ICOT Technical Report: TR-590

TR-590

Towards a Methodological Guide for the Design of Object Oriented Programs

by
Annya Romanczuk

September, 1990

© 1990, ICOT

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Towards a Methodological Guide for the Design of Object Oriented Programs

Annya ROMANCZUK[*]

Institute of New Generation Computer Technology (ICOT)[†]

August, 1990

Abstract

This report is an abstract of my thesis [5] designed at the University of Lille (FRANCE). This research proposes to incorporate the object oriented programming approach in a development method for large systems. The objective is to define, in a methodological way, a guide-line for the analysis and development of object oriented programs.
The use of this guide leads to the creation of a model of the real world directly deduced from the original problem space. It is obvious that the more the solution space fits the real world abstraction, the better we will reach some of the Software Engineering aims: modifiability, efficiency, reliability and understandability.
The convergence of the study on Software Engineering, algebraic requirements definitions, object oriented languages and natural language analysis is used to design the methodological guide-line.
This guide-line induces the development of programs in an incremental way, in the sense that we integrate step by step the specificities of object oriented programming.

Keywords: Methodological approach, software engineering, programming environment, object oriented analysis, algebraic requirements definitions, object oriented language and object oriented programming.

[*]Works realised with CRIL Paris and Univ. of Lille I. CRIL address: 146, bd de Valmy, 92707 Colombes, FRANCE — University of Lille address: Bât. M3, Cité scientifique, 59655 Villeneuve d'Ascq, FRANCE

[†]address: 4-28, Mita 1-chome, Minato-ku, Tokyo 108, JAPAN — Telex (81 3) 456 1618 — Tel (81 3) 456 31 93 — E-mail: annya%icot.jp@relay.cs.net

Contents

List of Figures

1 Introduction

This report discusses a method for the design of object oriented programs, from data analysis to programming with an object oriented language (OOL), through various specification steps (also called requirement definition steps).

This method induces an **incremental process**, in the sense that the specificities of the object oriented programming are integrated step by step in the various proposed steps.

Our method is used from original descriptions of a problem, i.e. precise but informal descriptions in natural language (English or French).

The use of this method leads to the creation of a model of the real world directly deduced from the original problem space, and intuitively, it's obvious that the more closely the solution space (in this case, the set of objects, the operations on these objects, and their properties that are present in the problem space and that affect the finding of a solution [2]) fits the real world abstraction, the better we will reach some Software Engineering aims: modifiability, efficiency, reliability, and understandability.

The convergence of the study on Software Engineering, algebraic requirements definition, object oriented languages, and natural language analysis is used to design the method.

This method can be used through a guide line developed with this method itself, and allows designers to **analyse the original problem descriptions** and to make an **object oriented decomposition** (independently from the OOL), to **define specifications** understandable by the customer or the user and by the developer, and then, from those specifications, to do an **automatic generation of code in the OOL choose:** KEOPS [3], based on Le-Lisp language.

More precisely, this approach consists of four steps:

- The analysis which is used to emphasize the need to map the "solution" on the original problem space, so, on the real world abstractions.
- A first specification step (requirement definition step) defined from a language named *Description Language* (DL) (static specification), which is used to define an algebraic type specification of the problem.
- A second specification step defined from a language named *Control Language* (CL) (dynamic specification) which is used to define controls which can be executed at any time during the program execution.
- During these steps, some verifications are done on the consistency of the given information, and some KEOPS code is automatically generated from the two first problem descriptions (with DL and CL).

Then, as the result of the guide use, we obtain Keops files satisfying the static and dynamic specifications, and we just need to write method bodies.

Before specifying each of these steps, see figure 1, the general diagram of the methodological guide functioning. This will point out the sequence of the various steps of the method.

In the tool, an interface is used to have direct access to the different steps.
As you can see in figure 1, some partial definition or backtracking is possible. With that, we want to insist on the incremental aspect of the method.

Then, this article will present a brief historic of this work and show why such a method is defined, and the four steps of the method.
This method will be presented first from the theoretical point of view then through a concrete example. This example represents the problem of the evaluation of the responsibility of drivers of vehicles when accidents happen.

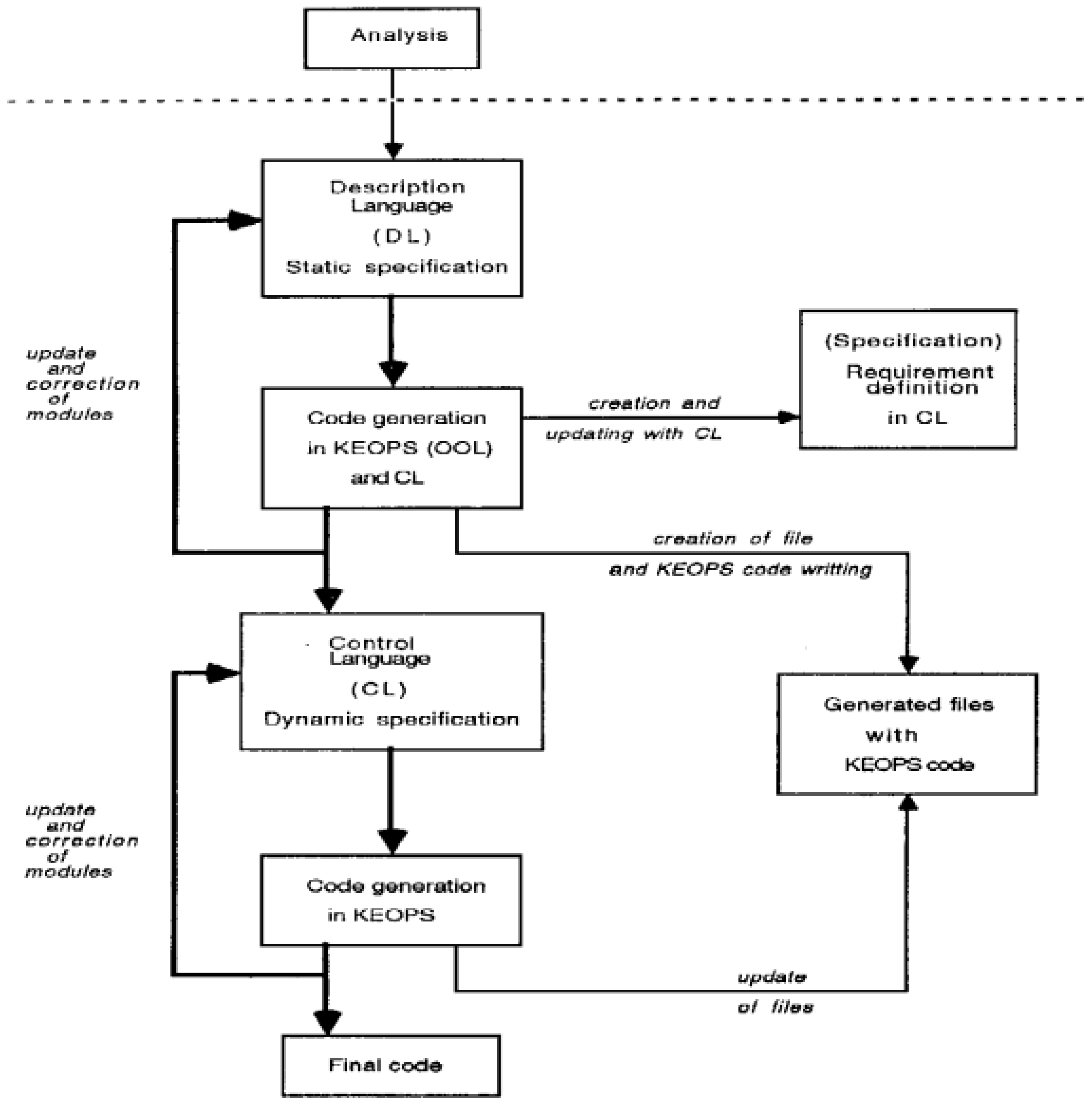

Analysis
Description Language (DL) Static specification
update and correction of modules
Code generation in KEOPS (OOL) and CL
creation and updating with CL
(Specification) Requirement definition in CL
creation of file and KEOPS code writting
Control Language (CL) Dynamic specification
Generated files with KEOPS code
update and correction of modules
Code generation in KEOPS
update of files
Final code

Figure 1: General diagram of the methodological guide functioning

2 Brief historic

The analysis of languages named "object oriented" shows that it is impossible, because of the diversity of their concepts and their objectives of realization, to define a general method for object oriented programming that can be adapted to any type of OOL and to any type of problem.
It is true that all OOL have common basic concepts (classes, methods, inheritance mechanism, instantiation mechanism, ...), but each of these languages wants to bring more.
It appears that a wide variety of concepts specific to some languages such as the metaclasses in Smalltalk or Loops, the production rules in Kool, the frames in Keops....
Of course, the significance of these languages is in their various application domains: simulation, A.I., system ...

Before a problem can be resolved, the most suitable language must be chosen. To do that, the problem needs must be defined, and the specificities of the various languages named "object oriented" must be known. However, the significance of some such basic concept can remain vague for someone who is unused to it.

More important is how to build an application with one of these languages ? The answer is as much difficult as there are many programming design "directives", when they exist, as languages named "object oriented" consecrate to such and such types of problems.

Moreover, all the OOL are based on other languages. Objective-C on C [11], KEOPS on Le-Lisp, Object-Pascal [12] on Pascal and so on.
Those programming languages introduce their own programming style, such as recursive programming, or functionnal programming. **Unfortunately, this also arises when we use the OOL based on these languages and it is really difficult to abstract the programming methodology underlying the OOL development.**

Also, each programmer has a particular computer background, and abstracting their programming habits in the face of an OOL without methods or tools for programming design is difficult.
We can point out that **it is impossible to think about a general programming method for these various types of languages named "object oriented"**, which are mostly developed in function for the needs of the moment.

Nevertheless, because of their common basic concepts, the design of a basic method for the design of object oriented programs, extendable to each "category" of "object oriented" languages is viewed.
Proposing specifications, this method abstracts for the first time the next implementation language, and in particular the language's specificities.

3 First step of the method: Analysis

This analysis approach is used to develop problem specifications from informal but precise decriptions in natural language (English or French). It is based also on the grammatical analysis of texts, and it is defined from the approach first proposed by Abbot, J.R., in [1] and developed by Booch, G., [2] and Vogel, C., [13].

> *"If we examine human languages, we find that they all have two primary components, noun phrases and verb phrases. A parallel structure exists in programming languages, since they provide constructs for implementing objects (noun phrases) and operations (verb phrases).*
> *However, most of the languages are primarily imperatives, that is, they provide a rich set of constructs for implementing operations but are generally weak when it comes to abstracting real word objects"*. BOOCH, G. in [2].

The essential approach of this method is to take into account the correspondence established between the real world and the software solution; it mainly consists of a reduction of the problem descriptions.

The next sub-sections describe this analysis method of an original description of a problem. Thanks to that method, we will get a program which implements a model of a real world as a set of objects which interact between themselves. The solution space is expressed here with graphic representations of object. The formalism of these graphic representations is defined in [5] and [7].

This method is used to develop programs from an original description in natural language. It shows how to design data types from common nouns, objects from direct references, properties or functionalities from verbs, and attributes and control structures from their equivalents in terms expressed in natural language.

For instance, we assimilate common nouns as a natural language, analogous to some programming language notions implementing abstract data types. This fits with an extension of standard understanding of the abstract data types. As a matter of fact, a type is usually used as a collection of values and a collection of operations on these values [4]. But common nouns are more than that. A common noun names a class or a concept even if there is no value, no operation yet defined on this class or this concept. Then, common nouns are used to declare the existence of a concept whose characteristics have to be developed.

The association between common nouns and abstract types makes more intuitive the notion of data types. If it is true that a large set of people have an intuitive understanding of common nouns, the concept of abstract data types is clarified by analogy with these common nouns.

Usually, the methodologies of program design are not easy because they insist on the fact that an abstract data type is defined as values and operators. But in natural languages, we frequently create and use common nouns before we have thought about all details of the concept.

Our analysis method consists of four steps :

1. The definition of problem descriptions, precise and clear, using specific terms of the problem domain.
2. The identification of the types, objects, and operators.
3. The rewriting of the problem descriptions from the elements pointed out during the identification step, and the organization of the operators defining the control structures found in the problem descriptions.
4. The definition of the elements used for the solution space of the problem. (This solution space will be represented with graphics)

The method can appear mechanical, but it is improbable that the method could be made automatic. Even if the identification method of types, objects, operators, and control structures is made easier by the problem descriptions in natural language, it requires a good knowledge of the real world and an intuitive understanding of the text. That is, the semantic aspect of the text should completed by the syntactic information. Probably it is this significant interpretation part of the text which considerably complicates the automation of a such process.

3.1 Description of our problem

The first step is to define clear and precise descriptions of the problem. Our problem is a sub-problem of the evaluation of the responsibility of drivers of vehicles when accidents happen. This sub problem is used in all companies of car insurance. The systemization of this evaluation has been done during the feasibility study of an expert system.

We will study the evaluation process of the responsibility of drivers of motor vehicles when accidents happen involving two vehicles.

When they are maneuvering legally, if vehicles are in two-way traffic, trespass over the median line of the road determines their responsibilities.

If the two vehicles are not in infraction:
If one of them trespasses over the median line of the road or overtakes it, his responsibility is full.
If both trespass, their responsibility is divided equally.

If one of the two vehicles doesn't respect the road signs, his responsibility is full. If both are in this case, their responsibilty is divided equally.

3.2 Identification of types, objects, and operators

The identification steps are :

- Identification of abstract data types
- Identification of objects of these data types
- Identification of operators characterizing these types and objects

Before defining each of these steps, we must remember that type, object, and operator identification is not a simple grammatical analysis of text, but requires a good knowledge and an intuitive undestanding of the problem domain.

3.2.1 Identification of abstract data types and objects

For the first identification step, which consists of pointing out abstract data types and objects, it is requisite to identify nouns and nominal groups in the original problem description. To help to understand this relation between the original descriptions and the software solution, take us off the problem itself to explain the various types of nouns and nominal groups. This presentation has its source in Abbott, J.R, [1], which points out that in the description of our real world model, we can find 3 nominal group types:

- **Common nouns** name a class of things or persons. *For instance: language, house ...*
- **Proper nouns** and **direct references:**

  A proper noun names a specific person or a specific thing. *For instance: Eiffel tower, Smalltalk, ...* Direct references are references to a specific person or a specific thing, previously defined without necessarily having referred to them by a noun. *For instance: my pen, the computer screen ...*
- **Nouns expressing quantities** and **units of measure**:

  Nouns expressing quantities name a quality, an activity, or a substance. *For instance: air, music, ...* Nouns expressing units of measure are used to name the quantity of a quality, an activity, or a substance. *For instance: bars (of music), humidity (of air) ...*

Common nouns are used to identify the **object classes** that we characterize here as **abstract data types**. **Units of measure** name arbitrary units which are common for everybody, and are used to subdivide what is referred to by the nouns expressing quantities. So they correspond to **types defined from numeric types**. Like nouns expressing quantities, they name qualities, substances and activities which don't have apriorism any organization between their units. These nouns expressing quantities are divided into units.

Objects are named by the **proper nouns** and **direct references** of the original descriptions of the problem.

Definitions and precise uses of those terms cannot be given in this papers, but an entire description of those terms and the analysis identification step can be found in [5] in French and [6] in English.

The example given here will help the understanding of this identification step.

In the next problem description, expressions giving indications on abstract data types are in bold print and those giving indications on objects are underlined.

We will study the **evaluation process** *of the responsibility* of **drivers** of **motor vehicles when accidents happen involving two vehicles.**

When they are **maneuvering** legally, if **vehicles** are in two-way traffic, **trespass over the median line of the road** determines their **responsibilities.**

If **the two vehicles** are not in infraction :
If one of them trespasses over the **median line of the road** or overtakes it, **his responsibility is full.**
If **both** trespass, **their responsibility is divided** equally.

If one of the two **vehicles** doesn't respect the road signs, his **responsibility is full.** If **both** are in this case, **their responsibility** is divided equally.

The terms which give the next abstract types from a first analysis of the original descriptions are *evaluation_process, responsibility, drivers* (of motor vehicles), *vehicles* (motor vehicles), *accidents* (happen involving two vehicles), *two vehicles* and *maneuvering.*
Here, we can reduce *drivers* and *motor vehicles* as *vehicles.*

As first abstract data types we obtain:
Evaluation_process

Responsibility
Vehicle
Accidents_involving_two
Pair_of_vehicles (from two vehicles)
Maneuvering
Trespass_median_line
Median_line
Road_sign

In the set of terms giving the next objects, no proper nouns exist, but the reconized direct references are: *the two vehicles*, *one* (of them), *responsibility is full*, *the other* (vehicle) and *responsibility is divided equally*.

We define from these direct references, the objects associated to their abstract type:
one_pair : → Pair_of_vehicles
vehicle_1 : → Vehicle (from one vehicle)
vehicle_2 : → Vehicle
responsibility_is_full : → Responsibility
responsibility_is_divided_equally : → Responsibility

These objects and abstract data types represent a first approximation. We will see that some of them are not useful for our sub problem, or that some objects or abstract data types are missing.

Of course, we can find another solution according to what we consider as essential in the problem context. Another solution is presented in [5] and in [8].

3.2.2 Identification of operators

The operators are used to characterize the behaviour of each object or object class. Here, we establish the semantic of object in determining the characteristics or operations which can be executed on the object or by the object.

For that, we must point out verbs, attributes, predicates and descriptive expressions from the original problem descriptions:

- **A verb** expresses that somebody or something exists, that is in such and such a state, or that does such and such an action. In our method, we will see through our exemple, that a verb is represented by a property characterizing the behaviour of an object of the abstract data type already defined or on which the action will be executed. The arguments are defined by this object or this abstract type and by the objects requisite for the execution of this action. The return value is the object on which the action has been executed, then possibly modified.
- **An attribute** shows the quality, good or bad, that we give or refuse to the subject through one verb: *This story is funny*. It is usually linked to the subject by the verbs: to be, to look like, to seem, to appear, to be taken for ... In our method, we think of an attribute as an abstract type property, which can be applied to all the elements (objects) of that type, and which has a return value equal to the value of this attribute.
- **A predicate** points out a property or a relation which can be considered as a boolean (true or false). In our tool, a predicate is represented as a property which characterizes the abstract type or the object on which it is applied, with, as arguments, the objects requisite to know the value of the predicate, and as return value the object or the abstract type used and its boolean value.
- **A descriptive expression** gives a description of one or several objects which cannot be known. The descriptive expressions are usually represented as properties characterizing the object or the set of objects (abstract type) describes, with as arguments, some characteristics of the described object, and as return value the object that has this description.

Remark: For all the operators defined from the original descriptions of the problem, the output value is an object of the abstract type, or the abstract type on which the operator is linked. Effectively, the action associated to this operator can (or cannot) modify the object or abstract type state on which it has been executed. It is requisite to return this type or this object which has its state modified. We can also explicitly ask another return value, which will be added to the concerned object or type.

We will take the same problem description to point out in bold type the verbs, attributes, predicates and descriptive expressions.

We will **study the evaluation process** *of the responsibility of drivers of motor vehicles when accidents happen involving two vehicles.*

When they are maneuvering legally, *if vehicles are in two-way traffic*, **trespass over the median line of the road determines their responsibilities.**

If the two vehicles **are not in infraction**:
If **one of them trespasses over the median line** *of the road or* **overtakes it, his responsibility** *is full.*
If both **trespass, their responsibility** *is divided equally.*

If **one of the two** *vehicles* **doesn't respect the road signs, his responsibility** *is full. If both* **are in this case, their responsibility** *is divided equally.*

The next operators represent a first approximation of what will be the functionalities and the characteristics of the objects needed for the evaluation of our program:

Study_the_evaluation_process: Operator which characterizes the behaviour of the type *Evaluation_process*, with an object from the type *Evaluation_process* and two objects from the type *Vehicle* as arguments.

Maneuvering_legally: Operator which characterizes each object from the type *Vehicle* by a boolean value.

Which_responsibility_?: Operator which characterizes the behaviour of *Trespass_median_line* with an object from the type *Trespass_median_line* and an object from the type *Pair_of_vehicles* as arguments. The return value is the object from the type *Tresspas_median_line* which has been treated.

His_responsibility: Operator which characterizes each object from the type *Vehicle* by the value of its responsibility (full or divided in 2).

2_are_not_in_infraction: Operator which is used to get information on an object from the type *Pair_of_vehicles* with, as argument, the present pair and, as return values, this pair treated and a boolean value.

If_1_only_trespass_median_line: Operator which tests an object from the type *Pair_of_vehicles*, updates this one, and sends as return value the modified object.

If_1_only_overtakes_median_line: Operator which tests an object from the type *Pair_of_vehicles*, updates this one, and sends as return value the modified object.

Verify_1_doesn't_trespass_and_1_trespass: Operator which characterizes the behaviour of the objects from the type *Pair_of_vehicles* and returns as output value this verified object.

If_2_trespass_median_line: Operator which tests an object from the type *Pair_of_vehicles*, updates each attributes *responsibility* of the two vehicles of the pair, and returns as output value the tested object from the type *Pair_of_vehicles*.

1_doesn't_respect_road_signs: Operator which tests an object from the type *Pair_of_vehicles*, updates the attribute *responsibility* of this object, and returns as output value the modified object from the type *Pair_of_vehicles*.

2_doesn't_respect_road_signs: Operator which tests an object from the type *Pair_of_vehicles*, updates each attributes *responsibility* of the two vehicles of the pair, and returns as output value the tested object.

car_1: Operator which characterizes an object from the type *Pair_of_vehicles* by an object from the type *Vehicle*. Remarks that this attribute, and the following, come from the term "one of the two" which points the components of a pair of vehicles.

car_2: Operator which characterizes an object from the type *Pair_of_vehicles* by an object from the type *Vehicle*.

We try to simplify these operators which are synonymous :
The operators *2_are_not_in_infraction*, *1_doesn't_respect_road_signs* and *2_doesn't_respect_road_signs* are replaced by *2_in_legal_maneuverings*, *1_only_in_illegal_maneuverings* and *2_in_illegal_maneuverings*. This to be near the attribute *maneuvering_legally*, and because its boolean value is known we can answer the three preceding questions.

At this level, the resolution method of the problem (corresponding to the problem descriptions) should be made more precise by the defined information, that is, the abstract types, the objects, and the operators. Effectively, we can verify that not all the concepts introduced by the problem descriptions are requisite for the solution space, or perhaps that some are missing. This step is used to make an adjustment: that is, to suppress the concepts that are not requisite and add those that are missing. In general, the concepts correspond to implicit notions that this step points out.

3.3 Rewriting the problem descriptions

After the update, we can rewrite the problem descriptions with the objects, operators and abstract data types found. Only the control structures stay the same as those in the first problem descriptions:

Ask the **Evaluation_process** *to* **Study_the_ evaluation_process (car_1, car_2)** *concerning the responsibility of drivers when accidents happen between two vehicles that form* **one_pair** *(Pair_of_vehicles).*

According to the **Trespass_median_line,** *determine* **Which_responsibility_? (one_pair)** *will have each of the two vehicles.*

If in **one_pair,** *the* **2_in_legal_maneuverings** *:*
when
- **If_1_only_trespass_median_line** *or* **If_1_only_ overtakes_median_line** *in* **one_pair, his_responsibility** *is equal to* **responsibility_full.**
- **If_2_trespass_median_line** *in* **one_pair,** *each of the two vehicles,* **car_1, car_2,** *has* **his_responsibility** *equal to* **responsibility_divided_in_equally.**

If in **one_pair, 1_only_in_illegal_maneuverings, his_ responsibility** *is equal to* **responsibility_full.**

If in **one_pair, 2_in_illegal_maneuverings, his_ responsibility** *is equal to* **responsibility_divided_in_equally.**

3.4 Updating the elements and the graphic representation of the solutio n space

3.4.1 Updating the abstract types

Some abstract types defined from the analysis of the original descriptions of the problem, are not really used in the program progress:

Maneuvering This type which has no object and no operator is introduced in the original descriptions to characterize the vehicles on the legality or otherwize of their maneuvers. So it is assimilated to the boolean type. The boolean values are used as the value of the attribute *legal_maneuverings* of the vehicles.

Median_line Common noun which is not used in our problem.

Road_sign Common noun which is not used in our problem.

Accidents_involving_two It is not used, but only to specify the context in which the study of the responsibility will evolve.

As information, we can say that the abstract types not kept here (because the original descriptions are a subset of the general problem) should probably be kept for the treatment of the complete program.

The type *Responsibility* has only two objects ***responsibility_is_full*** and ***responsibility_is_divided_in_equally***, and no operator. So we can replace this type by the type ***string*** because the program uses only the name of these objects as the value of the attribute ***his_responsibility*** of the vehicles. Our generic perspective leads us to keep the type *Responsibility*, and to use its objects in a such sense.

The abstract types defined are:
Evaluation_process
Vehicle
Pair_of_vehicles
Trespass_median_line
Responsibility

3.4.2 Updating the objects

To study the evaluation processus, we must define our own *Evaluation_process* which we call ***eval_proc_2_veh***. Also, for the trespass of the median line, we must define our own symbol for the ***trespass_median_line***, which we call ***trespass_for_pair***.

The objects defined are:
Eval_proc_2_veh : → Evaluation_process
Vehicle_1 : → Vehicle
Vehicle_2 : → Vehicle
One_pair : → Pair_of_vehicles
Responsibility_is_full : → Responsibility
Responsibility_is_divided_equally : → Responsibility
Trespass_for_pair : → Trespass_median_line

3.4.3 Updating the operators

The attributes *Trespass_median_line* and *Overtake_median_line* with boolean value, which characterize the vehicles, should be known in the problem to be able to answer to the questions : *If_1_only_trespass_median_line*, *If_1_only_overtakes_median_line* and *If_2_trespass_median_line*. Effectively, the original descriptions of the problem don't give any way to solve them. So, at the analysis level, two operators are missing:
Trespass_median_line : Operator which characterizes each object from the type *Vehicle* by a boolean value.
Overtake_median_line : Operator which characterizes each object from the type *Vehicle* by a boolean value.

The operator *Verify_1_doesn't_trespass_and_1_trespass* characterizing the type *Pair_of_vehicles*, is not requisite. Effectively, it is in the original descriptions only as verification or information, but is not used for the software solution.

The defined operators are:

For the type *Evaluation_process*
Study_evaluation_process (vehicle_1, vehicle_2)

For the type *Vehicle*
Maneuvering_legally
Trespass_median_line
Overtake_median_line

His_responsibility

For the type *Pair_of_vehicles*
Car_1
Car_2
If_2_trespass_median_line
If_1_only_overtakes_median_line
If_1_only_trespass_median_line
2_in_legal_maneuverings
1_only_in_illegal_maneuverings
2_in_illegal_maneuverings

For the type *Trespass_median_line*
Which_responsibility_? (one_pair)

Now that all concepts are defined, we can express our solution space with graphic representations.

3.4.4 The graphic representation of the problem solution space

In the remainder of this report, we will not develop the entire solution space; we will show only the graphic representation of the two most important abstract data types of our problem solution space. See figures 2 and 3.

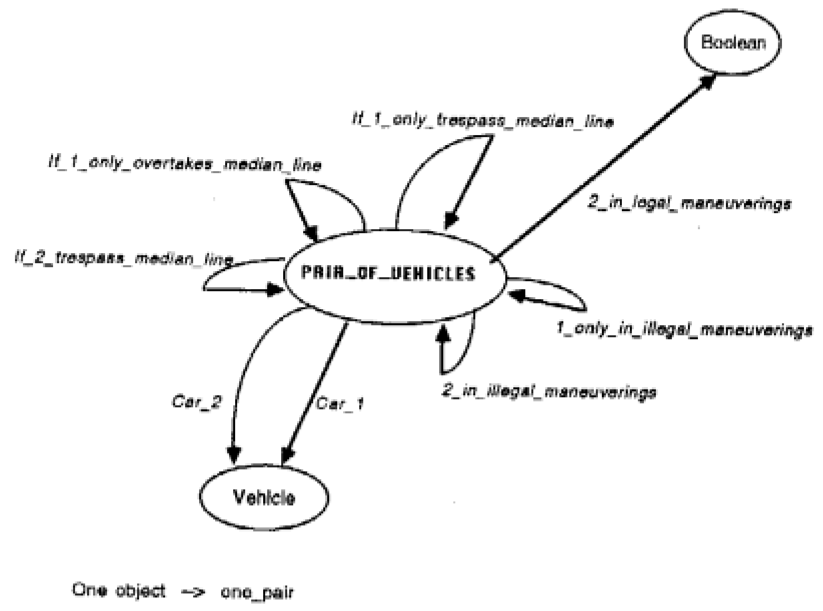

Boolean
If_1_only_trespass_median_line
If_1_only_overtakes_median_line
2_in_legal_maneuverings
If_2_trespass_median_line
PAIR_OF_VEHICLES
1_only_in_illegal_maneuverings
2_in_illegal_maneuverings
Car_2
Car_1
Vehicle
One object --> one_pair

Figure 2: Abstract data type : Pair_of_vehicles

From these graphic representations, it is easy to pass to the next step: the static specification (see figure 1).

4 Second step of the method: Static Specification

In order that programming satisfy the aims introduced by Software Engineering, a solution is a correct use of the abstractions. But, without a specification tool to help, the abstraction notion is too intangible to be useful.

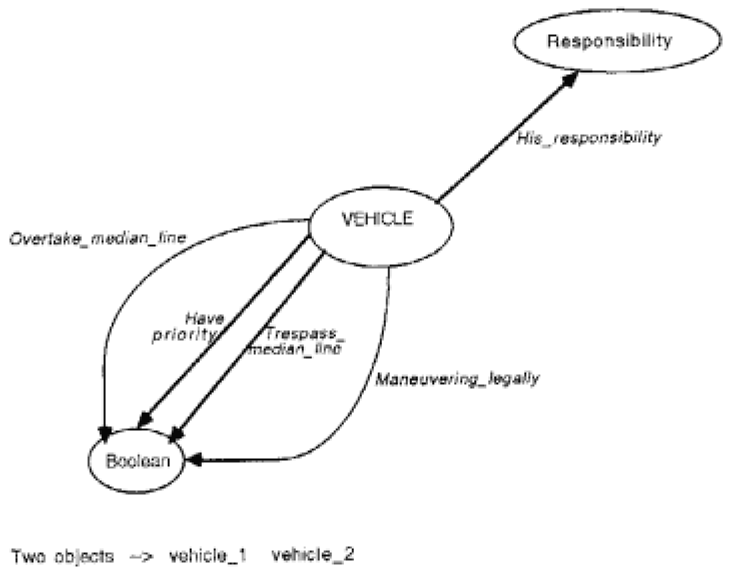


Figure 3: Abstract data type : Vehicle

As figure 1 shows, after the analysis step, we can now go into the static specification step. After specifying the problem in a static way, we can obtain an automatic generation of code. This code will be in Keops code and also in Control Language code.

The static specification step comes from methodological considerations. This step is based on the description language (developed with Keops).

From algebraic specification language technics, the DL point out that the static description of a system should be a cognitive model rather than an implementation model. With DL, we describe a system in such a sense that their properties should be precisely established, independently of implementation choice. With DL, a problem will be described by independent modules. This description be unlinear and incomplete. It is used to create modules independently of their creation order, to modify their hierarchical organization, and to reuse them to define other modules.

So DL is not a real algebraic specification language but a tool to help to the definition of modules. This language makes easier the modular and incomplete conception of a problem.

A DL module is defined by 6 parts:

Module : (module name)
    Properties : (list of properties and their signature)
    Axioms : (declaration list)
        (equation list)
    Links : (list of link type and linked modules)
    Access rights : (list of modules which have
        a visibility right)
    Comments : (character string)
End of module

Now we can see in detail the basic entity of DL.

- The **heading** is used to know the new data set that we want to introduce.

- The **list of properties** and their signature, which represent the only tools authorized to create or use the elements of the set. So they must cover all the actions that the programmer wants to execute on one of these data.
  The signature corresponds to the definition domain of the property.

- The **list of axioms** are used to define constraints or precisions on the module properties. In a modular way, the axioms must support only the module properties.

These axioms and properties came directly from the abstract data types definition.

- The **list of hierarchical links** between modules. These links are used to reuse or structure the module already specified by DL.
  The links are defined by a type and a module list. Those links are predefined or can be defined by the user.
  The predefined links express relations as: specialization/generalization or is_split_into/is_a_part_of.
  We can find in [9] or in [10], a real expert system in the field of nutrition designed with an OOL, and using these differents kinds of link.

  At an execution level, these different relations will be translated by various mechanisms of information search in the graph defined by those links.

- The **list of access right** are used to give visibility to the specified module the modules given in the list. But at first, the modules are independent and considered as "black boxes".

To help understanding, see the entire static specification of the modules *Pair_of_vehicles* and *Vehicle*.
The first part (heading and properties) of the module definition comes directly from the diagram of the abstract data type defined during the analysis step. See figures 2 and 3.

STATIC SPECIFICATION OF THE MODULE "VEHICLE"

```
Module : Vehicle
Property :
    Car_1 : → Vehicle
    Overtake_median_line : Vehicle → Boolean
    Maneuvering_legally : Vehicle → Boolean
    His_responsibility : Vehicle → Responsibility
    Have_priority : Vehicle → Boolean
    ...
Axiom :
    V : Vehicle
    Legal_maneuverings (V) : IF (Have_priority V) THEN ι
Link :
Access_right :
    Pair_of_vehicles Comment :
    "This is the static specification of the module :
    Vehicle".
```

STATIC SPECIFICATION OF THE MODULE "PAIR_OF_VEHICLES"

```
Module : Pair_of_vehicles
Property :
    One_pair : → Pair_of_vehicles
    Vehicle_1 : Pair_of_vehicles → Vehicle
    2_in_legal_maneuverings : Pair_of_vehicles
                              → Pair_of_vehicles x Boolean
    If_2_trespass_median_line : Pair_of_vehicles
                                → Pair_of_vehicles
    ...
Axiom :
    C : Pair_of_vehicles
    Vehicle_1 (C) : (not (eq () (Vehicle_1 C)))
    1_only_in_illegal_maneuverings (C) :
            (not (O= (1_only_in_illegal_maneuverings C)
                    (2_in_illegal_maneuverings C)))
    2_in_legal_maneuverings (C) :
            IF (eq (2_in_legal_maneuverings C) t)
            THEN (print "no infraction of the highway code")
Link :
Access_right :
Comment :
    "This is the static specification of the module :
    Pair_of_vehicle".
```

On the module *Vehicle*, an axiom which marks a constraint on the property *legal_maneuvering* is defined, and points out that if a vehicle *is_prioriter*, then it is in *legal maneuvering*.
For our subproblem, there is no link here. But an access right on the *Pair_of_vehicles* means that all pairs of vehicles can have a visibility on the properties of their cars.

On the module *Pair_of_vehicles*, we can point out that there are two kinds of axiom. Some axioms expressing **conditions** that specify deductions linked to a test, as the axioms relating to the property *2_in_legal_maneuverings*.
This axiom expresses that if two cars of a pair of vehicles are in legal maneuverings, then we can deduce that there is no infraction of the highway code.
We have also axioms expressing **affirmations** on properties without conditions, as the axioms relating to the cars of a pair of vehicles.
This axiom affirms that a pair of vehicles has two known cars.

We think that these two kinds of axiom are sufficient, in practice, to specify our modules.

As we use this tool progressively, we obtain a library of modules which can be reused to define the static specification of a new problem.

Now, as shown in figure 1, we can do a first automatic generation step.

The code generation in Keops from the DL modules defines some type checking on the various entities of Keops. See section 6.
Now we can see the dynamic specification step (before the automatic generation) because the first generation of code gives Keops code but also CL code. So we need to know the CL.

5 Third step of the method: Dynamic Specification

Contrary to DL, which is situated at a static division level of a problem, CL is mainly concerned with dynamic aspects (controls during the execution) of the program to define. It is through this event control language that we design, from the specification step, controls on the executing environment of the program, or specify some error cachings, or control event appearances (for consistency controls for instance) in the environment.

The description of the next system is done by the search of dependent events throught a temporal control or by the search of events which are executed in a specific environment, or simply by the definition of events which are launched by the appearance of other events. CL can also be used to specify traces, essential for following program execution.

CL is a language which is used, from the specification step, to define controls known from the reading of the original problem description. It is not necessary to wait until the implementation step to define controls.

Those controls will be generated in KEOPS code during the second automatic generation step (see figure 1), and the tool is used to be able to remove them at any moment from the KEOPS program code.

The second automatic generation of code in the OOL KEOPS start when the dynamic specifications are completed. The implementation of controls is really boring, so it is interesting to be able to specify them from the beginning, in a more attainable language, and take advantage of an automatic generation of code.

CL can be used without CL modules yet defined in the environment, or after the first automatic generation of code. Because the first automatic generation of code produces modules of CL from the axioms defined in DL modules, it is possible to take those CL modules to modify them, to complete them ...

Each control specification is linked to a CL module by a KEOPS entity on which the control is concerned. For instance, if the control concerns a property access, the associated module will be named by the corresponding KEOPS entity.

The next subsection will show the execution work of a CL module, the definition of a CL module, and some examples of CL modules from our insurance problem.

5.1 Execution work and definition of a CL module

The reading of a CL module should be as follows, and be understood in the same way as for the automatic generation in KEOPS code, and approximately in the same way as during the execution program.

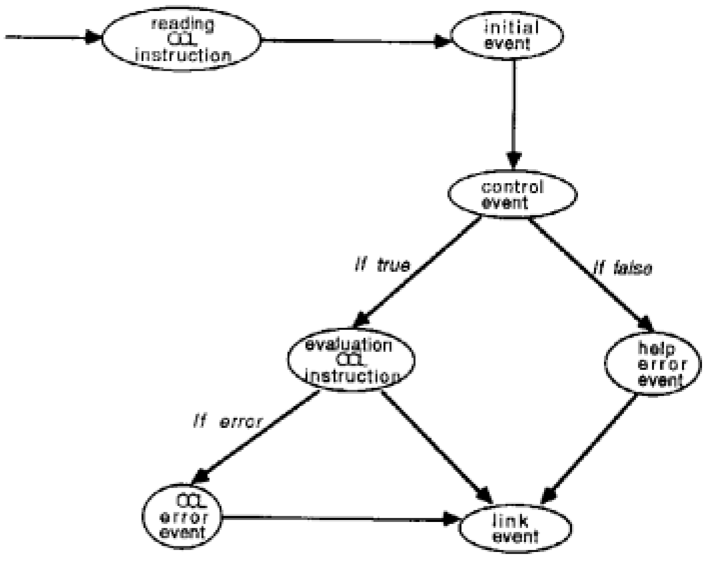


Figure 4: Execution work of a CL module

Each time the KEOPS instruction defining a CL module appears in the system, the events defining this module will be activated in the following way: (Look at figure 4 at the same time)

- **The initial event** is activated, in which there are declarations of variables found in other events. Values are substituted for the variables.

- **The control event**, and the control(s) are checked:
  - If the *control is valid*, then the *KEOPS instruction* is normally evaluated. If a **KEOPS error** appear, it is possible to stop it and to replace it by the **KEOPS error event**, if this is specified.
    There is an evaluation of the instruction defined in the **link event**, which is used to realise a treatment, for instance according to the return value of the instruction, before retaking in order, the instructions that form the program.

  - If the *control is not valid*, then there is an evaluation of the instruction defined in the **help error event**, which can be, for instance, an error message.
    Then, there is the evaluation of the instruction defined in the **link event** as before.

We can see this process in detail through our examples.

The basic entity of CL is the **module**, which is named by an **instruction** and defined by a **control bloc**. Each of these modules of control is associated with a description module, and defines one or several controls on the environment as soon as the corresponding instruction appears in the system.

A CL module is mainly defined in two parts: a heading and a control bloc.

Instruction associated :
    (instruction)
    Initial event : (list of variable declarations)
    Control event : (list of controls in Keops)
    Help error event : (list of Keops instructions to execute)
    Keops error event : (Keops instruction)
    Link event : (Keops instruction)
End of module

Now we can see in detail the basic entity of CL.

- The **heading** is defined by the definition of the instruction linked to the module. It is defined as a filter during the program execution.

  *For instance, we define a CL module relating to the property "height" of the module "symbol_stack" by: "Instruction associated to the module : (height symbol_stack_Object? *value*). In that sense, we specify that controls will be made on the property "height" as soon as a modification is done on the property "height" from any object of type "symbol_stack".*

  There are two instructions with which to define a CL module: the message sendings and the access in writing on the properties of an object.

- The **control bloc** is defined by five events: initial, control, help error, Keops error, and link, which have a fixed linking order. Each event is associated to a specific treatment which will be executed in such or such a situation. Those treatments can be empty, incomplete or easily modifiable.

  - The **initial event** is used to define some Lisp variables, local to the control bloc. The constants will be replaced by their value during the second step of the Keops code generation. A constant declaration is :
    $$constant\_name = value$$

  - The **control event**. Whose associated treatment is used to make consistent tests between the instruction linked to this bloc and the environment. It is possible to define several controls, each of them can be associated to a help error event.
    The instruction specifying the control is defined in Keops (see the examples). The definition of the syntax can be found in [5].

  - The **help error event**. It is possible to define, for each control, an instruction linked to the help event. The help event instructions are evaluated only if their corresponding control is false.
    The instruction defining the help event is defined in Keops.

– The **Keops error event**. Only one instruction can be linked to this event. It is defined also in Keops.

– The **link event**. Only one instruction can be defined in this event, and in Keops. If it is defined, this instruction is automatically launched after the evaluation of the instruction linked to the bloc, or after the evaluation of the instruction defined in the Keops error event, or after the evaluation of the last instruction defined in the help error event.

All the instructions in the control bloc are automatically generated in Keops through the axioms of a DL module, during the first generation step from DL modules in CL.
The instruction linked to the control bloc is evaluated or not as soon as changes occur in the environment.

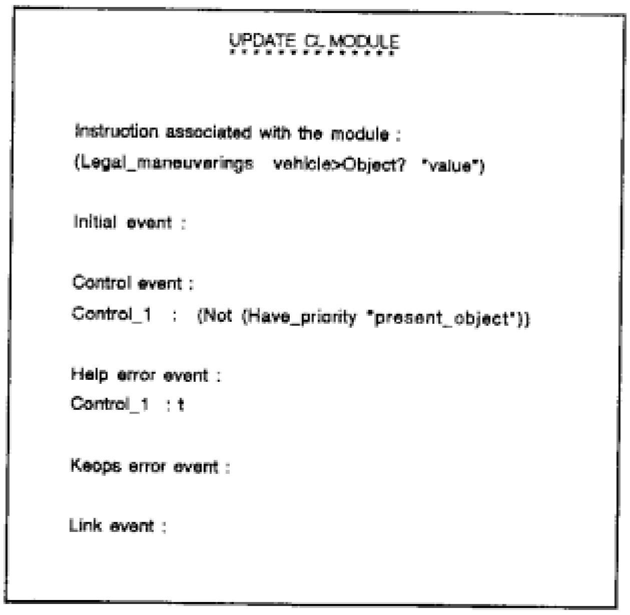
We can follow the explanation with our example given in the next subsection. The given CL modules came straightly from the first step generation. Nothing is added.

5.2 Dynamic specification of our problem

5.2.1 Module from the generation of the DL module "Vehicle"

1 - CL module generated from the axiom (See figure 5) :

Legal_maneuvering (V) : IF (have_priority V) THEN t



```
UPDATE CL MODULE

Instruction associated with the module :
(Legal_maneuverings   vehicle>Object?   "value")

Initial event :

Control event :
Control_1   :   (Not (Have_priority "present_object"))

Help error event :
Control_1   : t

Keops error event :

Link event :
```

Figure 5: CL module from DL module "Vehicle"

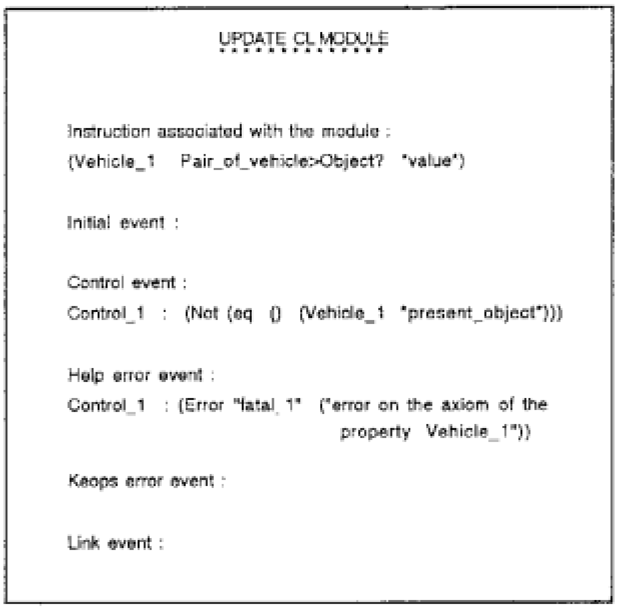
This axiom express that when a writting access is done on the property "Legal_maneuvering", the control verifie whether the present vehicle has priority. If yes, the writing value on the property "legal_maneuvering" is systematically "true".

In the control event, there is a logic inversion of the axiom test part. If the control event is false, the help error event is evaluated. So there is logical coherence between the axiom and the generated CL module.

5.2.2 Modules from the generation of the DL module "Pair_of_vehicles"

1 - CL module generated from the axiom (See figure 6) :

Vehicle_1 (C) : (not (eq () (Vehicle_1 C)))



```
UPDATE CL MODULE

Instruction associated with the module :
(Vehicle_1    Pair_of_vehicle>Object?   "value")

Initial event :

Control event :
Control_1   :   (Not (eq   ()   (Vehicle_1   "present_object")))

Help error event :
Control_1   : (Error "fatal_1"   ("error on the axiom of the
                                    property  Vehicle_1"))

Keops error event :

Link event :
```

Figure 6: CL module from DL module "Pair_of_vehicles"

The instruction is a writting on the property. The control is to verify that the value of the property "vehicle_1" of the present object is not "NIL".
The CL module generated from the axiom on the property "Vehicle_2" has the same shape.

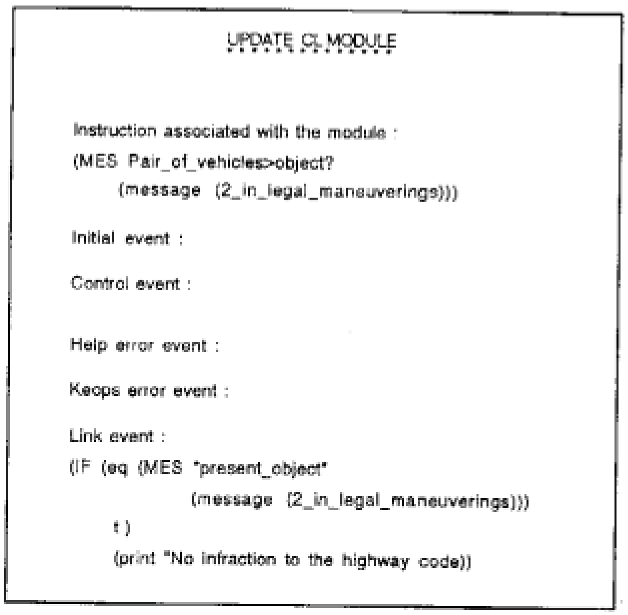
2 - CL module generated from the axiom (See figure 7) :

2_in_ legal_maneuverings (C) :
IF (eq (2_in_legal_maneuverings C) t)
THEN (print "no infraction to the highway code")

Here, the axiom is associated with the functioning of a pair of vehicles. So, from this condition axiom, the generation will built a CL module on a method. The axiom condition part is on the result of the method, then the control must not be done before the evaluation of the method, so, the control should not be specified in the control event but in the link event. At this time, the control will be done after the evaluation of the method.

The axioms on the properties *1_only_in_illegal_ maneuverings* and *2_in_illegal_maneuvering* are represented as a method by the automatic code generation in Keops.
The two axioms express affirmations (not conditions), and as we will see in the next section on the automatic generation of code, no CL module will be generated.

The presentation of all the control modules is finished. We now have the dynamic specification of our problem. (Those modules came from the axioms specified during the static specification and were automatically generated)
Of course it is possible to retake those modules and complete them, or to define some other way to refine the



```
UPDATE CL MODULE

Instruction associated with the module :
(MES Pair_of_vehicles>object?
      (message  (2_in_legal_maneuverings)))

Initial  event  :

Control event :

Help error event :

Keops error event :

Link event :
(IF (eq (MES "present_object"
                (message  (2_in_legal_maneuverings)))
      t )
      (print "No infraction to the highway code))
```

Figure 7: CL module from DL module "Pair_of_vehicles"

controls on the problem environment.

The CL is designed in KEOPS and we can use it through a convenient interface. A CL module library can be constituted as soon as the use of the tool. The modules can be reused and completed to define dynamic specifications of other problems.

Now we can do the second step of the automatic generation, which is done from the CL module in KEOPS. In the tool, there are five possible generation types of CL module that can be derive from the axioms.

As figure 1 shows, after the first step of automatic generation, which gives code in KEOPS and also in CL, we can use this CL. Perhaps, for this problem, we don't need to complete the obtained CL modules, as we did in our example. In that case, the third step proposed by the methodological guide is used only to ask the second automatic generation of code to obtain the entire architecture of our program in the KEOPS OOL.

The next section shows the two steps of the automatic generation of code.

6 Fourth step of the method: Automatic Generation of Code

The research aim in the design of this methodological guide is to bring to OOL a methodological complement which facilites the software design. The methodological guide makes it possible to define static and dynamic specifications, and also **to automatically generate KEOPS code directly through those specifications**. One point concerning this automatic generation of code is that the **obtained program will strictly satisfy the specifications**, because it is deduced from them.

Looking again at figure 1, we remark two steps of the automatic generation of code:

- After we completely define the static specification of our problem, with DL modules, the first generation step will give some code in KEOPS and some code in CL.

- After we retake or update the CL modules defined in the first generation step, and create other CL modules, the second generation step gives KEOPS code corresponding to the "architecture" of our program. That means it gives the final program with all declarations of the needed KEOPS entities and controls to evaluate. Of course the procedural attachment and the method body, corresponding to "How" to execute the program should be written by the programmer.

6.1 The first automatic generation of code

The first step of the automatic generation of code relies on a set of description modules (defined with DL) describing the static specifications of the problem. It is necessary to have well-formed modules, syntactically and semantically, both individually and together. The order in which the modules are used for the generation is not important.

This first generation step gives a part of code in KEOPS and another part in CL. The two parts are presented independently.

6.1.1 Code generation in KEOPS from DL modules

This first part needs to take into account the identity of the module, its properties, its links and its access rights. To each generated module there corresponds a KEOPS file, named by the module name and the extension ".ll".
The program code is split up into as many files as the description module needs for the solution space. In each file, the definition of the corresponding description module is written.

Before defined the correspondence between the entities of the DL and those of the KEOPS OOL, we give the KEOPS file defined from the module *Pair_of_vehicles*.
This file named *pair_of_vehicle.ll* is explained immediately after the description of the code generation.

```
;*****************************************************
;
;
;              FILE     pair'of'vehicles.ll
;
;*****************************************************
;

;Specification module
;
;Module : pair-of-vehicles
;Property :

 ......     Definition of the DL modules in comment   ........


;End of writing the specification module


;Class declaration
;****************
;

(CLASS pair'of'vehicles
       (SURCLASS root)
       vehicle'1
       vehicle'2
);End of the class declaration


;Instance declaration
;*******************
;

(OBJECT one-pair pair'of'vehicles)


;Declaration of a procedural attachment in reading
```

```
;*******************************************
;for the property :  vehicle'1
;**************

(PROC'AT (PROP vehicle'1)
        (CLASS pair'of'vehicles)
        (TYPE reading)
        (BODY VE ()
           ;Control of transmitter
           (ifn (member *origin* (instances pair'of'vehicles))
                (error "fatal'1" ("the transmitter type"
                                  *origin*
                                  is not correct")
                )
           );end of the transmitter control

           ; WRITING OF THE PROC ATT BODY

        );End of the body
);End of the proc att


;Declaration of the procedural attachment at the beginning of writing
;*****************************************************************
;for the property : vehicle'1
;**************

(PROC'AT (PROP vehicle'1)
        (CLASS pair'of'vehicles)
        (TYPE writingbeg)
        (BODY VE (*value*)
           ;Control of transmitter
           (ifn (member *origin* (instances pair'of'vehicles))
                (error "fatal'1" ("the transmitter type"
                                  *origin*
                                  "is not correct")
                )
           );end of the transmitter control
           ;Control on the writing value
           (ifn (member *value* (instances vehicle))
                (error "fatal'1" ("the parameter type"
                                  *value*
                                  "of the property"
                                  "vehicle'1"
                                  "of the object"
                                  *object*
                                  "is not correct")
                )
           );End of the writing value control

           ; WRITING OF THE PROC ATT BODY

        );End of the body
);End of the proc att


;Declaration of the procedural attachment in reading
;***********************************************
;for the property : vehicle'2
;**************

 .......   Same definition as the procedural attachments  .........
 .......   associated to the property "Vehicle'1"         .........


;Declaration of the procedural attachment at the  beginning of writing
;*****************************************************************
;for the property : vehicle'2
;**************

 .......   Same definition as the procedural attachments  .........
 .......   associated to the property "Vehicle'1"         .........


;Declaration of the method : 2'in'legal'maneuvering
;***********************

(METHOD 2'in'legal'maneuvering
   (CLASS pair'of vehicles)
   (BODY VE ()
      ;Control of the transmitter
      (ifn (member *origin* (instances pair'of'vehicles))
           (error "fatal'1" ("the transmitter type"
                             *origin*
                             "is not correct")
           )
      );end of the transmitter control
      (let ((res
          (progn

            ;WRITING OF THE METHOD BODY

           );End of progn
         ));End of the calcul of the result
         ;Control of the out type
         (ifn (MES treathment`specif`error-initial`object
                 (message  (primitive`types`?  res))
             )
             (error "fatal`1"  ("The type of the return value of
                                the method is not correct")
             )
         );End of the out control
     );End of let
  );End of body
);End of the method


;Declaration of the method : 1`only`in`illegal`maneuvering
;***********************

(METHOD 2`in`legal`maneuvering
  (CLASS pair`of`vehicles)
  (BODY VE ()
     ;Control of the transmitter
     (ifn (member *origin* (instances pair`of`vehicles))
         (error "fatal`1" ("the transmitter type"
                           *origin*
                           "is not correct")
         )
     );end of the transmitter control
     (let ((res
           (progn

            ;WRITING OF THE METHOD BODY

           );End of progn
         ));End of the calcul of the result
         ;Control of the out type
         (ifn (member res (instances  pair`of`vehicles))
             (error "fatal`1"  ("The type of the return value of
                                the method is not correct")
             )
         );End of the out control
     );End of let
  );End of body
);End of the method


;Declaration of the method : 2`in`illegal`maneuvering
;***********************

  ......   Same definition as the method     ......
  ......  "1`only`in`illegal`maneuvering"  ......
```

The correspondence between the entities of the DL (module, property, link and access right) and those of the KEOPS OOL is the following:

- Each **declaration of a module** is associated to a *class declaration* in Keops. Effectively, the class is considered as a model for a set of objects.

- The **properties of a module** can be defined differently according to their arity and coarity. To each of these properties a declaration of KEOPS entity is associated:

  * *If the arity has no sets and the coarity has one set which is the module itself* module_name === (property_name : → module_name) === The property specifies that an element exists and is named with the name of the property in the set defined by the module. In KEOPS, this corresponds to a declaration of a class instance.
    In our example, see the declaration of the object *one_pair* instance of the class *pair_of_vehicles*.

  * *If the arity has one set which is the module itself* module_name *or the set of the modules* {module_name} *and the coarity has one set different from the one represented by the module*

=== (property_name : module_name → set1 or property_name : {module_name} → set1)
=== The property which is defined for a module element or for the module, specifies that its value has the set type defined in the coarity, and that there is no treatment that should be define to get this value.
In KEOPS, this property definition corresponds to the declaration of a generic property (instance variable in Smalltalk) or a common property (class variable in Smalltalk).
The set specified by the coarity is used to control the type of the property value. Then, for each property, there is a declaration of a procedural attachment at the beginning of writing, with a Keops instruction which is used to control the type of the given value.

There are also some controls on the transmitter of the reading or writing access requested for the properties (see the treatment for the access rights).
In our example, see the declaration of the procedural attachment link to the property *Vehicle_1*.

* *If the arity has several sets in which one is the module itself* module_name *and the coarity has several sets in which one is the module itself* module_name *or the set of the modules* {module_name}
=== (property_name : module_name x set1 x set2 x ... → module_name x setA x setB x ... or property_name : {module_name} x set1 x set2 x ... → {module_name} x setA x set B x ...)
=== The property specifies that it must have some entry parameters with the type set1, set2, ... to determine its return value which is a list with elements of the type ensA, ensB, and so on.
In KEOPS, this corresponds to the declaration of an instance method or class method.

The sets set1, set2, ... specified in the arity are used to make type control on the parameter value of the method, and setA, setB, ... specified in the coarity are used to control the output value of the method.
In the method body, a Keops instruction is defined to control the type of the parameters and another controls the return value of the method.
Some controls are done on the message transmitter, see the access rights. In our example, see the declarations of the methods named *2_in_legal_maneuverings* and *1_only_in_illegal_maneuverings*.

* If one of the controls is not valid, the system will send a KEOPS fatal error which stops the program execution.

– The **links of a module** correspond to the definition of a sur_class link between the present class and the classes corresponding to the modules specified by this link.
If a new type of link is introduced, its method of information search in the class graph should be redefined. Actually, in the tool, the programmer should define in the method body the moment at which the system must use such and such a type of link.

– The **access rights of a module** are used to make visible this module for the specify modules. The class corresponding to the present module is no longer a black box, but is considered to be a glass box for some classes.
In KEOPS, control instructions are defined on potential transmitters in all procedural attachments and methods yet defined. A procedural attachment is defined in reading for each property, and is used to control the potential transmitters.

Now, there remain the axioms which are used to generate CL code.

6.1.2 Code generation in CL from DL modules

This second part of the first generation step needs to take into account the axioms of a module. To each axiom of a module there corresponds a CL module, referenced by the declaration of a KEOPS instruction defined from the property of the DL module controlled by this axiom.

It should be useful to follow, at the same time, the CL modules of our problem (figures 5, 6, 7) and the presentation of the correspondence between the entities of the description language and those of the control language.

The axioms of a module are used to define constraints or controls on the properties of this module and they can be specified as affirmation or condition.

Several KEOPS entities can be generated from the axioms because we have :

- Element declarations
- Affirmation axioms
- Condition axioms

and also because the properties of a DL module on which the axioms are associated can be generated in KEOPS as :

- Generic or common properties
- Instance or class methods.

Whatever the case, the code generation phase of the DL modules in CL modules will not define any initial event and any Keops error event for evident reasons. See the definition of a control block.

The correspondence between the various axioms of a DL module and the generated CL modules are the following.

- The **declarations** are used to know the types of the several objects used to define the axioms. So the objects are associated to their type in the definition of the CL modules.
- The **affirmation axioms** are generated differently:
  - * If the DL module property on which the axiom is associated is used to generate a generic or common property in KEOPS, then **this axiom expresses a constraint on the writing value of this property**.

    The generation gives:
    - · *A CL module identified by the writing instruction on this KEOPS property* for any object of class corresponding to the present DL module.
    - · *A control instruction in the control event which is used to verify the constraint expressed by the axiom.*
    - · *An instruction in the help error event which will send a KEOPS fatal error if the control is false.*
  - * If the DL module property on which the axiom is associated is used to generate a class or instance method in KEOPS, this axiom expresses **an affirmation on the composition law of the DL module properties** which is identified to *an invariant in the program*, and then *doesn't need any control during the execution of this one.*
    The generation doesn't give any CL module in this case.
- The **condition axioms** are differently generated :
  - * If the DL module on which the axiom is associated is used to generate a generic or common property in KEOPS, then this **axiom expresses a condition for instance, on the writing value of this property.**

    The generation gives:
    - · *A CL module identified by the writing instruction on this KEOPS property* for any object of the class corresponding to the present DL module.
    - · *A control instruction in the control event which is used to verify the control expressed by the axiom.*

- *An instruction in the help event which is the same as the one defined in the "THEN" part of the axiom.*

* If the DL module property on which the axiom is associated is used to generate a class or instance method in KEOPS, this axiom expresses a **condition on the method result or on the parameters or...**

  If the axiom condition is on something other than the method result, this condition should be evaluated before the method body, to stop the method execution if the axiom test is right and replace it with the evaluation of the "THEN" part of this axiom.

  The generation gives:

  - *A CL module identified by the KEOPS message sending instruction.* This message corresponds to the property with which the axiom is associated. The message sending can be evaluated on all the objects of the class corresponding to the present DL module.
  - *A control instruction in the control event which is used to verify the control.*
  - *An instruction in the help event which is the same as the one defined in the "THEN" part of the axiom.*

  If the axiom condition is on the method result, this condition should be evaluated after the method body to obtain its return value.

  The generation gives:

  - *A CL module identified by the KEOPS message sending instruction.* This message corresponds to the property on which the axiom is associated. The message sending can be evaluated on all the objects of the class corresponding to the present DL module.
    Because the control is on the result of the method, it cannot be done during the control event evaluation (See the execution mode of a CL module). This control should be defined in the link event, which will be evaluated after the instruction evaluation.
  - *An instruction in the link event which corresponds to the complete axiom.*

At the end of this automatic generation of code from the description modules, we obtain some KEOPS files and a list of CL modules. The second code generation step is done from the CL modules to complete the KEOPS code defined in the file previously given.

Now we can see how the generation is done from the CL modules, how to pass from the events to the KEOPS instructions, and how to locate these instructions in the previous generated KEOPS code.

6.2 The second automatic generation of code

The second step of the automatic generation of code relies on a set of control modules (defined with CL) describing the dynamic specifications of the problem. It is necessary to have well-formed modules, syntactically and semantically. The order in which the modules are used for the generation is not important.

This second step gives the final code corresponding to the complete use of the methodological guide.

For this generation, it is necessary to take into account all the entities defined in a CL module, and the generated KEOPS code, in order to insert the additional controls brought by those modules in the existing KEOPS code.

Before we define the correspondence between the entities of the CL and those of the KEOPS OOL, we give the final KEOPS file defined from the module *Pair_of_vehicles*.
This file is explained immediately after the description of the code generation, but it is interesting to compare this file to the one given by the first generation step. We can remark the added controls and their location in the method and procedural attachment bodies, according to the event defined in a CL module.

```
;****************************************************
;
;              FILE     pair'of'vehicles.ll
;
;****************************************************

;Specification module
;
;Module : pair-of-vehicles
;Property :


 ......     Definition of the DL modules in comment   ........



;End of writing the specification module


;Class declaration
;****************

(CLASS pair'of'vehicles
      (SURCLASS root)
      vehicle'1
      vehicle'2
);End of the class declaration


;Instance declaration
;*******************

(OBJECT one-pair pair'of'vehicles)


;Declaration of a procedural attachment in reading
;*********************************************
;
;for the property :  vehicle'1
;**************

(PROC'AT (PROP vehicle'1)
        (CLASS pair'of'vehicles)
        (TYPE reading)
        (BODY VE ()
          ;Control of transmitter
          (ifn (member *origin* (instances pair'of'vehicles))
               (error "fatal'1" ("the transmitter type"
                                 *origin*
                                 is not correct")
               )
          );end of the transmitter control

          ; WRITING OF THE PROC ATT BODY

        );End of the body
);End of the proc att


;Declaration of the procedural attachment at the beginning of writing
;*****************************************************************
;
;for the property : vehicle'1
;**************

(PROC'AT (PROP vehicle'1)
        (CLASS pair'of'vehicles)
        (TYPE writingbeg)
        (BODY VE (*value*)
          ;Control of transmitter
          (ifn (member *origin* (instances pair'of'vehicles))
               (error "fatal'1" ("the transmitter type"
                                 *origin*
                                 "is not correct")
               )
          );end of the transmitter control
          ;Control of the writing value
          (ifn (member *value* (instances vehicule))
               (error "fatal'1" ("the parameter type"
                                 *value*
                                 "of the property"
                                 "vehicle'1"
                                 "of the object"
                                 *object*
                                 "is not correct")
               )
          );End of the writing value control
          (TAG entry'axiom'control
               (ifn (not (eq () *value*))
                    (exit out'axiom'control
                    (Error "fatal'1"
                           ("Error on the axiom of the"
                            "property Vehicle'1")
                    )
                );End of exit
            );End of entry'axiom'control

            ; WRITING OF THE PROC ATT BODY

        );End of tag
     );End of the body
);End of the proc att


;Declaration of the procedural attachment in reading
;***********************************************
;for the property : vehicle'2
;**************

  .......  Same definition as the procedural attachments  .........
  .......  associated to the property "Vehicle'1"        .........


;Declaration of the procedural attachment at the beginning of writing
;**************************************************************
;for the property : vehicle'2
;**************

  .......  Same definition as the procedural attachments  .........
  .......  associated to the property "Vehicle'1"        .........


;Declaration of the method : 2'in'legal'maneuvering
;************************

(METHOD 2'in'legal'maneuvering
   (CLASS pair'of'vehicles)
   (BODY VE ()
      ;Control of the transmitter
      (ifn (member *origin* (instances pair'of'vehicles))
           (error "fatal'1" ("the transmitter type"
                             *origin*
                             "is not correct")
           )
       );end of the transmitter control
       (let ((res
              (progn

                ;WRITING OF THE METHOD BODY

              );End of progn
            ));End of the calcul of the result
            ;Control of the out axiom
            (if (eq res t)
                (print " No infraction of the highway code")
            );End of if
            ;Control of the out type
            (ifn (MES treatment'specif'error--initial'object
                      (message  (primitive'types'?  res))
                 )
                 (error "fatal'1"  ("The type of the return value of
                                     the method is not correct")
                 )
            );End of the out control
       );End of let
   );End of body
);End of the method


;Declaration of the method : 1'only'in'illegal'maneuvering
;************************

(METHOD 1'only'in'illegal'maneuvering
   (CLASS pair'of'vehicles)
   (BODY VE ()
      ;Control of the transmitter
      (ifn (member *origin* (instances pair'of'vehicles))
           (error "fatal'1" ("the transmitter type"
                             *origin*
                             "is not correct")
           )
       );end of the transmitter control
       (let ((res
              (progn

                ;WRITING OF THE METHOD BODY

              );End of progn
```

```
          ));End of the calcul of the result
          ;Control of the out type
          (ifn (member res (instances  pair'of'vehicles))
               (error "fatal'1"  ("The type of the return value of
                                  the method is not correct")
               )
          );End of the out control
      );End of let
   );End of body
);End of the method


;Declaration of the method : 2'in'illegal'maneuvering
;************************

  ......   Same definition as the method     ......
  ......  "1'only'in'illegal'maneuvering"   ......
```

The correspondence between the entities of the CL and those of the KEOPS OOL is the following:

- From the **instruction identifying the module** we know on which entity the control are made. The generation doesn't give any KEOPS code.
- The **initial event** is used to replace the constants with their value before starting the generation. Any KEOPS code is given.
- The **control event** and the **help error event**.
  To each instruction defined in the event control is associated an instruction in the help error event, then we present those two events together.

  There are some differences in the type of the instruction associated to the CL module which divide the generation into three parts :

  - If the **CL module instruction expresses a writing on a KEOPS property from a condition axiom**, then the generation produces some code in its procedural attachment at the beginning of writing.
    This generated code will have the following shape:

```
(TAG  entry_axiom_control
      (IF  (NOT  ''control event'')
           (EXIT  entry_axiom_control
                  (SETQ  *value*
                         ''corresponding help error event''
                  )
           );End of exit
      );End of entry_axiom_control

      ; WRITING OF THE PROC ATT BODY

);End of tag
```

  - If the **CL module instruction expresses a writing on a KEOPS property from an affirmation axiom**, then the generation produces some code in its procedural attachment at the beginning of writing.
    This generation produces the same code as before.
  - If the **CL module instruction expresses a message sending**, the generation produces some code in the method called by this message sending.
    The generated code is the following:

```
(TAG  entry_axiom_control
      (IF  (NOT  ''control event'')
           (EXIT  entry_axiom_control
                  (SETQ  *value*
                         ''corresponding help error event''
                  )
           );End of exit
      );End of entry_axiom_control

      (PROGN

      ; WRITING OF THE METHOD BODY

      );End of progn

);End of tag
```

Because several control definitions in the control event and of course in the help error event can exist, the automatic generation of code will successively introduce the corresponding KEOPS code where it is necessary.

- The **KEOPS error event**.

  If an instruction is defined in this event, a *definition of a new error method corresponding to this instruction is done in the KEOPS system*. Effectively, the calls to the fatal error methods in KEOPS are modifiables during the problem execution.
  The generation will define a new method in the system, and the mechanims necessary to activate this method when it is required according to the CL module definition.

- The **link event**.

  The generation produces some code in the procedural attachment at the beginning of writing if the instruction associated to the module expresses an access on a KEOPS property, or some code in the method, if the instruction associated with the module expresses a KEOPS message sending.
  The generated code is located after the comment on the writing of the procedural attachment body, or after the end of the first argument of the "LET" which is used to keep the method result.
  The code is the control instruction corresponding to the complete instruction defined in the link event.

After this second automatic generation of code, we can point out that all the controls introduced by the generation are located in pre or post condition on the methods and on the procedural attachments. It is important to remember that all the controls can be deleted, by simple request, at any moment of the problem execution.

Now we obtain some KEOPS files that correspond to the solution of our problem and that are automatically generated from the static and the dynamic specifications.

The method and the procedural attachment bodies should be completed in KEOPS in writing the code corresponding to their functioning at the place where the generation wrote in comments "procedural attachement body" and "method body".

Note that the file loading depends only on the class hierarchy.

7 Conclusion

This report describes a progressive method and a guideline for the design of object oriented programs. The tool is qualified to be a methodological guide because it operates during a large part of the software life cycle: for the analysis, specifications, conception, and programming.

- The analysis step is not automatic. The treatments performed on the original description of a problem, to define the concept needed for the specification step, essentially depend on the context in which the problem evolves, on the knowledge which the reader has about the real world for this domain, and on what the reader understands.
  The automation is really difficult to realize, and was not my first research topic. This analysis is easy to apply to render any kind of problems as abstractions; it is not specific to only OOL.

- The specification step is in two parts: the static and the dynamic specification phases. The static phase is used to quickly obtain a global view of a problem, directly represented as modules. The dynamic phase is used to define controls or constraints which come directly from the original problem descriptions.
  The specification order is first the static one and then the dynamic. A go and come back can be done, but not with any effect on the generated code. The automatic generation of code should be redone for the modified modules.
  The interesting thing about those specifications is that the customer, the user, and the programmer can easily understand the problem during the computerization process.

- The conception step represents here the change from the specifications to the program architecture. This step is performed by the automatic generation of code in KEOPS OOL. This generation is used to get programs well written, with comments, and well structured in files. The resulting programs automatically satisfied the problem specifications because the code generation is made directly from them.

- The programming step must begin from the file containing the code corresponding to the program architecture and must define the actions in the same KEOPS language. In other words, it defines the "how to do", because the "what to do" is defined through the specifications.

The prototype of the guide shows the interest and the feasibility of a such approach. It has been realised with our object oriented conception method.

Our first objective was to define a general object oriented conception method satisfying any kind of OOL and adapted to any kind of problem.
The result is represented here as a methodological guide, organised in several help phases for the analysis, for the static and dynamic specifications, and for an automatic partial implementation, which is based on a common kernel of the concepts of various kind of OOL.

From a critic point of view:

- The side "help for the analysis" has been added to complete the object oriented conception method and obtain a methodological guide which covers all the steps of software development.
  Actually, something is missing from this analysis method: we would like to define the hierarchical links between the abstractions directly from the original description of the problem. Now, those links can be defined only during the static specification step but we think that they can be found from sentences in natural language.
- The prototype has been implemented with the KEOPS OOL, and the automatic generation of code is also in KEOPS language. Of course this generation can be easily done in any other OOL.
- The fact that the method is based on the commom concepts of OOL leads us to make two remarks :
  - We can't enjoy, for the program conception, the power of concepts specific to various OOL, such as several type of message sendings, the frames, the production rules ...

- This guide has been realised with KEOPS, but we took only the basic concepts of OOL and the procedural attachments for its implementation, not the other specificities of KEOPS.
  So the methodological guide can be easily translated into any other OOL.

The change from the prototype to a real tool has to be made to allow for a correct and industrial use of this methodological guide. It should be used to rapidly define prototypes in KEOPS. Then the program can be operational in KEOPS or, for better efficiency, translated into OBJECTIVE-C [11] which is an OOL based on C, by a cross-compiler LeLisp in C.

References

[1] E. Abbott. Program design by informal english description. In *Communications of the ACM*, 1983.

[2] G. Booch. *Software components with ADA*. The Benjamin Cummings Publishing Company, 1987.

[3] J.M. Goetz. *LRO2 = Langage de programmation et de représentation des connaissances par objet.* CRIL FRANCE, 1986.

[4] J.V. Guttag, D.R. Musser, and E. Horowitz. Abstract data types and software validation. In *Communication of the ACM*, 1978.

[5] A. Romanczuk. *Vers un guide méthodologique pour la conception de programmes orientés objet.* PhD thesis, Université de Lille - FRANCE, Laboratoire d'informatique fondamentale de Lille - Université des sciences et techniques de Lille, November 1989.

[6] A. Romanczuk. Expert system design with an object oriented analysis method. submited to the seventh IEEE conference on Artificial Intelligence Applications - Miami - February, 1991.

[7] A. Romanczuk and G. Comyn. An implemented guideline for developing and generating object programs. Technical Report IT 124, University of LILLE - FRANCE, 1987.

[8] A. Romanczuk and G. Comyn. Présentation par l'exemple d'un guide méthodologique pour la programmation orientée objet. Technical Report IT 171, University of LILLE - FRANCE, 1989.

[9] A. Romanczuk, G. Comyn, and R. Beuscart. Use of an object oriented language keops for the realization of an expertise in the field of nutrition. In *A.I. and Cognitive Sciences - Manchester University Press*, 1987.

[10] A. Romanczuk, G. Comyn, and R. Beuscart. An expertise in the field of nutrition with object oriented language. In *IEEE Engineering in Medecine and Biology Society - NEW ORLEANS.*, 1988.

[11] STEPTONE, Connecticut. *Objective C, User reference manual*, 1986.

[12] A. Synder. Common objects: An overview. In *SIGPLAN Notices*, pages 19 – 28, October 1986.

[13] C. Vogel. *Génie Cognitif.* Masson, 1988.