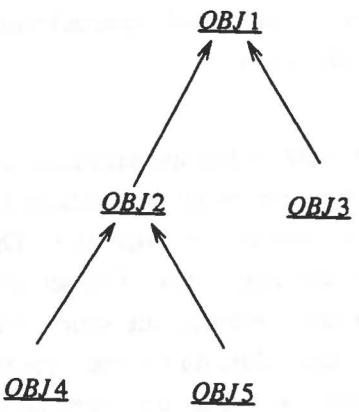


Fig. 4.4.3.1.

This is a syntactic diagram, showing only the templates SELF of each object. If we consider the semantic situation, we have for example, for *OBJ2*:

UP(*OBJ2*) = *OBJ1*
SELF(*OBJ2*) = *OBJ2*
DOWN(*OBJ2*) = *OBJ4* + *OBJ5*

which correspond to the semantic diagram showed in fig. 4.4.3.2:

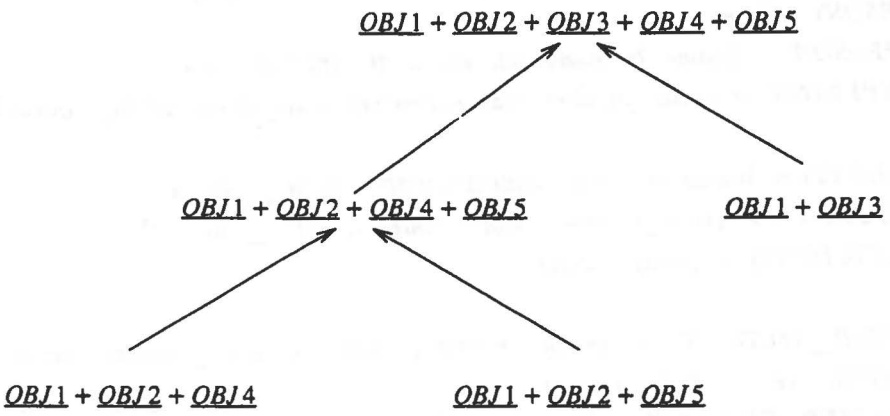


Fig. 4.4.3.2.

Here, the arrows indicate *inclusion morphisms*, the morphisms associated with the syntactic construct **view** (see 3.2.3).

A formal semantics of the mechanism of specialization does not yet exist. The OBLOG group is working on it, at this moment. It is intended that it will solve the problem of non-monotonic specialization. As we said, in case of non-monotonic specialization the superclass must be aware of same of the events defined within the subclass, because such events modify attributes of the superclass. If, we extend the inheritance from children to parents, the superclass will inherit the events defined in the subclass and thus non-monotonic specialization can be treated as monotonic just by looking to the schema "upside down".

Specialization of events in MOKUM

As we already said, MOKUM does not refer to the specialization of dynamics of objects. If A **is_a** B, and B has an associated script S_B, then, as all elements of A are also elements of B, an object of A must follow the script of B as well as the script of A. Thus, for each type an object can have, the object is in a state, called the active state. The set of all possible states is called the state set. We can say then that a subclass "inherits" the script of its superclass(es) and it can have a script of its own. However, this script shouldn't contain redefinitions of events from the inherited script because it causes a conflict. Or better, the redefinition of an event, syntactically permitted (at least not forbidden) has no effect at all, as the system will continue executing the first definition for the event.

Non-monotonic specialization is permitted. No distinction is made between monotonic and non-monotonic specialization. Moreover, as no semantics are given for scripts, the designer can define within a script, whatever s/he likes, and the task of verifying the consistency of the system is left to the designer himself.

Specialization of events in TAXIS

As mentioned before, the notion of event in TAXIS is somehow different from our standard notion. However, we will explain the structure of the transaction hierarchy within this section.

Being classes, transactions in TAXIS follow the *IS_A* hierarchy defined for all kinds of classes. Thus, from the Postulates of specialization (see section 2.4.3.) we can deduce that a sub-transaction class will have all the prerequisites, actions and results of its super-transaction class

and possibly some more. For example, consider the transaction class *GET_GRADE*,

TRANSACTION_CLASS GET_GRADE **with**
parameter_list
get_grade(*s*, *c*, *g*);
locals
s: *STUDENT*;
c: *COURSE*;
g, *g*1: {|0::100|};
prereqs
enrolled_course?: (*s*, *c*) **instance_of** *STUDENT_COURSE*
didnt_got_grade?: (*s*, *c*, *g*1) **not instance_of** *STUDENT_GRADE*
actions
grade: **insert_object in** *STUDENT_GRADE* **with**
stud←*s*, *course*←*c*, *grade*←*g*;
decrement_current_courses: *s.current_courses*←*s.current_courses*-1;
end

representing the event *get-grade* for students, and consider its specialization for engineering students, in which we add the prerequisite that the mid_term grade was already given before the student can get is final grade:

TRANSACTION_CLASS ENG_GET_GRADE **is_a** *GET_GRADE* **with**
prereqs
got_mid_grade?: (*s*, *c*, *g*1) **instance_of** *ENG_STUD_MID_GRADE*
end

TAXIS has the problem we discussed in section 2.3. Preconditions are defined as sufficient and treated as necessary. For example, as engineering students are also "normal" students they satisfy all the requisites of *GET_GRADE*, thus an engineering student could get a final grade without having received the mid_term grade, because we can apply *GET_GRADE* to an engineering student.

The problem arises from the fact that we do not have in TAXIS, a unique way to attach transaction class to class. I.e., if *A* is_a *B* members of *A* can as well use a transaction defined for *B* as a transaction defined for *A*. So we must take care that when define a transaction for *B*, we really mean *B* and not *elements_of_B_not_in_A*. A solution to this problem is the use of exceptions. I.e, the transition associated with class *A* is defined as an exception to the transitions defined for *B*. This will be treated in the next section.

Specialization of events in GALILEO

Transactions in Galileo are defined by extending a class definition with a function type. No explanation is given to the effects of specialization over transactions. These functions, as part of the type of the superclass, are inherited by its subclasses. By definition, the type of the objects in the subclass is a subtype of that of the elements of the superclass, so inherited functions can also be applied to the objects in the subclass. Moreover, other functions can be declared within the definition of the subclass. As in Taxis, one possible way to specify specialization of dynamics, is using exceptions. This will be discussed in the next subsection.

We must note that, systems like TAXIS or GALILEO are not concerned about dynamics of

objects. They have a more traditional way of viewing objects, for which dynamics is like updating mechanisms in database systems.

The main notions, which makes a language "object-oriented", are for these systems, the notions of inheritance and specialization, and uniqueness of objects [Car88]. Thus, dynamics is not an important feature of these systems, and we get it more as a "by-product" of the language than as a specially cared aspect.

3.5. Exception Handling

Some of the studied languages, namely TAXIS and GALILEO[9], use the concept of **exception** to model the dynamic part of a system. Actually, exceptions are simply actions which trigger is the failure of another action. In general, exceptions are actions performed when *static* conditions of an action fail. For example if a precondition to enrollments of a student in a course is that the student can not be currently enrolled in more than 5 other courses, an exception can be defined for the case a student which is already enrolled in 5 courses tries to enroll a new one. However, exceptions can also be used to deal with *specialization* and with the failure of *obligations*.

It should be noted that the concept of exception is not a logical concept. An exception is formally just an action. Although, for the sake of the implementation it is some times useful to treat exceptions as a different sort of action. Exceptions are non-monotonic processes. We mean that an exceptions is only reached if the normal execution of a process fails.

Exceptions and Specialization

In the previous section we mentioned that one common error when defining an event e for a class B which is a superclass of A, is that, some times by B we mean *elements_of_B_not_in_A*. In this case exceptions can be successfully used. Consider for example the event *get_grade* of students. Because we want the subclass *Eng_students* to follow a different procedure from the other students we can add the precondition to *get_grade*,

is_a_eng_student?

and define the event *eng_get_grade* for *Eng_students* as the exception to be performed when this condition fails.

In this way we avoid the problem identified in section 4.3. of a engineering student, because s/he is also a student, uses the action defined for the other students and not his/her specific action. The price to be paid is that now, we must have in the superclass the knowledge about its subclass(es).

Modelling specialization of dynamics using exception handling is possible in TAXIS and GALILEO, and we will come back to it when discussing exception handling in these systems.

Exceptions and Obligations

An **obligation** is an condition which failure leeds to a violation. Consider the case that one borrows a book from the library. A condition to borrowing a book is that the book will be returned within a certain interval. This condition is an example of obligation. Such conditions cannot be verified before the action is performed. I.e. we can check the number of courses a student

9- MOKUM also makes possible the use of exceptions, i.e. actions to be performed when a condition for a transition fails, but it do not uses the concept of exception.

enrolled before we allow another enrollement, but we can only verify if a book is returned in time, after we borrowed it.

If an obligation fails, the action has been performed already, i.e. the book has been already borrowed and there is no way one can undo the operation borrowing. We have then to define an exception action, or a chain of exception actions to increase the power of our system against getting an inconsistent state. For example, the borrower will receive a notice to return the book till the end of the current week. If again the book is not returned the borrower is due to pay a fine. If, after one month, the book is not returned and the fine is not paid, the borrowers library card is cancelled. And the process could be extended indefinitely.

We see then, that exceptions can be modeled as a chain of actions, i.e. we can define a next exception to be called when the action indicated by the first exception is violated, and so on. This procedure is particularly important when the exception derives not from the failure of a precondition but of an obligation. When a pre-condition fails the normal action will not be performed but, alternatively, the exception action will. When an obligation fails the normal action was, possibly, already performed and there is no way to undo it. If no exception is performed, and successfully completed, the system will reach an inconsistent state.

It is the wish of avoiding reaching an inconsistent state that drives the designer of the system to the definition of a chain of exceptions, hoping that one of them will, eventually, not fail. In our previous example, the system would only become inconsistent after four exceptions failed. The following figure shows the chain of actions for the borrow procedure:

return_borrowed_book → OK
↓ fail
get_notice → OK
↓ fail
pay_fine → OK
↓ fail
cancel_card → OK
↓ fail
inconsistent_state

By OK we mean that the system goal has been achieved, i.e. the book has been finally returned, and by fail we mean that the action failed and the book has been not yet returned.

The chain can grow bigger or smaller depending in how concerned we are in avoiding a inconsistent state in our system.

From the studied systems, TAXIS and Galileo are the ones which give more attention to the definition of exceptions. In MOKUM, exceptions are not explicitly referred but it is possible to define actions, within a script, to deal with the failure of "normal" actions. ABSURD and OBLOG do not use exceptions. We will first consider the handling of failure in TAXIS, and later in Galileo.

Exception handling in TAXIS

In TAXIS, exceptions are treated within the framework of classes and the *IS_A* hierarchy. Thus, exceptions classes are defined and organized into an *IS_A* hierarchy, like all other classes. The built-in metaclass *EXCEPTION_CLASS* has as instances all exception classes.

Exception are raised when a prerequisite or a result expression (pre- and post-conditions) evaluates to a value other than **true**. To specify which exception is raised, one must associate with a prerequisite or result *p_value* an expression class. Remember that the *p_value* of prerequisites and results is always an expression. Consider the transition class, *ENROLL_COURSE*,

TRANSITION_CLASS ENROLL_COURSE **with**
(...)
prereqs
can_enroll?: s.*courses* < 5 **exc**
TOO_MANY_COURSES(*stud*: s)
actions
exc_handler for *TOO_MANY_COURSES* **is**
NOTIFY_STUDENT
(...)
end

TOO_MANY_COURSES is then, the exception class to be raised if the prerequisite that a student can, at maximum, enroll 5 courses in a semester, fails. Such exception class is defined as follows:

EXCEPTION_CLASS TOO_MANY_COURSES **with**
attribute_properties
stud: *STUDENT*
end

When a exception is raised within a transaction T, it is up to the caller of T to specify what should be done to handle it. Such specifications come in the form of properties called *exception_handlers* that take as subject an expression E and an exception class *EXC*, and which *p_value* is a exception_handling transaction T_h. When an instance of *EXC* is raised during the execution of E, then T_h is called. In the previous example if the exception *TOO_MANY_COURSES* arises, the transaction *NOTIFY_STUDENT* would be called.

Using exceptions handlers in this way, means that it is possible to define chains of actions to deal with the failure of a condition. I.e. within the exception_handler class we could define another exception to be raised if the handler procedure failed, and so on, as we describe for the procedure of borrowing a book.

Up till now we have been explaining the use of exceptions handling in constraint invariance. But, as we said before, one interesting aspect of the exception mechanism of TAXIS, is that it can also be used to model specialization of the dynamics for a class, in a better way than hierarchies of transaction classes. Consider again the transaction *GET_GRADE*. This transaction is associated with the class *STUDENT*, and we saw in the previous section that this transaction can be specialized to deal with engineering students. However, we also saw that engineering students are not actually obliged to use the transaction *ENG_GET_GRADE* but can always use *GET_GRADE* and then prerequisites specific of engineering students can be left unchecked. If we add an exception in *GET_GRADE* to deal with engineering students, as follows

TRANSITION_CLASS GET_GRADE **with**
(...)

prereqs

not_eng_student?: major ≠ "ENGINEERING" **exc**

IS_ENG_STUD(*stud*: s)

actions

exc_handler for *IS_ENG_STUD* **is**

ENG_GET_GRADE(*s*, *c*, *g*)

(...)

TRANSACTION_CLASS ENG_GET_GRADE **with**

prereqs

got_mid_grade?: (*s*, *c*, *g* 1) **instance_of** *ENG_STUD_MID_GRADE*

actions

grade: **insert_object in** *STUDENT_GRADE* **with**

stud←*s*, *course*←*c*, *grade*←*g*;

decrement_current_courses: *s.current_courses*←*s.current_courses*-1;

Note that in this way to define the transaction *ENG_GET_GRADE* we have to rewrite the actions to be executed. When we define *ENG_GET_GRADE* as a specialization of *GET_GRADE* that was not necessary because then *ENG_GET_GRADE* inherited the properties of *GET_GRADE*. Now that is not the case, because there is no taxonomic relation between *GET_GRADE* and *ENG_GET_GRADE*.

The main problem with this way of treating specialization of transactions is that, here the class *GET_GRADE* must have knowledge about the specializations of student. We would prefer to have such knowledge in a lower level of the transaction classes hierarchy, i.e. at the same level of *ENGINERING_STUDENTS* in the class hierarchy.

Exception handling in GALILEO

In section 2.5. we already gave an example of the use of exceptions in GALILEO. We have two ways of expressing exceptions:

if_fails: The result of evaluating "Exp_0 **if_fails** Exp_1" is usually the result of Exp_0, unless it raises an failure, in which case it is the result of Exp_1.

case_fails: The result of the evaluation of "Exp_0 **case_fails** $String_1$ Exp_1,...,$String_n$ Exp_n", (where Exp_0 evaluates to a sequence of strings $String_i$), is normally the result of Exp_0, unless it fails, in which case it is the result of Exp_i, according to which string $String_1$ Exp_0 then evaluates.

For example, we could rewrite the definition of class *Students*, in mode to express different cases of failures:

University := (

(...)

and *Students* **subset of** *Persons* **class**

Student ↔

(**is** *Person*

and *stud#*: num

and *Major*: string

and *courses*: **optional seq** *Course*)

```
        key (stud#)
    and use rec Enroll_course(s: Student, c: Course): Student:=
        assert with "TooManyCourses" count(courses of s) > 5
        assert with "CourseNotAvailable" start_date of c) ≧ "today"
            c::(courses of s)
        case_fails
            "TooManyCourses"
                print "you enrolled already the maximum possible courses"
            "CourseNotAvailable"
                print "the course is not available at the moment"
(...)
);
```

We are more interested in see how exceptions can be used to model specialization of dynamics of objects. Using the operator **alsoin**, which checks whether or not an object in a superclass also belongs to a given subclass, one can define a selective exception for the different subclasses of a class as follows, where *B* is_a *A*:

```
use rec fun Transaction_on_A (a: A): res_type :=
    (
    if a alsoin B then ...
    else ...
    )
```

In the same way, but then using **case**, one could define different exceptions for the case that *A* has several subclasses. In appendix C5 some examples of specialization of dynamics defined in this way are given.

3.6. Process Specialization

Now the question is, how is process specialization constructed? We will only refer to ABSURD, because this is the only system which dedicates some effort to the to formalization aspects of specialization and generalization of processes.

If for each natural kind a generic process is specified, then there should be consistency requirements between processes of taxonomically related natural kinds. The desired relation is that the generalized generic process must be an abstraction of the specialized generic process [Wie89b].

Remember how things happen for attribute and event generalization. Linguistically, and considering only monotonic specialization of events, generalization from *Student* to *Person* is a *renaming operation* on the specification, which deletes attribute specifications and replaces every event e of *Student* by e_{Person} if e occurs in *space(type(Person))* or deletes it if it takes place outside *space(type(Person))*.

It would seem logical, therefore, that the desired renaming operation on process specifications would also simply rename the events in the process of *Student* to e_{Person}, or delete them, according to whether they belong or do not belong to *space(type(Person))*. This cannot be done because of the deadlock behaviour in nondeterministic processes and instead of deleting events we have to rename them to τ, the "silent" event.

It is important to understand the root of the deadlock problem. A *Student* object is a *Student*-

observer in the CM looking at a student in the UoD. Unlike what is the case with attributes and events, the *Student*-observer sees the student process not as an isolated process. He observes the whole UoD, and every event not executed by the observed student is abstracted away, i.e. it is renamed to τ. Thus the student is viewed in the context with which s/he communicates.

Attribute and event generalization is always *context-free*. Process generalization is also context free, if and only if the the observed process does not communicate with the environment. In this case the generalized process of *Person* can be obtained from the process of the subclass *Student* by deleting events taking place outside *space(type(Person))* and otherwise renaming the events to e_{Person}.

If a process communicates with the environment, then deadlock is possible and obviously the environment is not any more irrelevant. Communication destroys object modularity. For this reason process generalization is usually *context-dependent*, which means that a process behaves differently if placed in a different context. The operator || is not compositional with respect to generalization, i.e if $x \leqq y$ (process x is a generalization of process y) then

$x \leqq x'$ and $y \leqq y'$ do not imply $x \| y \leqq x' \| y'$

We must therefore take the environment into account when generalizing a process. The abstraction mechanism is, thus, applied to the *complete domain process*, i.e. the process which the CM is specified to execute, and not merely to the process executed by a single object. The process executed by any $s \in S$ is a component of the domain process D, where now we have the case that s can have several different natural kinds. If *species*(s) is the most specialized natural kind of s, then we use in the specification of D the most specific process executed by s, which is an instance of the process species(s), i.e the process associated with the natural kind *species*(s): parallel composition of the individual processes executed by all objects in the domain:

$$D = \partial_H(\partial_{H_1} \Lambda^{s_1}_{\sigma_1} \mathit{birth}.\text{species}(s_1) \| \partial_{H_2} \Lambda^{s_2}_{\sigma_2} \mathit{birth}.\text{species}(s_2 \| \ldots).$$

where $H_i = H_{s_i}, \mathit{species}(s_i)$.

Now, we must see what the relation is between species(s) and all processes k executed by s for $s \in \mathcal{k}$, ($\mathcal{k}$ is a more general natural kind of s). Informally, each such k is related to species(s) by observation, i.e. an observer of s in D at the level of generality of k sees a generalization of what an observer at level of species(s) sees. This places the consistency requirement that a process cannot deadlock at one level of observation and continue to execute at another, as all the processes of the same object are just observations of the most specialized process of the object.

We define now the renaming operator, $\mathit{view}^s_{\mathcal{k}}$ on processes, such that $\mathit{view}^s_{\mathcal{k}}$(D) is the process executed by s in the context of the domain process D as observed by a $\mathcal{k}$-observer of s. Such an operator allows us to get the more generalized processes of an object s from the more specific process of s (which appears in the specification of D). There are three requirements that this operator must satisfy:

1) The special events τ (silent step) and δ (deadlock) must be renamed to themselves. For example consider an object of natural kind *Student* which also belongs to *Person*. Any deadlock of the process of this object must be observable by a *Person*_observer. In the same way, if an event is invisible to begin with it must remain invisible in a more generalized observation of the process.

2) Any event not executed by s must be renamed to τ, for if D stagnates, so will the process

observed by any observer of D. If we can guarantee that D never stagnates, for example if at any moment there is a choice between a synchronous execution of events in the life of any number of objects, then we do not need this.

3) Any event e of s must be renamed to e^{ℓ}, or, if it occurs completely outside *space(type(ℓ))*, to τ.

The formal definition of this operator is given in [Wie89b]. We define $\alpha(k)$, the **alphabet** of k, as being the set of events occurring in k. Then, the ℓ-observer of s in D sees only synchronous occurrences involving at least one event from $\alpha(k)$ executed by s, and of those only the $\alpha(k)$-events executed by s, and of those only the effect on attributes on *type(ℓ)*.

Intuitively, we can think of a process as a trajectory through a state space, and a generalization of a process is the "projection" of the process in a subspace, "leaving out" parts which occur outside this subspace. Because objects communicate with others, and communication can lead to deadlock, we cannot just ignore events occurring outside the subspace but we rather rename them to τ.

One important assumption of the concept of generalization is that, if o is an object of kind ℓ (i.e. its identity has that kind) and o' is the same object but now viewed as an object of the more generalized kind ℓ' (i.e. $\ell \subseteq \ell'$), then, nothing in the "appearance" of o' shows that the object is also of the more specialized kind ℓ. I.e., more specialized kinds should have no knowledge at all about possible subkinds. This cannot always be achieved because, we must rename events occurring outside *space(type(ℓ'))* to τ, and then, although a ℓ'-observer does not see the occurrence of an event in ℓ, it does "see" the silent step, which leads him to suspect that there is a subkind of ℓ' and, more, there is "something" happening in that subkind. See, for example, the process specification for kind *Student* in appendix C1.

4. Conclusions

The main reason to write this report, as we mentioned in the introduction, was our idea that is was important to have means to compare the work being done in the field of object-oriented databases. Interesting aspects were the theoretical foundations of the studied systems, and, specially, the specification of inheritance of dynamics.

Another contribution of this paper is its detailed and uniform analysis of the select systems. Although the design of most of the studied models are not yet completed, and changing everyday, it was our porpose to make the most accurate possible comparison between them, even though it may not reflect the most recent devellopments in some of the models. The comparison tables in appendix A are a summary of this analysis. Table 1 provides a list of the terms and concepts used to uniformly discuss each system, as well as the corresponding terms used in the models of interest. Table 2 presents the evaluation of static features of the selected systems and table 3 presents a summary of the dynamic aspects of them.

Now, we summarize the main ideas of each studied system concerning specialization of dynamics.

In ABSURD generalization is **abstraction** in the sense that in a superclass there is less knowledge and a larger set of observable entities. (The classic paradigm of quality versus quantity.) Less knowledge signifies less constraints, less events, less effects of events and so on. This

notion of "less" is formalized as a partial ordering.

In OBLOG, an attempt is made to formalize non-monotonic specialization of events, by having superclasses inheriting events and attributes from their subclasses. We are, since very recently, aware that this notion of inheritance (described in section 3.4.3) has been changed, but no literature has been produced, yet, and because of time limitations, we choosed for keeping the description of the later formalism.

In TAXIS static and dynamic specialization is treated in different ways. Static specialization is adding constraints and dynamic specialization is adding exceptions. More important is that we could find that the way static specialization is done is incorrect. Namely, preconditions which are formally necessary are treated as sufficient and then, adding preconditions is inconsistent (see section 3.3.)

MOKUM and GALILEO make no attempt to describe their mechanism of specializing dynamics of objects. Moreover, in MOKUM it is not even possible to use syntactic inheritance in the specification of scripts. The script associated with a subclass in a static specialization, is a completely independent entity from the script associated with the correspondent superclass.

ABSURD and OBLOG are the systems more concerned with giving a denotational semantics to the language, although OBLOG does not complete the task, as for example the semantics of specialization are not (yet) defined. GALILEO also tries to define the semantics of the language (cf. [Car88]), but only for the stactic aspects of the language. TAXIS and MOKUM are more concerned with implementation issues than with semantics.

1. References

[Alb85]. A. Albano and L. Cardelli, "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Transactions on Database Systems* **vol. 10, no. 2**, pp. 230-260 (June, 1985).

[Alb86]. A. Albano, M.E. Occhiuto, and R. Orsini, *Galileo Reference Manual, VAX/UNIX Version 1.0*, Department of Computer Science, Universita' di Pisa, Pisa, Italy (1986).

[Ame87]. P. America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," in *Proceedings of the European Conference on Object-Oriented Programming*, Paris, France (1987).

[Atk87]. M. Atkinson and P. Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys* **vol. 19, no. 2**, pp. 185-207 (June, 1987).

[Bak89]. J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Springer-Verlag (1989).

[Ber86]. J.A. Bergstra and J.W. Klop, "Process Algebra: Specification and Verification in Bisimulation Semantics," in *Proc. CWI Symposium Mathematics and Computer Science II*, ed. M. Hazewinkel, J. Lenstra, and L. Meertens, North-Holland, Amsterdam (1986).

[Bir67]. G. Birkhoff, *Lattice Theory*, American Mathematical Society (1967). (3th edition)

[Bor84]. A. Borgida, J. Mylopoulos, and H.K.T. Wong, "Generalization/Specialization as a

Basis for Software Specification," pp. 87-117 in *On Conceptual Modeling*, ed. M.J. Brodie and J. Mylopoulos, Springer (1984).

[Bri89]. E. Brinksma, *On the Design of Extended LOTUS*, Faculteit Informatica, University of Twente, The Netherlands (1989). (PhD. Thesis)

[Car88]. L. Cardelli, "A Semantics of Multiple Inheritance," *Information and Computation* **76**, pp. 138-164 (1988).

[Clo81]. W. Clocksin and C. Mellish, *Programming in Prolog*, Springer-Verlag (1981).

[Cos89]. J. Costa, A. Sernadas, and C. Sernadas, "OBL-89: User's Manual," version 2.3, INESC, IST, Lisbon, Portugal (1989).

[Dah67]. O.-J. Dahl, B. Myhraug, and K. Nygaard, *SIMULA 67, Common Base Language*, Norwegian Computing Center, Oslo, Norway (1967).

[Dig89]. V. Gamito Dignum, *MOKUM (Prolog) Kernel - User Manual*, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1989).

[Ehr89]. H-D. Ehrich, A. Sernadas, and C. Sernadas, "From Data Types to Object Types," *Journal of Information Processing and Cybernetics*, (in print) (1989).

[Gol83]. A. Goldberg and D. Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley (1983).

[Hoa85]. C. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, UK (1985).

[Moo86]. D. Moon, "Introduction to the Flavors System," in *Reference Guide to Symbolics Common Lisp: Language Concepts* (1986).

[Myl80]. J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Transactions on Database Systems* **vol. 5, no. 2**, pp. 185-207 (June, 1980).

[Rei84]. R. Reiter, "Towards a Logical Reconstruction of Relational Database Theory," pp. 191-233 in *On Conceptual Modelling*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt , Springer-Verlag (1984).

[Rie89]. R.P. van de Riet, "MOKUM: An Object-Oriented Active Knowledge Base System," *Data and Knowledge Engineering*, Amsterdam, The Netherlands **vol. 4, no. 1**, North-Holland (1989).

[Ser87]. A. Sernadas, C. Sernadas, and H-D. Ehrich, "Object-Oriented Specification of Databases: An Algebraic Approach," pp. 107-116 in *Proceedings of the 13th VLDB*, ed. P.M. Stocker, and W. Kent, Morgan-Kaufmann Publ. Inc., Los Altos, USA (1987).

[Ser89]. A. Sernadas, C. Sernadas, and H-D. Ehrich, *Object-Oriented Language Features for Information Systems Specification*, INESC research report, Lisbon, Portugal (1989). (submitted)

[Ste87]. L. Stein, "Delegation is Inheritance," pp. 138-146 in *OOPSLA'87 Proceedings* (1987).

[Tak71]. G. Takeuti and W.M. Zaring, in *Introduction to Axiomatic Set Theory*, Springer-Verlag, New York (1971).

[Urb86]. S. Urban and L. Delcambre, "An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models," pp. 382-387 in *Proceedings of the International Conference on Data Engineering*, Los Angeles, USA (1986).

[Wie88a]. R. Wieringa and R. van de Riet, "Algebraic Specifications of Object Dynamics in Knowledge Base Domains," in *The Role of Artificial Intelligence in Databases and Information Systems*, ed. C-H. Kung and R. Meersman, North-Holland, Guangzhou, China (1988).

[Wie88b]. R. Wieringa, "Event Inheritance in Database Domains," IR-157, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1988).

[Wie89a]. R. Wieringa, "Conceptual Foundations for Conceptual Models," IR-..., Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1989).

[Wie89b]. R. Wieringa, "Process Generalization in Conceptual Models," IR-187, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1989).

[Wie89c]. R. Wieringa, H. Weigand, J-J. Meyer, and F. Dignum, *The Inheritance of Dynamic and Deontic Integrity Constraints*, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1989). (to appear)

[Wie89d]. R. Wieringa, "An Equational Language for Specifying, Querying and Updating Database Domains," IR-192, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1989).

[Wie89e]. R. Wieringa, "Role Change in Database Domains," IR-180, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1989).

Appendix A - Tables for Comparison of the Different Systems

Table 1:

Terminology					
TERM	**ABSURD**	**OBLOG**	**MOKUM**	**TAXIS**	**GALILEO**
Object	Object	Object	Object	*Token*	Tuple
Class	Class	Type	Class	(Variable) Class	Class
Data Type	Data Type	Data Type	Data Type	(Test defined) Class	Type
Attribute	Attribute	Attribute	Attribute	Property	*Property*
Type	Type	*Template*	Type	*Class*	(Abstract) Type
Event	Event	Event	Event	(Transaction) Class	*Transaction*
Process	Process	*Life cycle*	Script	no	no
Communication	Message	Event sharing	Message	*yes* (as transaction)	no

Words in italics refer to the closest concept of the system to the standard term.

Table 2:

Static Criteria					
Criteria	**ABSURD**	**OBLOG**	**MOKUM**	**TAXIS**	**GALILEO**
Object identity					
internal	surrogate	surrogate	surrogate	token	*yes*
external	*key*[1]	key	yes	key	key
Aggregation					
single valued attributes	yes	yes	yes	yes	yes
set valued attributes	yes?	no	yes	no	yes
object valued attributes	yes	yes	yes	yes	yes
derived attributes	?	?	yes	*yes*[2]	yes
constraints	yes	yes	yes	yes	yes
Classification					
Class vs. Type	yes	no (only type)	yes	no (only class)	yes
Gen/Spec					
Inheritance of attributes	yes	yes	yes	yes	yes
Restrictions of inherited values	yes	yes	yes	yes	yes
Multiple inheritance	yes	yes	yes	yes	yes
Association					
associated concept	group-object	composite-object	collection	metaclass	no
def. of attributes	yes	yes	no	yes	--
heterogeneous types	yes	2 types	no	no	--

Words in italics refer to the closest concept of the system to the standard term.

1 - Keys in ABSURD can only be defined by constraints.

2 - In TAXIS, derived attributes are defined using transaction classes.

Table 3:

Dynamic Criteria					
Criteria	**ABSURD**	**OBLOG**	**MOKUM**	**TAXIS**	**GALILEO**
Events					
preconditions	yes	yes	yes	yes	yes
postconditions			no	yes	?
messages	yes	no	yes	no	no
inheritance					
redef of effects for parent atts	*yes*[1]	?	yes	yes	?
def of effects for child atts	yes	?	yes	yes	?
Role playing	yes	yes	yes	yes	no
Processes				no	no
formal definition	using ACP	implicit or CSP	implicit	—	—
inheritance	yes	*yes*	*yes*	—	—
Exceptions	no	no	yes in script	yes	yes
for pre_cond	—	—	yes	yes	yes
for post_cond	—	—	no	yes	?
chained	—	—	yes	yes	?
Parallelism					
interleaving vs. partial order	interleaving	?	part. order	part. order	part. order?
Nondeterminism	yes	yes	no	no	no

Words in italics refer to the closest concept of the system to the standard term.

1 - Not always possible

Appendix B - Axioms of Algebra of Communicating Processes

$x+y=y+x$	A1	$x\tau=x$	T1
$x+(y+z)=(x+y)+z$	A2	$\tau x+x=\tau x$	T2
$x+x=x$	A3	$e(\tau x+y)=e(\tau x+y)+ex$	T3
$(x+y)z=xz+yz$	A4		
$(xy)z=x(yz)$	A5		
$x+\delta=x$	A6		
$x\delta=\delta$	A7		
$e_1 \mid e_2=e_2 \mid e_1$	C1	$e_1 \mid e_2=\gamma(e_1,e_2)$ if $\gamma(e_1,e_2)=\top$	
$(e_1 \mid e_2) \mid e_3=e_1 \mid (e_2 \mid e_3)$	C2	$e_1 \mid e_2=\delta$ if $\gamma(e_1,e_2)=\perp$	
$\delta \mid e=\delta$	C3		
$x \mid\mid y=x \lfloor\!\lfloor y+y \lfloor\!\lfloor x+x \mid y$	CM1		
$e \lfloor\!\lfloor x=ex$	CM2	$\tau \lfloor\!\lfloor x=\tau x$	TM1
$ex \lfloor\!\lfloor y=e(x \mid\mid y)$	CM3	$\tau x \lfloor\!\lfloor y=\tau(x \mid\mid y)$	TM2
$(x+y) \lfloor\!\lfloor z=x \lfloor\!\lfloor z+y \lfloor\!\lfloor z$	CM4		
$e_1 x \mid e_2=(e_1 \mid e_2)x$	CM5	$\tau \mid x=\delta$	TC1
$e_1 \mid e_2 x=(e_1 \mid e_2)x$	CM6	$x \mid \tau=\delta$	TC2
$e_1 x \mid e_2 y=(e_1 \mid e_2)(x \mid\mid y)$	CM7	$\tau x \mid y=x \mid y$	TC3
$(x+y) \mid z=x \mid z+y \mid z$	CM8	$x \mid \tau y=x \mid y$	TC4
$x \mid (y+z)=x \mid y+x \mid z$	CM9		
$\partial_H(e)=e$ if $e \notin H$	D1	$\tau_I(\tau)=\tau$	TI1
$\partial_H(e)=\delta$ if $e \in H$	D2	$\tau_I(e)=e$ if $e \notin I$	TI2
$\partial_H(x+y)=\partial_H(x)+\partial_H(y)$	D3	$\tau_I(e)=\tau$ if $e \in I$	TI3
$\partial_H(xy)=\partial_H(x).\partial_H(y)$	D4	$\tau_I(x+y)=\tau_I(x)+\tau_I(y)$	TI4
		$\tau_I(xy)=\tau_I(x).\tau_I(y)$	TI5
$\partial_H(\tau)=\tau$	DT		

Appendix C1. Example Specification in ABSURD

natural kind spec Grades
import
Naturals
sorts
GRADE
functions
minimum: GRADE → NAT
maximum: GRADE → NAT
failure: GRADE → BOOL
success: GRADE → BOOL
variables
g: GRADE
equations
[E1] minimum(x) = 0
[E2] maximum(x) = 10
[E3] failure(x) = true **when** x < 6
[E4] success(x) = true **when** x ≧ 6
end spec Grades

natural kind spec Persons
import
Identities, Names
identity
PERSON **specializing** ID
attributes
name: PERSON → NAME [1-1]
end spec Persons

natural kind spec Courses
import
Identities, Names
identity
COURSE **specializing** ID
functions
c0: COURSE
attributes
code: COURSE → NAME [1-1]
from_major: COURSE → MAJOR
end spec Courses

natural kind spec Students

import

Identities, Courses, Grades,

Sets **using** Courses **for** Items
binding [ITEM → COURSES,
eq → eq]
renaming [SETS → COURSES],

Sets **using** Course × Grades **for** Items
binding [ITEM → COURSE × GRADE,
eq → eq × eq]
renaming [SET → GRADES]

identity

STUDENT **specializing** PERSON

functions

s0: STUDENT

attributes

major: STUDENT → MAJOR
courses: STUDENT → COURSES
grades: STUDENT → GRADES

events

choose_major: STUDENT × MAJOR → STUDENT
enroll_course: STUDENT × COURSE → STUDENT **external message**
get_grade: STUDENT × GRADES → STUDENT **external message**
finish_study: STUDENT → STUDENT **final event**

variables

s: STUDENT
m: MAJOR
c: COURSE
g: GRADE

equations

[E1] *major*(*choose_major*(s, m)) = m

[E2] *courses*(*choose_major*(s, m)) = empty

[E3] *grades*(*choose_major*(s, m)) = empty

[E4] *courses*(*enroll_course*(s, c)) = *courses*(s) + {c}

[E5] *grades*(*get_grade*(s, (c, g))) = *grades*(s) + {(c, g)}

[E6] *courses*(*get_grade*(s, (c, g))) = *courses*(s) - {(c)}

preconditions

[P1] *enroll_course*(s, c)
when c not in *courses*(s)
and (c, g) not in *grades*(s)
and card(*courses*(s)) < 5
and *from_major*(c) = *major*(s)

invariants

[I1] card(*courses*(s)) ≦ 5

process selection

[S1] STUDENT = *choose_major*.LIFE_ST

[S2] LIFE_ST = *enroll_course*.τ.*get_grade*.LIFE_ST+ *finish_study*

end spec Students

natural kind spec Engineering_Students

import

Identities, Courses, Grades, Students

Sets **using** Course × Grades **for** Items

binding [ITEM → COURSE × GRADE,

eq → eq × eq]

renaming [SET → MID_GRADES]

identity

ENG_STUD **specializing** STUDENT

functions

es0: ENG_STUD

attributes

mid_grades: ENG_STUD → MID_GRADES

events

get_mid_grade: ENG_STUD×MID_GRADES→ENG_STUD **external message**

variables

es: ENG_STUD

m: MAJOR

c: COURSE

mg: GRADE

equations

[E1] *mid_grades*(*choose_major*(s, m)) = empty

[E2] *mid_grades*(*get_mid_grade*(es, (c, mg))) = *mid_grades*(es) + {(c, mg)}

invariants

[I1] *major*(es) = "Engineering"

process selection

[ES1] ENG_ST = *choose_major*.LIFE_ENG

[ES2] LIFE_ENG = *enroll_course.get_mid_grade.get_grade.* LIFE_ENG + *finisdh_study*

end spec Engineering_Students

natural kind spec Teachers

import

Identities, Courses, Enrollments, Grades

identity

TEACHER **specializing** PERSON

functions

t0: TEACHER

attributes

course: TEACHER → COURSE

events

became_teacher: TEACHER → TEACHER

teaches_course: TEACHER × COUSE → TEACHER

term: TEACHER → TEACHER **external message**

term_remind: TEACHER → TEACHER **external message**

give_grade: TEACHER × ENROLL × GRADE → TEACHER **external message**

mid_term: TEACHER → TEACHER **external message**

mid_term_remind: TEACHER → TEACHER **external message**

give_mid_grade: TEACHER × ENG_ENROLL × GRADE → TEACHER **external message**

quit: TEACHER → TEACHER **final event**

variables

t: TEACHER
s: STUDENT
c: COURSE
e: ENROLL
mg, g: GRADE

equations

[E1] *course*(*teaches_course*(t,c)) = c

preconditions

[P1] *give_grade*(t,e,g)
when *course*(e) eq *course*(t) = true

[P2] *give_mid_grade*(t,e,mg)
when *course*(e) eq *course*(t) = true
and *from_major*(*course*(e)) = "Engineering"

process selection

[T1] TEACHER = *became_teacher*.LIFE_TEACHER

[T2] LIFE_TEACHER = *teaches_course*.(*mid_term*.T1 + *term*.T2).(LIFE_TEACHER + *quit*)

[T3] T1 = *give_mid_grade*.*term*.T2 + *mid_term_remind*.T1 + *term*.T3

[T4] T2 = *give_grade* + *term_remind*.T2

[T5] T3 = *give_mid_grade*.T2 + (*term_remind* + *mid_term_remind*).T3

end spec Teachers

natural kind spec Enrollments

import

Identities, Courses, Grades, Students

identity

ENROLL **specializing** ID

functions

enr0: ENROLL

attributes

stud: ENROLL → STUDENT

course: ENROLL → COURSE

end_term: ENROLL → BOOL

events

enroll: ENROLL × STUDENT × COURSE → ENROLL **external message**

term: ENROLL → ENROLL **external message**

issue_grade: ENROLL × GRADE → ENROLL **external message**
final event

variables

e, e1, e2: ENROLL

s: STUDENT

c: COURSE

g: GRADE

equations

[E1] **when** *stud*(e1) eq *stud*(e2) = true
and *course*(e1) eq *course*(e2) = true
then e1 eq e2 = true

[E2] *end_term*(*enroll*(e,s,c)) = false

[E3] *end_term*(*term*(e)) = true

preconditions

[P1] *enroll*(e,s,c)
when *major*(s) eq *from_major*(c) = true

[P2] *issue_grade*(e,g)
when *end_term*(e) = true

process selection

ENROLL = *enroll.term.issue_grade*

end spec Enrollments

natural kind spec Eng_Enrollments

import

Identities, Enrollments, Courses, Grades, Engineering_Students

identity

ENG_ENROLL **specializing** ID

functions

enr0: ENG_ENROLL

attributes

after_mid_term: ENG_ENROLL → BOOL

events

mid_term: ENG_ENROLL → ENG_ENROLL **external message**

issue_mid_grade: ENG_ENROLL×GRADE→ENG_ENROLL **external message**

variables

ee: ENG_ENROLL

es: ENG_STUDENT

c: COURSE

g: GRADE

equations

[E1] *after_mid_term*(*enroll*(ee,es,c)) = false

[E3] *after_mid_term*(*mid_term*(ee)) = true

preconditions

[P1] *issue_mid_grade*(ee,g)

ewhen *after_mid_term*(ee) = true

process selection

[EE1] ENG_ENROLL = *enroll.mid_term*.E

[EE2] E = *issue_mid_grade.term.issue_grade* + *term.issue_mid_grade.issue_grade*

end spec Eng_Enrollments

natural kind spec ExistenceMonitor

import

Persons, Students, Engineering_Students, Teachers, Courses, Enrollments, Eng_Enrollments, Grades, Names

attributes

person_names: DB → NAME

student_names: DB → NAME

teacher_names: DB → NAME

course_names: DB → NAME

events

create-PERSON

create-COURSE

create-STUDENT

create-TEACHER

variables

pn, sn, tn, cn: NAME

p: PERSON

t: TEACHER

s: STUDENT

c: COURSE
e: ENROLL
g: GRADE

equations

[E1] *person_names*(*create_PERSON*(**db**,pn)) = *person_names*(**db**) + {pn}

[E2] *student_names*(*create_STUDENT*(**db**,sn)) = *student_names*(**db**) + {sn}

[E3] *teacher_names*(*create_TEACHER*(**db**,tn)) = *teacher_names*(**db**) + {tn}

[E4] *course_names*(*create_COURSE*(**db**,cn)) = *course_names*(**db**) + {cn}

preconditions

[P1] *create-PERSON*(**db**,pn)
when pn not_in *person_names*(**db**)

[P2] *create-COURSE*(**db**,cn)
when cn not_in *course_names*(**db**)

[P1] *create-STUDENT*(**db**,sn)
when sn not_in *student_names*(**db**)
and sn in *person_names*(**db**)

[P1] *create-TEACHER*(**db**,tn)
when tn not_in *teacher_names*(**db**)
and tn in *person_names*(**db**)

end spec ExistenceMonitor

natural kind spec UNIVmodel

import
ExistenceMonitor, Clock Persons, Students, Engineering_Students, Teachers, Courses, Enrollments, Eng_Enrollments, Grades, Names

variables
pn, sn, tn, cn: NAME
p: PERSON
t: TEACHER
s: STUDENT
es: ENG_STUDENT
c: COURSE
e: ENROLL
ee: ENG_ENROLL
g: GRADE

communications

[C1] *enroll_course*(s,c) | *enroll*(e,s,c)

[C2] *get_grade*(s,c) | *issue_grade*(e,g) | *give_grade*(t,e,g)
when *student*(e) = s
and *course*(e) = c
and *course*(t) = c

[C3] *get_mid_grade*(es,c) | *issue_mid_grade*(ee,g) | *give_mid_grade*(t,ee,g)
when *student*(ee) = es
and *course*(ee) = c
and *course*(t) = c

[C4] *term*(t) | *term*(e) | tick(clock)
when *day*(*today*(clock)) = 30
and *month*(*today*(clock)) = january or *month*(*today*(clock)) = june

[C5] *mid_term*(t) | *mid_term*(e) | tick(clock)
when *day*(*today*(clock)) = 15
and *month*(*today*(clock)) = april or *month*(*today*(clock)) = november

[C6] *term_remind*(t) | tick(clock)
when *day*(clock)) = *day*(term(t)) + 7*x
and x = 1, 2, 3,....

[C7] *mid_term_remind*(t) | tick(clock)
when *day*(clock)) = *day*(mid_term(t)) + 7*x
and x = 1, 2, 3,....

initializations

[I1] (**db**, ext: empty, old: empty)

end spec UNIVmodel

Comments

For the specifications of natural kinds: Sets, Clock, Identities, Items, Integers, Names and Dates see [Wie89d].

Appendix C2. Example Specification in OBLOG

```
data type grade
    operations
        0,1,2,3,4,5,6,7,8,9: → grade;
        failure: grade → bool;
        success: grade → bool;
    equations
        for x: grade;
        failure(x) = true if x < 6;
        success(x) = true if x ≧ 6
end

homogeneous object type PERSON
    surrogate
        construct(name_of:string)
    template
        events
            birth birth;
            death death
end

homogeneous object type COURSE
    surrogate
        construct(code:string)
    template
        events
            start;
            finish
        state
            attributes
                from_major: string
end
```

Within the specification of type *STUDENT*, we define attributes of type **set**. These constructor is mentioned for the first time in the recent repport [Cos89]. For this reason, in our report, we never mention the operator set.

```
homogeneous object type STUDENT
    type view of PERSON
    template
        events
            birth choose_major(string);
```

```
            death drop_out;
            enroll_course(|COURSE|);
            get_grade(|COURSE|, grade);
        state
            attributes
                major: string;
                courses: set(|COURSE|);
                grades: set(|COURSE|,grade)
            constraints
                leq(5, card(courses)) = true;
        life cycles
            safety
                for C: |COURSE|, G: grade;
                {in?(C,courses) = false and in?((C,G),grades) = false and
                from_major(C) = major and card(courses) < 5} enroll_course;
                {Penroll_course(S)} get_grade(S,G)
            valuation
                for C: |COURSE|, G: grade, M: string;
                [choose_major(M)]major = M;
                [choose_major(M)]courses = empty;
                [choose_major(M)]grades = empty;
                [enroll_course(C)]courses = insert(C,courses);
                [get_grade(C,G)]grades = insert((C,G),grades);
                [get_grade(C,G)]courses = delete(C,courses);
end

homogeneous object type ENG_STUDENT
    selection of STUDENT
        such that major = "engineering"
    type view of STUDENT
        template
            events
                get_mid_grade(|COURSE|, grade)
            state
                attributes
                    mid_grades: set(|COURSE|, grade);
            life cycles
                safety
                    for C: |COURSE|, G, MG: grade;
                    {Penroll_subject(C) and
                    notPget_grade(C,G)} get_mid_grade(C,MG)
                valuation
                    for C: |COURSE|, MG: grade;
                    [choose_major("engineering")] mid_grades = empty;
```

```
                [get_mid_grade(C,MG)] mid_grades =
                    insert((C,MG),mid_grades)
end

object CALENDAR
    template
        events
            birth start_calendar;
            day;
            is_term;
            is_mid_term;
            time_to_remind_term;
            time_to_remind_mid_term
        state
            attributes
                days: nat
        life cycles
            procedure
                start_calendar → CALENDAR
                where CALENDAR =
                    {days≠75+7x & days≠150+7x & x = 0,1,2,...}day→CALENDAR |
                    {days=75}is_mid_term → day → CALENDAR |
                    {days=150}is_term → day → CALENDAR |
                    {days=75+7x & x = 1,2,3,...}time_to_remind_mid_term →
                        day → CALENDAR |
                    {days=150+7x & x = 1,2,3,...}time_to_remind_term →
                        day → CALENDAR
            valuation
                [start_calendar]days = 0;
                [dayfR]days = days+1
end

object ADMINISTRATION
    linked to CALENDAR {
        template map
            events
                is_term to is_term
                is_mid_term to is_mid_term
                time_to_remind_term to time_to_remind_term
                time_to_remind_mid_term to time_to_remind_mid_term
    }
    linked to TEACHER {
        template map
            events
```

```
                is_term to term
                is_mid_term to mid_term
                time_to_remind_term to remind_term
                time_to_remind_mid_term to remind_mid_term
        }
        linked to ENROLLMENT {
            template map
                events
                    is_term to term
                    time_to_remind_term to remind_term
        }
        linked to ENG_ENROLLMENT {
            template map
                events
                    is_mid_term to mid_term
                    time_to_remind_mid_term to remind_mid_term
        }
end

homogeneous object type TEACHER
    type view of PERSON
    template
        events
            birth became_teacher;
            death quit;
            teaches_course(|COURSE|);
            give_grade((|STUDENT|, |COURSE|), grade);
            term;
            remind_term;
            give_mid_grade((|ENG_STUDENT|, |COURSE|), grade);
            mid_term;
            remind_mid_term;
        state
            attributes
                course: |COURSE|
        life cycles
            procedure
                became_teacher → TLIFE
                where TLIFE = teaches_course → T1 → (TLIFE | quit)
                and T1 = {from_major(course) = "engineering"}mid_term → T2 |
                  {from_major(course) ≠ "engineering"}term → T3
                and T2 = give_mid_grade → term → T3 |
                  mid_term_remind → T2 | term → T4
                and T3 = give_grade | term_remind→T3
```

```
                and T4 = give_mid_grade → T3 |
                  (mid_term_remind | term_remind) → T4
            valuation
                for C: |COURSE|
                [give_course(C)]course = C;
end

homogeneous object type ENROLLMENT
    type linked to STUDENT
        surrogate map
            stud
        instance map
            enroll to enroll_course;
            issue_grade to get_grade
    type linked to COURSE
        surrogate map
            crs
    surrogate
        construct(stud:|STUDENT|,crs:|COURSE|)
    template
        events
            enroll(|STUDENT|,|COURSE|);
            issue_grade(|STUDENT|,|COURSE|,grade);
            term;
            term_remind
        life cycles
            safety
                for S: |STUDENT|, C: |COURSE|, G: grade;
                {Pterm}issue_grade(S,C,G);
                {Pterm and not(Pissue_grade(S,C,G)}term_remind
            liveness
                issue_grade(S,C,G)
end

homogeneous object type ENG_ENROLLMENT
    type view of ENROLLMENT
    type linked to ENG_STUDENT
        instance map
            issue_mid_grade to get_mid_grade
    template
        events
            issue_mid_grade(|ENG_STUDENT|,|COURSE|,grade);
            mid_term;
            mid_term_remind
```

```
        life cycles
            safety
                for E: |ENG_STUDENT|, C: |COURSE|, MG: grade;
                {P mid_term} issue_mid_grade(E,C,MG);
                {P mid_term and not(P issue_mid_grade(E,C,MG))} mid_term_remind
            liveness
                issue_mid_grade(E,C,MG)
end

interaction among ENROLLMENT, TEACHER
    for E: |ENROLLMENT|, T: |TEACHER|, S: |STUDENT|, C: |COURSE|, G: grade;

        E.issue_grade((S,C),G) = T.give_grade((S,C),G)

interaction among ENG_ENROLLMENT, TEACHER
    for E: |ENG_ENROLLMENT|, T: |TEACHER|, S: |ENG_STUDENT|, C: |COURSE|,
        MG: grade;

        E.issue_mid_grade((S,C),MG) = T.give_mid_grade((S,C),MG)
```

Appendix C3. Example Specification in MOKUM

```
type personT
    has_a name:string.

type courseT
    has_a code:string
    has_a from_major:majorT.

type stud_gradeT
    has_a course:courseT
    has_a student:studentT
    has_a grade:integer
    restriction grade:value_restr_on_grade
    proc
        value_restr_on_grade:-
            grade ≦ 10,
            grade ≧ 0.
    end_proc.

type eng_gradeT is_a stud_gradeT
    has_a mid_grade:integer
    restriction value:mid_grade_restr
    proc
        mid_grade_restr:-
            mid_grade ≦ 10,
            mid_grade ≧ 0.
    end_proc.

type studentT is_a personT
    has_a stud#:string
    has_a major:majorT
    has_a courses:collection(courseT)
    has_a grades:collection(stud_gradeT)
    restriction birthday:birthday_restriction_on_student
    restriction courses:courses_restr_on_student
    has_a age:integer:=compute_age(birthday)
    proc
        birthday_restriction_on_student:-
            birthday > 19600000,
            birthday ≦ current_date - 150000.
        courses_restriction_on_student:-
            number_of_members(courses) ≦ 5,
```

```
        compute_age(birthday):-
            age is (current_date - birthday)/10000.
    endproc
    script
        initial state
            assign (∅ to courses),
            assign (∅ to grades).
            next_state stud_life
        state stud_life
            at_trigger wish_to_enroll_course(self, C):          (external message)
                number_of_members(courses) < 5,
                not C in courses,
                not O in grades where course = C,
                from_major of C = major of self,
                new(M), add_type(M, st_to_sa_messageT,
                    [(name,create_enroll),(stud,self),(course,C)]),
                send(Student_administration, M),
                insert(C, courses).
                next_state stud_life
            at_trigger get_grade(self,C,G):
                C in courses,
                new(O), add_type(O, stud_gradeT,
                    [(student,self),(course,C),(grade,G)]),
                insert(O, grades),
                delete(C, courses).
                next_state stud_life
    endscript.

type eng_studentT is_a studentT
    has_a egrades: collection(eng_gradesT)
    restriction major:major_restr_on_eng_student
    proc
        major_restr_on_eng_student:-
            major = "engineering",
    script
        initial state
            assign (∅ to courses),
            assign (∅ to egrades).
            next_state eng_life
        state eng_life
            at_trigger eng_wish_to_enroll_course(self, C):      (external message)
                number_of_members(courses) < 5,
                not C in courses,
                not O in egrades where course = C,
```

```
                    from_major of C = "engineering",
                    new(M), add_type(M, st_to_sa_messageT,
                        [(name,create_eng_enroll),(stud,self),(course,C)]),
                    send(Student_administration, M),
                    insert(C, courses).
                    next_state stud_life
                at_trigger eng_get_mid_grade(self,C,MG):
                    C in courses,
                    new(O), add_type(O, eng_gradeT,
                        [(student,self),(course,C),(mid_grade,MG)]),
                    insert(O, egrades).
                    next_state stud_life
                at_trigger eng_get_grade(self,C,G):
                    C in courses,
                    O in egrades where course = C,
                    assign(G to grade of O),
                    delete(C, courses).
                    next_state stud_life
        endscript.

type teacherT is_a personT
        has_a course:courseT.
        has_a students_without_grade:collection(studentT).
        script
            initial_state
                assign(∅ to students_without_grade).
                next_state teaching
            state teaching
                at_trigger end_term(LS):
                    assign(LS to students_without_grade).
                    next_state giving_grades
            state giving_grades
                at_trigger give_grade(S,G):                    (external message)
                    S in students_without_grade
                    new(M), add_type(M, tc_to_sa_messageT,
                        [(name,issue_grade),(stud, S), (course,course of self),(grade,G)]),
                    send(Student_administration,M),
                    delete(S, students_without_grade).
                    next_state giving_grades
                at_trigger end_term_remind:
                    next_state giving_grades
        end_script.

type eng_teacherT is_a teacherT
```

```
    has_a students_without_mid_grade:collection(studentT)
    restriction course:course_restr_on_teacher
    proc
        course_restr_on_teacher:-
            from_major of course ="engineering".
    script
        initial_state
            assign(∅ to students_without_grade),
            assign(∅ to students_without_mid_grade).
            next_state teaching
        state teaching
            at_trigger end_mid_term(self,LS):
                assign(LS to students_without_mid_grade)
                next_state giving_mid_grades, teaching
            at_trigger end_term(self,LS):
                assign(LS to students_without_grade)
                next_state giving_mid_grades, giving_grades
        state giving_mid_grades
            at_trigger give_mid_grade(S,G):                    (external message)
                S in students_without_mid_grade,
                new(M), add_type(M,tc_to_sa_messageT,
                    [(name,issue_mid_grade),(stud, S),
                    (course,course of self),(grade,G)]),
                send(Student_administration,M),
                delete(S, students_without_mid_grade).
                next_state giving_mid_grades
            at_trigger mid_term_remind:
                next_state giving_mid_grades
        state giving_grades
            at_trigger give_grade(S,G):                        (external message)
                S in students_without_grade,
                not S in students_without_mid_grade,
                new(M), add_type(M,tc_to_sa_messageT,
                    [(name,issue_grade),(stud, S), (course,course of self),(grade,G)]),
                send(Student_administration,M),
                delete(S, students_without_grade).
                next_state giving_grades, giving_mid_grades
            at_trigger end_term_remind:
                next_state giving_grades, giving_mid_grades
    end_script.

type EnrollmentT
    has_a student:studentT
    has_a course:courseT.
```

```
type student_administrationT
    has_a enrollments: collection(enrollmentT)
    script
        initial state
            assign(∅ to enrollments).
            next_state accepting_commands
        state accepting_commands
            at_trigger create_enroll(S,C):
                not O in enrollments where student=S and course=C,
                new(E), add_type(E, enrollmentT,
                    [(student,S), (course,C)]).
                next_state accepting_commands
            at_trigger create_eng_enroll(S,C):
                not O in enrollments where student=S and course=C,
                major of S = "engineering",
                from_major of C = "engineering",
                new(O), add_type(O, enrollmentT,
                    [(student,S), (course,C)]).
                next_state accepting_commands
            at_trigger issue_grade(T,S,C,G):
                E in enrollments where student=S and course=C,
                course of T = C,
                new(M),add_type(M,sa_to_st_messageT,
                    [(name,get_grade),(stud,S), (course,C),(grade,G)]),
                send(S, M),
                delete(E, enrollments).
                next_state accepting_commands
            at_trigger issue_eng_grade(T,S,C,G):
                E in enrollments where student=S and course=C,
                course of T = C,
                from_major of C = "engineering",
                major of S = "engineering",
                new(M),add_type(M,sa_to_st_messageT,
                    [(name,eng_get_grade),(stud,S), (course,C),(grade,G)]),
                send(S,M),
                delete(E, enrollments).
                next_state accepting_commands
            at_trigger issue_eng_mid_grade(T,S,C,MG):
                E in enrollments where student=S and course=C,
                course of T = C,
                from_major of C = "engineering",
                major of S = "engineering",
                new(M),add_type(M,sa_to_st_messageT,
                    [(name,eng_get_mid_grade), (stud,S),(course,C),(grade,MG)]),
```

```
                send(S, M).
                next_state accepting_commands
            at_time end_term:
                for_all T in teacherT do:
                    assign((select(student in enrollments where course=course of T) to
                    L),
                    new(M),add_type(M,sa_to_tc_messageT,
                        [(name,end_term),(enrolled_students,L)]),
                    send(T,M).
                next_state accepting_commands
            at_time end_term + 7*N:
                for_all T in teacherT do:
                    new(M),add_type(M,sa_to_tc_messageT,
                        [(name,term_remind)]),
                    send(T,M).
                next_state accepting_commands
            at_time end_mid_term:
                for_all T in eng_teacherT do:
                    assign((select(student in enrollments where course=course of T) to
                    L),
                    new(M),add_type(M,sa_to_tc_messageT,
                        [(name,end_mid_term),(enrolled_students,L)]),
                    send(T,M).
                next_state accepting_commands
            at_time end_mid_term + 7*N:
                for_all T in eng_teacherT do:
                    new(M),add_type(M,sa_to_tc_messageT,
                        [(name,mid_term_remind)]),
                    send(T,M).
                next_state accepting_commands
        endscript.

type messageT
        has_a name:string
        has_a sender:thing.

type st_to_sa_messageT is_a messageT
        has_a stud:studentT
        has_a course:courseT.

type sa_to_st_messageT is_a messageT
        has_a stud:studentT
        has_a course:courseT.
        has_a grade:integer
```

type *sa_to_tc_messageT* **is_a** *messageT*
    has_a *enrolled_students*:**collection**(*studentT*)

type *tc_to_sa_messageT* **is_a** *messageT*
    has_a *stud*:*studentT*
    has_a *course*:*courseT*.
    has_a *grade*:integer

Appendix C4. Example Specification in TAXIS

```
VARIABLE_CLASS PERSON with
    keys
        name
    characteristics
        name: PERSON_NAME
end

VARIABLE_CLASS COURSE with
    keys
        code
    characteristics
        code: COURSE_NAME
        from_major: MAJOR_NAME
end

VARIABLE_CLASS STUDENT is_a PERSON with
    keys
        stud#;
    characteristics
        stud#: INTEGER;
        major: MAJOR_NAME;
    attribute_properties
        age: {|16::50|}
        current_courses: {|0::5|}          /*currently taken courses*/
end

VARIABLE_CLASS ENG_STUDENT is_a STUDENT with
    characteristics
        major: {"engineering"};
end

VARIABLE_CLASS TEACHER is_a PERSON with
    attribute_properties
        course: COURSE_NAME
end
```

The next class implements the set of courses taken by each student. It must be defined as a class because TAXIS does not support set-valued attributes.

```
VARIABLE_CLASS STUDENT_COURSE with
    keys
```

```
        sc: (stud, course);
    characteristics
        stud: STUDENT;
        course: COURSE
    attribute_properties
        grade: {|0::10|}
end

VARIABLE_CLASS ENG_COURSE is_a STUDENT_COURSE with
    characteristics
        stud: ENG_STUDENT;
    attribute_properties
        mid_grade: {|0::10|}
end

TRANSACTION_CLASS ENROLL_COURSE with
    parameter_list
        enroll_course(s, c);
    locals
        s: STUDENT;
        c: COURSE;
    prereqs
        available_course?: c.from_major < s.major
        student_not_max?: s.current_courses < 5;
    actions
        enroll:
            insert_object in STUDENT_COURSE with
                stud←s, course←c;
        increment_courses: s.current_courses←s.current_courses+1;
end

TRANSITION_CLASS GET_GRADE with
    parameter_list
        get_grade(s,c,g)
    locals
        s: STUDENT;
        c: COURSE;
        g: {|0::10|}
    prereqs
        is_enrolled?: (s,c) instance_of STUDENT_COURSE
        not_eng_student?: s.major ≠ "engineering" exc
            IS_ENG_STUD(stud: s)
    actions
        exc_handler for IS_ENG_STUD is
```

```
            ENG_GET_GRADE(s,c,g)
        issue_grade: (s,c).grade←g
        decrement_current_courses: s.current_courses←s.current_courses-1;
end

TRANSACTION_CLASS ENG_GET_GRADE with
    parameter_list
        eng_get_grade(s,c,g)
    locals
        s: ENG_STUDENT;
        c: COURSE;
        g: {|0::10|}
    prereqs
        got_mid_grade?: (s,c).mid_grade=mg instance_of ENG_COURSE
    actions
        issue_grade: (s,c).grade←g
        decrement_current_courses: s.current_courses←s.current_courses-1;
end

TRANSITION_CLASS ENG_GET_MID_GRADE with
    parameter_list
        eng_get_mid_grade(s,c,mg)
    locals
        s: ENG_STUDENT;
        c: COURSE;
        mg: {|0::10|}
    prereqs
        eng_student?: s.major = "engineering"
        is_enrolled?: (s,c) instance_of ENG_COURSE
    actions
        issue_mid_grade: (s,c).mid_grade←mg
end
```

Appendix C5. Example Specification in GALILEO

```
University := (
    rec type Grade ⇔
        (course: Course
        and grade: num this within (0,10))

    and type Egrade ⇔
        (is Grade
        and mid_grade: num this within (0,10))

    and Persons class
        Person ↔
            (Name: string
            and Birthday: num)
        key (Name)

    and Courses class
        Course ↔
            (Code: string
            and from_major: string)
        key (Code)

    and Students subset of Persons class
        Student ↔
            (is Person
            and stud#: num
            and Major: string
            and courses: optional seq Course
                this with count(courses) ≦ 5
            and grades: optional seq Grade)
        key (stud#)

    and Eng_students restriction of Students
        with Major = "engineering"
    class
        Eng_student ↔
            (is Student
            ext grades: optional seq Egrade)

    and Teachers subset of Persons class
        Teacher ↔
            (is Person
```

```
        and course: var Course)
    key (name)

and Enrollments class
    Enrollment ↔
        (student: Student
        and course: Course)
    key (student,course)

use fun enroll_course(s: Student, c: Course): Enrollment:=
    if
        count(courses of s)<5
        and major of s=from_major of c
        and no x in Enrollments with (student=s and course=c)
    then
        (c::(courses of s)
        and mkEnrollment
                (student:=s
                and course:=c))

use fun eng_enroll_course(s: Eng_student, c: Course): Enrollment:=
    if
        count(courses of s)<5
        and from_major of c="engineering"
        and no x in Enrollments with (student=s and course=c)
    then
        (c::(courses of s)
        and mkEnrollment
                (student:=s
                and course:=c)

use fun issue_grade(s: Student, c: Course, g:num this within (0,10)):=
    if
        x in Enrollments with (student=s and course=c)
        and not s alsoin Eng_students
    then
        (course:=c and grade:=g)::(grades of s)
        and remove c from courses
        and remove e from Enrollment with
                (student:=s
                and course:=c)

use fun issue_eng_grade(s:Eng_student, c:Course, g:num this within (0,10)):=
    if
```

```
        x in Enrollments with (student=s and course=c)
        (mid_grade of grades of s with course=c)≧0
    then
        (grade of grades of s with course=c):=g
        and remove c from courses
        and remove e from Enrollment with
                (student:=s
                and course:=c)

use fun issue_eng_mid_grade(s:Eng_student, c:Course, g:num this within
(0,10)):=
    if
        x in Enrollments with (student=s and course=c)
    then
        (course:c and mid_grade:g)::(grades of s)
);
```