A development shell for cooperative problem-solving environments

Jutta WILLAMOWSKI
INRIA Rhone-Alpes
LIFIA / IMAG
46, av. Felix Viallet
F-38031 GRENOBLE Cedex 1
tel. 33 - 76 57 45 71
Jutta.Willamowski@imag.fr

François CHEVENET
URA CNRS 243
Universite Claude Bernard - Lyon1
F-69622 VILLEURBANNE Cedex
tel. 33 - 72 44 82 38
chevenet@biomserv.univ-lyon1.fr

François JEAN-MARIE
Cap Gemini Innovation
7, Chemin de Vieux Chene
ZIRST 4206
F-38942 Meylan Cedex
tel. 33 - 76 76 47 47
jeanmarie@capsogeti.fr

Abstract

In complex domains such as scientific computing, users need support in choosing, chaining, and executing the adequate programs for problem solving. Problem solving environments are developed in order to solve automatically routine problems, decomposing them recursively in more and more elementary subproblems, and finally executing the corresponding programs. But often, the problem solving process cannot be completely automated ; the user is requested to provide missing parameter values, to solve specific subtasks, or to validate input or output data. Besides, the user must have the possibility to supervise the whole problem solving process and all the decisions made by the computer system. He must be able to intervene whenever he wants, either to modify the systems decisions, or his own choices, concerning parameter values for example. SCARP is a shell that allows to develop problem solving environments providing the necessary cooperation facilities for interactive problem solving. Its architecture, structured in different layers, is first presented ; then we discuss SLOT, a problem solving environment in exploratory data analysis, developed using SCARP.

Key-words: problem solving environment, human-computer cooperation, object-centered knowledge representation, task representation, data analysis.

1 Introduction

The concept of a cooperative problem-solving environment is proposed here, in order to assist the user in the many choices that have to be made when using computers to solve problems in scientific applications. In the following, we define a problem solving environment as a knowledge-based system whose knowledge base integrates the knowledge necessary to solve problems in an application domain: knowledge about the types of problems to be solved, knowledge about problem- solving strategies, knowledge about the available scientific computer programs, and knowledge about the entities that are manipulated in a domain. Based on this knowledge, the problem-solving environment supports the user in choosing the elementary scientific computer programs that are necessary to solve his particular problem, and in chaining them one after another in an appropriate way. Furthermore the system has to keep track of the problem solving process and to allow the user to review it easily. So far, this corresponds to the concepts presented in [Gall&91], [Rech&92].

Such a system might then solve automatically, on its own, complex problems [Rech91], [Will&93]. But subproblems might exist for which no solving strategy is known, or which can only be solved by the user, e.g. interpretation of graphics. Here, the user has to intervene into the problem solving process. Parts of the problem solving process have therefore to be attributed to the user. The definition of specific user-tasks in this respect has been proposed by [Gree&92] and [Stol91]. Besides, parameter values are needed during the problem-solving process, and to be obtained from the user. Therefore the system must be interactive. The user might not always be sure about the specific values that he has to fix. He should be able to reconsider them at any moment. Especially in complex domains it is furthermore a fact that a knowledge base is never exhaustive ; strategies might exist that are not integrated in the knowledge base, being not commonly used or not well defined [Kant88]. Therefore the system needs to be cooperative. The user must always have the possibility to escape the system's control, to modify the problem-solving strategy, or even to replace it by a completely new strategy.

Specific dialogue facilities are necessary to manage the interaction, necessary for system-user cooperation. They precise how turn-taking in the control of the problem solving process should take place [Allp89]. Furthermore, the cooperative problem-solving environment should be able to manage the different versions corresponding to the modifications made during the problem solving process. Moreover, the system has to be be open to different kinds of users, more or less competent in the application domain. Not every user has the same needs concerning the functions offered by the system. So the functioning of the system has to be easily adaptable to the user's needs [Fisc90]. SCARP[1] is a shell for developing such cooperative problem-solving environments. In SCARP, problem solving is based on successive decomposition of complex problems in more elementary subproblems. Its general architecture is presented here. Its components, the knowledge handler, the task engine, the cooperative dialogue handler, and the user interface are hereafter detailed. The last section presents SLOT, a cooperative problem-solving environment in multivariate exploratory data analysis, developed with SCARP.

2 The architecture

SCARP is structured in different layers (**figure 1**). The whole architecture is domain independent. The *knowledge base* constitutes the core of the system. Within SCARP knowledge is represented under object form. Generic formalisms are provided to describe domain specific knowledge within the system. The development of a cooperative problem solving environment for an application domain with SCARP consists in filling the system's knowledge base with domain specific problem solving knowledge.

The knowledge is then managed by the *knowledge handler* that gives access to the *knowledge base*. It provides functionalities to exploit and to modify the represented knowledge. On one hand it allows to access generic descriptions of problem solving strategies within the knowledge base. On the other hand it manages run time information concerning the problems under consideration and the associated problem solving process.

The next layer, the *task engine*, uses the functionalities provided by the knowledge handler to extract the knowledge necessary for problem-solving out of the knowledge base: problem descriptions, solving strategies, etc. It then manages the problem solving process according to these strategies. Whenever the problem-solving process requires communications with the outside world, the task engine activates the *cooperative dialogue handler*.

During problem solving, communication may be necessary towards two directions, represented by the external layers *user interface* and *system interface* (the interface to the operating system). Any kind of communication is managed by the *cooperative dialogue handler*. The *system interface* allows to access external data files and especially to control the execution of external software programs. The *user interface* allows to cooperate with the user. Different dialogue protocols are defined that allow to manage the system-user interaction. Such a dialogue might be initiated by the *task engine* in order to obtain information necessary for the problem solving process, such as parameter values for example. But also the user can activate a dialogue in order to modify

[1]This project is partially funded by the French Ministry of Research and Technology

different characteristics of the problem solving process. In any case, the *cooperative dialogue handler* supervises this dialogue.

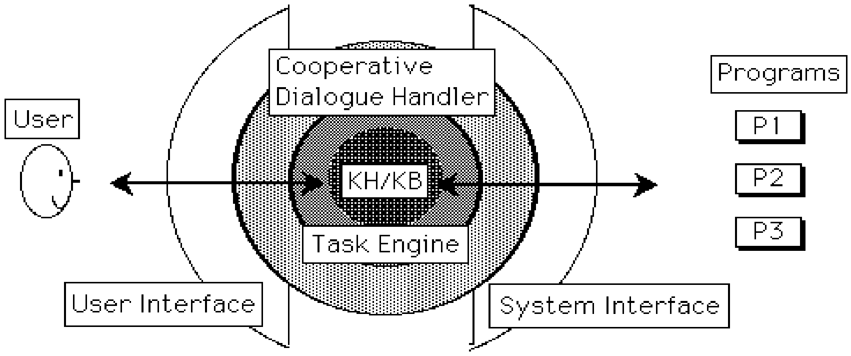

Figure 1 - the system architecture: the system is structured in layers, the *knowledge handler* (KH) with the *knowledge base* (KB) , the *task engine*, the *cooperative dialogue handler*, the *user interface* and the *system interface*.

3 The knowledge base and the knowledge handler

The knowledge base and the knowledge handler that provides access to the knowledge base form the core of SCARP. Within the knowledge base, three different knowledge types are distinguished, concerning tasks, methods and entities. Tasks describe problems and associate problem-solving strategies if possible ; in SLOT examples for such problems are correspondence analysis or principal component analysis. In SCARP problem solving is based on recursive decomposition of complex problems in more elementary subproblems. Methods constitute the leaves of these recursive task decomposition strategies. They refer to external software programs and describe their instructions of use within the system ; in SLOT methods do elementary calculation on matrices or graphical representations for example. They are executed to solve elementary tasks. Entities are introduced to represent the objects manipulated by the tasks (and methods) within the application domain, for instance matrices and graphical representations in SLOT.

Within SCARP the complete knowledge is represented under object form. The knowledge concerning entities is directly represented using Shirka, [Rech&89], an object-centered knowledge base management system, based on a notion similar to frames [Fike&85]. To represent the knowledge concerning tasks and methods a specific object centered model has been defined for SCARP. This model relies on the same elementary object centered concepts. Therefore the uniform object centered knowledge representation allows to apply the same reasoning mechanims to tasks, methods and entities. The classification mechanism is used in general to reason about object class hierarchies. It is also used by the task engine to reason about tasks. In the following, we first expose with Shirka elementary object centered concepts. This immediately shows how entities are represented via object classes. Afterwards, the object centered representation of tasks and methods is explained.

3.1 Object centered knowledge representation

Shirka is an object-centered knowledge base management system that relies on the class-instance shift. A class, in the same way as its instances, is defined in a scheme via its set of slots. A slot is in turn defined by a list of facets, and a facet by a list of values. Such a value is either a scheme or a reference to a scheme. Every slot is typed via the facet *a*(*or*list-of, for slots that support multiple values). This facet associates an elementary type (such as integer, real, ...) or a complex type (an object class) to the slot. An object is called complex if the type of at least one of his slots refers to an object class (**figure 2**). The description of a slot might be completed by other facets that constrain the domain of the slot value or that define ways to infer this value. The classes are organized in acyclic graphs, in class hierarchies. Classes describing different concepts are organized in different hierarchies. Via inheritance a class in the hierarchy transmits the knowledge and constraints defined for its slots to all its subclasses (**figure 2**). This knowledge might be precised and completed in the subclasses, but not contradicted.

Using Shirka means first to define classes and then to create instances for these classes. These instances may be incomplete in the sense that certain slot values might be missing. Two kinds of actions are then possible.

On one hand, missing slot values might be requested. Shirka tries to infer these values using the knowledge introduced with the class definition. In order to do this, Shirka disposes of inference mechanisms, such as procedural attachment, filtering, and the definition of default or class values for slots.

On the other hand the class(es) an instance might or might not be attached to, can be identified via a classification mechanism. This mechanism exploits the class hierarchy. An instance is created for a certain class. It therefore satisfies all the constraints defined for this class. The classification mechanism tries to verify for every class inside the class hierarchy whether the instance might be attached to this class or not. Starting with the initial class of the instance it examines for each of its direct subclasses whether the slot values of the instance satisfy the additional constraints defined in the subclass. If all of them are satisfied the class is said *certain*. If some constraints are violated for a class, the class itself and all its subclasses are declared *impossible*. If a new slot appears in a subclass and its value cannot be determined this class is declared *possible*. This classification algorithm is recursively applied to all the certain or possible classes, down to the leaves of the class hierarchy. The result of the classification are the lists of the certain, the possible, and the impossible classes (**figure 3**). The classified instance might afterwards be attached to one of the possible or certain subclasses of its initial class. This is called *specialization*.

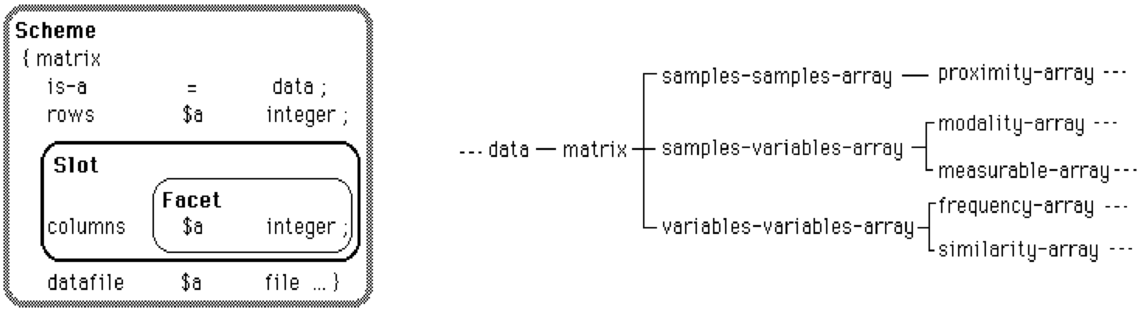

Figure 2 - A simplified class scheme and the associated class hierarchy, describing the class *matrix* in the SLOT knowledge base. The left part of the figure shows that *matrix* is a complex object: the slot *datafile* refers with *file* to another object class. This figure presents only a subset of the *matrix'* slots and facets. The right part of the figure shows a simplified part of the corresponding class hierarchy. It describes different types of matrices distinguished in data analysis.

If a slot refers to another instance, it might be necessary to classify also this refered instance inside its own class hierarchy. This is necessary if the type of the slot refering to this instance has been precised, if the type has been specialized to a subclass of the original type (**figure 3**). But here the classification is lazy ; it stops when all the constraints concerning the specialized type of the slot are verified.

For the reason that a complex instance might refer to other complex instances, this classification process might recursivley initiate partial classifications in different parts of the knowledge base's class hierarchy. As we will see, this classification mechanism is extensively used within SCARP ; it is applied to choose appropriate solving strategies depending on the actual problem context.

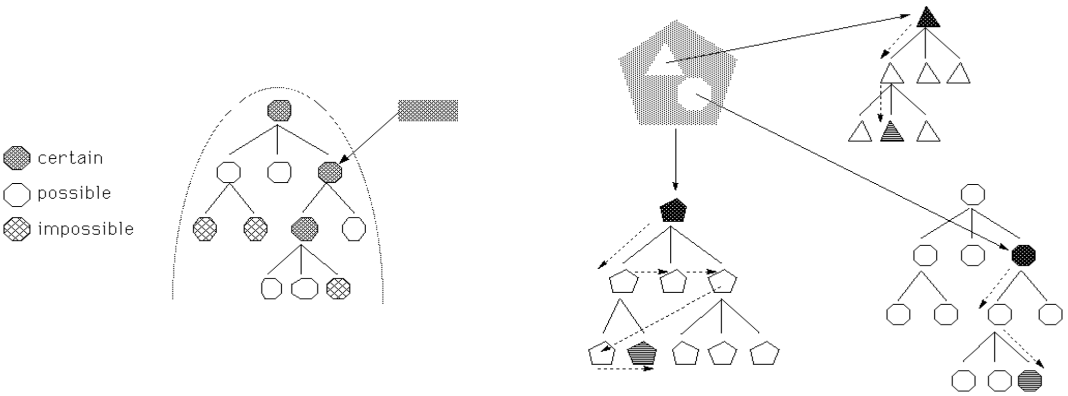

Figure 3 - The classification algorithm and the classification of a complex instance. Classification produces the lists of the certain, the possible, and the impossible classes for an instance. In the right part of the figure a complex instance represented via a pentagon is classified. In order to class the complex instance the type-constraints concerning its slots must be verified. Two of the slots refer to more specialized classes than those refered by the initial class of the complex instance. They are represented by a triangle and a hexagon. The refered instances must be partially classified in their proper class hierarchies.

3.2 Representation of tasks and methods

For a problem solving environment the essential part of knowledge concerns the tasks. Different approaches to task-oriented modeling are discussed in [Chan&92] and the importance of a declarative task description was advocated in [Smit&84]. We define a task as a model of a problem and its solution, defining a specific set of input and output entities. One problem might be defined at different abstraction levels by different, more or less specific task descriptions. At a low abstraction level, the problem defined by a task is sufficiently precise, so that a problem solving strategy can be associated to the task description. The set of input and output entities and the problem solving strategy constitute the most essential parts of a task definition. The problem solving strategy describes how a complex task can be decomposed into more elementary subtasks. Each subtask may in turn be a complex task, or refer to a method ; in this case it is directly solved through the execution of an associated software program.

Within SCARP, tasks are modeled as object classes ; the following types of slots are defined for a task:

- The *global input* and *output entities* define the functionality of the task. *Strategic input entities,* in contrast, are for example parameters needed for the task execution. *Strategic output entities* are the raw output of the task execution.
- *Pre-tasks* may be defined in order to assign values to strategic input entities. Strategic output entities may have to be transformed or synthetized to determine the global output entities. This is to be done by the *post-tasks.*
- Specialized *visualization tasks* may be associated to a task, to give a synthetic overview on the set of input or output entities of the task.
- *Preconditions* and *postconditions* can constrain the possible values for input and output entities ; one condition can concern only one or a set of these entities.
- The *control flow* describes how the subtasks are to be chained to solve the complex task itself. Different operators are available to combine subtasks: *sequence, parallel, iteration, choice* and *user-task.* A subtask is a user-task if the user indicates the result of the task. For such a task the system has no direct solving strategy, but it may have a kind of help-strategy to support the user in determining the result. Such a help-strategy might consist for example in calculating values associated to the task's results or in displaying graphics.
- For each *subtask* a specific *execution-context* describes how its execution is integrated into the execution of the complex task (data flow, whether the user has to validate input or output entities or not, whether the task is optional, ...).

Figure 4 gives an example for a task description out of the SLOT knowledge base.

As tasks are modeled by object classes, they are organized in hierarchies. Each global problem is modeled inside a separate class hierarchy. This organization provides a way to structure for each task the available problem solving strategies. Each class inside this hierarchy models the task at a different abstraction level, defining a specific problem context, and associating the adapted problem solving strategy. The problem context is described by a specific set of input / output entities and more or less constraints concerning them. A more specific task defines more constraints on its problem context and is therefore located at a lower level in the hierarchy than a more general task. Depending on the characteristics of the entities that form a precise problem context, the choice of the adequate problem solving strategy can therefore be supported by a classification process over the hierarchy of tasks. In order to execute a task the corresponding class is first instanciated. The actual application context, i.e. the input entities, is then provided. The classification – the determination of certain, possible and impossible subclasses – distinguishes here beyond the more specific tasks those that are certainly, possibly or certainly not adapted to this context. An example for task classification is given in **figure 5**.

As indicated above, the recursive task decomposition strategies finally refer to methods. Also methods are defined via object classes. These classes describe the input / output entities of the methods and associate direct means to solve them. Internal and external methods are distinguished within SCARP. Internal methods refer to a LISP-function defined by the designer of the knowledge base in order to solve an elementary problem. External methods integrate software programs and their instructions of use into the system. Their invocation is done in three phases: the first one consists in preparing the data for program execution, e.g. in extracting these data out of the input entities of the method and putting them into the specific input file used by the software program, the second one in executing the program, and the third one in concluding its execution, e.g. in reading the results out of the specific output file created by the program and in putting them into the coresponding output entities of the method. Therefore, a class modeling an external method has additional slots refering to the preparation-procedure, the software program and the concluding- procedure. Besides, it may have other slots concerning the execution, such as the one containing the directory the software program is stored in.

The knowledge handler provides the functionalities to access the knowledge represented in the knowledge base. Furthermore it provides the structures to store run-time information about task and method execution. This information is necessary to manage different versions created during the problem solving process. It concerns for example the kind of modification that led to each version or furthermore the reason for its failure.

```
{ classic-analysis
                  a-kind-of                =        simple-linear-ordination ;
  &input          data                     $a       matrix
                                           $com     "initial data - n rows, p columns" ;
  &strategic-input  label                  $a       string
                                           $default "ols"
                                           $com     "generic file name" ;
  ...
  &strategic-output eigenvalues-sticks-curve $a     sticks-curve
                                           $com     "Sticks curve of the eigenvalues" ;
  &output         analysis                 $a       simple-linear-ordination-analysis ;
                  strategy                 $value   (sequence opening-analysis making-triplet
                                                              preparing-diagonalization
                                                              diagonalization
                                                              (user-task selection-factors)
                                                              making-component-scores
                                                              closing-analysis)
  &subtask        opening-analysis         $a       { context ... } ;
        ...
  &subtask        selection-factors
              $a   { context-user
                     exec              $a       visualize-curve ;
                     value-to-ask      $default number-of-factors ;
                     where-to-put      $default .analysis.retained-factors ;
                     number-of-factors $a       integer
                                       $com     "Select the number of factors to be retained";
                     data-flow         $value
                             (.eigenvalues-sticks-curve.model > exec.model) ... ;
  &subtask        closing-analysis          $a      { context ... } }
```

Figure 4 - Simplified definition of the *classic-analysis* task out of SLOT. This class describes a classic simple linear ordination analysis. The data to analyse are structured in a *matrix* refered by the input slot *data*. The *generic-analysis-label* allows to name the files that will contain intermediate and final results of the analysis. The output of this task, *analysis* resumes the essential parameters used and the results obtained within the analysis. The execution context of each of the subtasks enumerated in the *strategy* slot is defined by a specific context. For the *user-task selection-factors* this context defines that the value to be obtained from the user is *number-of-factors*. To support the user's choice the method *visualize-curve* is called.

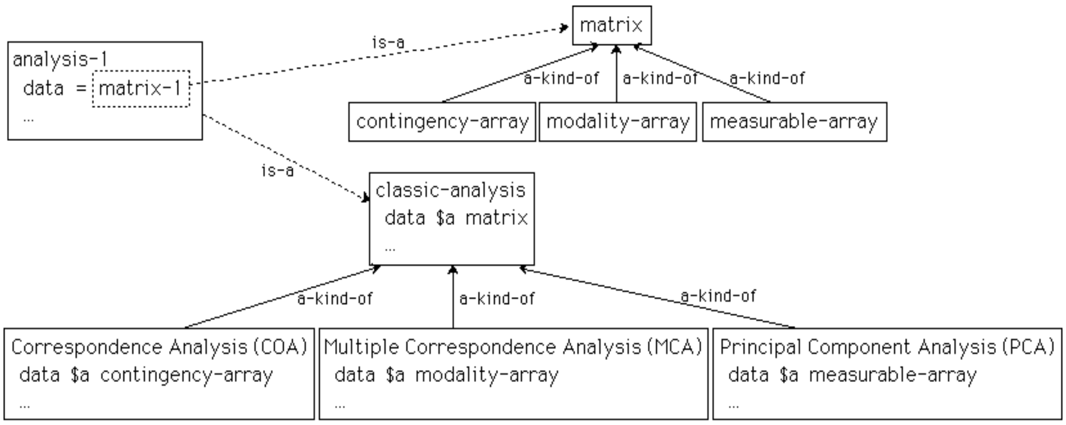

Figure 5 - Example for the classification of tasks taken out of SLOT. This example shows in the lower part the simplified class hierarchy concerning the task *classic-analysis*. The *data* to analyse are represented by a *matrix*. But the more specialized analysis types (COA, MCA and PCA) are only adapted to more specific types of matrices. When an analysis has to be done (*analysis-1*) the initial data array (*matrix-1*) is provided (upper left). The classification of *matrix-1* inside its proper class hierarchy, shown in simplified form in the upper right , allows to determine what specific analysis type can be executed.

4 The task engine

The task engine exploits the knowledge base via the knowledge handler for automatically solving complex problems. Different phases are distinguished in the problem solving process and in the task engine's functioning:

- The **instantiation phase**: the class corresponding to the initial task to execute is instantiated ; its global input entities are determined. The constraints concerning them are verified.
- The **classification phase**: according to the characteristics of these input entities, the solving strategy is chosen ; this is done applying the classification mechanism that determines which subclasses are appropriate in the specific problem context. Missing input information can be infered or asked to the user during classification. The constraints concerning this information are then verified. The task-instance is finally attached to the chosen subclass.
- The **input completion** and **validation phase**: missing strategic input entities or parameters are obtained and the preconditions defined for the task verified.
- If required by the execution-context, a general view of all the input entities is shown to the user.
- The **execution** or **decomposition phase**: the solving strategy associated to the problem description is executed. Each sub-task refers in turn either to a complex task, which is executed applying recursively the same algorithm, or to a method and with the method a software program, that is directly executed. A subtask's execution is integrated in the global problem solving process as defined in the execution-context that links it to the complex task.
- The **output completion** and **validation phase**: the global output entities are synthesized. The constraints concerning them are verified.
- If required by the execution-context, finally a general view of all the output entities is shown to the user.

A task may fail during any of these phases, either because the selected specialization was a bad one, or because some constraints concerning input or output were not satisfied, or because one of its subtasks failed, or furthermore because of a user intervention. Then the task engine backtracks: it turns back to the last decision point in order to explore another branch of reasoning. Each (sub)task is only adapted to the actual context before its immediate execution. The alternation of classification and decomposition phases allows to dynamically adapt the predefined problem solving strategy to the evolving problem context (**figure 6**).

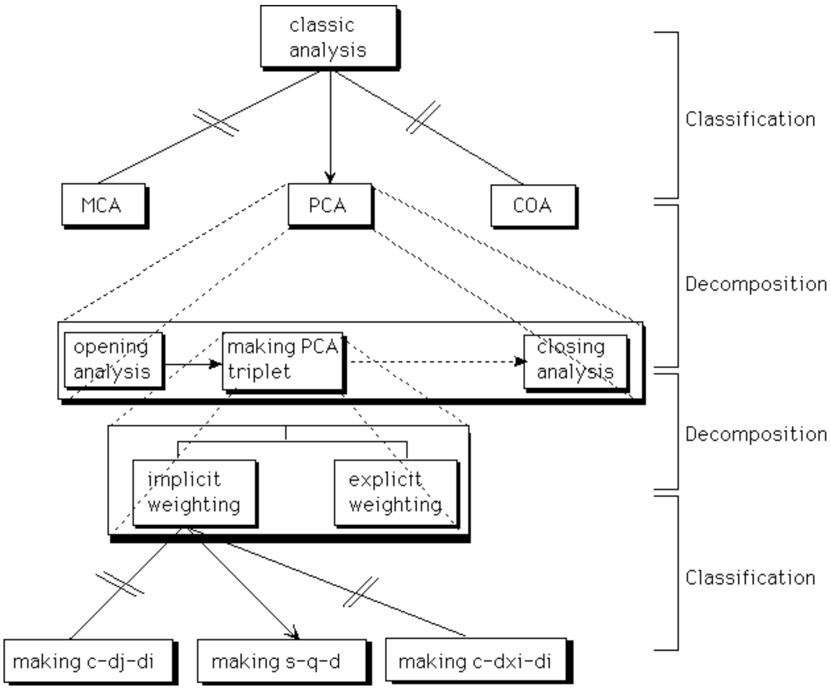

Figure 6. The alternation of classification and decomposition phases during problem solving: A classic analysis is done within SLOT. In a first problem solving phase, this task is specialized according to the problem context (especially the characteristics of the matrix to analyse) into a PCA. This task is then decomposed into its subtasks. After the execution of opening analysis the next subtask, making PCA triplet, has to be executed. It has no subclasses and is therefore directly decomposed into its subtasks ; in fact, the solving strategy indicates a choice between implicit and explicit weighting. Implicit wheigthing is chosen and then in turn classified.

This results finally in a very specific method chaining, providing an appropriate solution to the particular problem. The task engine integrates thus strategic, tactical, and operational knowledge: strategic knowledge through the ability to exploit the description of complex tasks and their decomposition, tactical knowledge via the specialization process exploiting the hierarchical structure of the knowledge representation in order to adapt the problem solving process to the actual context, and operational knowledge allowing to control the execution of external software modules.

Executing a task necessitates communication with the outside world, usually either because it indicates an external program to execute (a communication via the system interface), or because parameter values have to be fixed by the user, or in order to ask the user to solve a user-task (a communication via the user interface). Then, the task engine interrupts itself and activates the cooperative dialogue handler. The information necessary to reactivate the task engine at the end of the required communication is stored.

5 The cooperative dialogue handler

The functioning of the task engine described above corresponds to the most automatic function mode, that is the function mode where the task engine takes all the decisions it is able to take. In fact, the user might intervene in any of the problem solving phases. The problem solving process might be more or less automatic, more or less guided by the task-engine or in turn by the user. Actually, the user can decide to intervene at one or more of the different problem solving phases. Global parameters define whether the user or the system controls the different phases. The user might for example decide to control the classification phase ; this means that he will choose the specialization to execute beyond the possible more specific tasks. He migth also control the input (or output) completion and validation phases. This means that he will be asked for each task to validate the corresponding set of input (ouput) entities. Concerning the decomposition phase the user might for example decide to control all the strategic choices that have to be made. When the user has the control of a problem solving phase the task engine will, in this phase, interrupt itself and activate the cooperative dialogue handler.

The cooperative dialogue handler manages all the communications necessary between the task engine and the outside world. It is usually activated by the task engine that needs information from the user or to execute an external program. The dialogue handler then manages this request ; it initiates a specific dialogue with the user or activates the required program. In the end of the dialogue or the program execution it sends the "result" of the communication to the task engine, and reactivates it in order to continue the problem solving process. This is the "normal case": the dialogue handler is activated by the task engine, that is itself interrupted during the dialogue, expecting the result.

Besides, the cooperative dialogue handler can also be activated by the user, after the end of the problem solving process or even while the task engine is working and not aware of the dialogue. This is where further cooperative abilities are especially required. Via the user interface, described in the next section, the user always follows the problem solving process. He might at any moment decide to modify its characteristics, either parameter values, specialization or strategic choices made previously. The user then activates the dialogue handler, engaging a modification dialogue. In the end, he validates the modifications. The dialogue handler then interrupts the task engine if it was active. It creates a new version of the current problem solving process, copying unmodified parts of the previous version. Finally it (re)activates the task engine with this new, modified version. The task engine will then reexecute the part of the problem solving process that is in relation with the modification.

Different dialogue protocols are defined within SCARP corresponding to the different possible types of interaction or modification concerning the problem solving process. They define how turn taking on the control of the problem solving process can take place. Most of these dialogues are very punctual; they concern a specific decision, such as the choice of a parameter value, a strategic choice or a specialization. Within SCARP, such an interaction is managed via a separate graphic widget with its proper behaviour ; this graphic widget might be invoked either by the system or by the user. It allows the user to indicate the information requested or the modification he wants to make and to validate it. **Figure 7** shows the graphic widget that allows to choose or to modify specialization.

Another important cooperation facility is the exploration of new problems / problem solving strategies, that is of problems / strategies that are not defined within the system's knowledge base. As cited above, in our system a task models a problem with its solution, and associates the adequate problem solving strategy. A task is essentially defined by an object class via a specific set of input and output entities on one hand and a decomposition strategy on the other hand. Thus the interactive definition of new tasks consists first in indicating its input / output entities, and then in executing an adequate strategy by indicating which are the subtasks, how they are to be chained together and what is the dataflow in order to solve the global task. Each of the subtasks may in turn be a new task and interactively defined. An example for new task definition is given in section 7.3. Newly explored strategies may be stored in the system's knowledge base. They are not placed at the corresponding place inside the task class hierarchy yet, but in a seperate hierarchy.

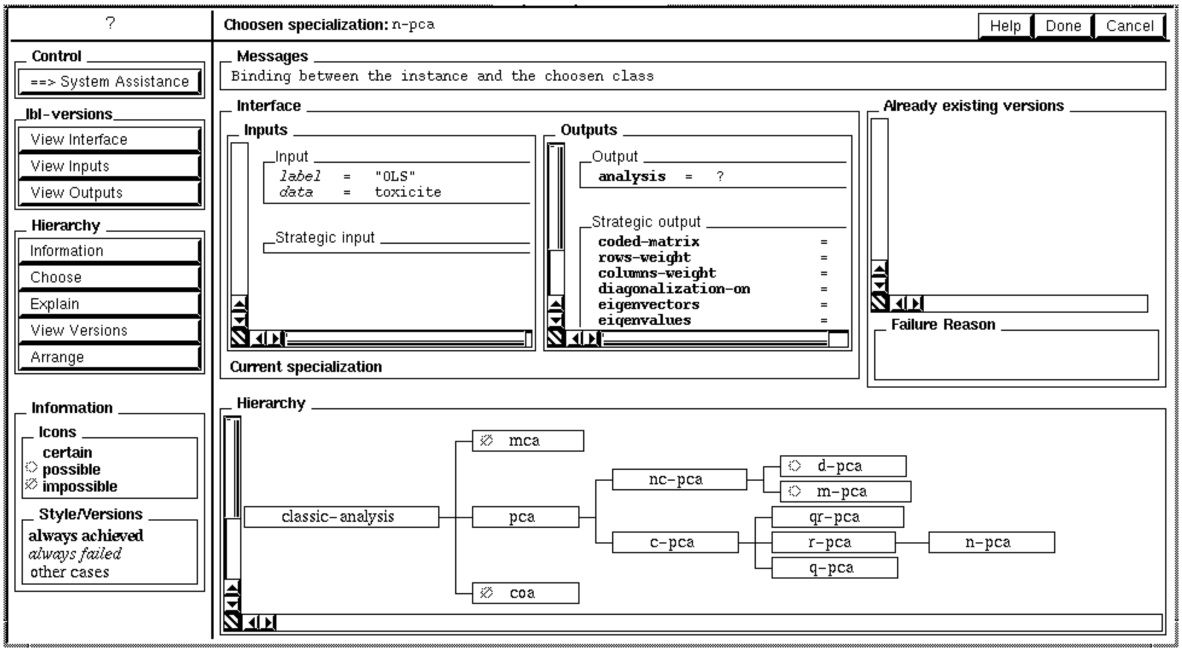

Figure 7. The graphic widget allowing to control specialization: This screen is structured in three parts. In the upper left part the input / output description of the task to specialize are shown. In the upper right part existing versions for a more specialized class are accessible. In the lower part the corresponding task hierarchy is shown. Corresponding to the characteristics of the known input entities of the task (here the data *toxicite*, a measurable array, (cf. **figure 5**) the specializations are marked sure (*pca*, ...), possible (*d-pca, m-pca*), or impossible (*mca, coa*). The buttons on the left allow to interact will the graphic widget, to ask for supplementary information about the tasks, to choose a specialization, to ask for system assistance etc.

6 The user interface

To enable the user to follow the problem solving process, to interact with the dialogue handler in order to adapt the problem solving process to his current needs, and to explore new strategies, an appropriate user interface is essential. The user interface of our system (**figure 8**) always graphically shows the current state of the problem solving process, i.e. the current task decomposition tree. It indicates which subtasks are already executed, and if they were successfully executed. Each task is represented by a separate node within the task decomposition tree. Furthermore any decision point is marked and can be inspected by the user. Different types of decisions are represented by different types of graphical nodes: S for specialization, C for strategic choice, and U for user task.

Via the interface, the user has access to all information necessary to evaluate the correctness of the problem solving process. These are the input / output entities of task and their existing versions on one hand, and the decisions made during the problem solving process on the other hand. To get this information the user just has to select the corresponding node with the mouse. A separate graphic widget then shows to him the requested information. These graphic widgets are in fact the same as the ones used by the dialogue handler for asking information to the user, e.g., the selection of a S- node will open a graphic widget such as the one shown in **figure 7**. In order to engage in a modification dialogue with the dialogue handler the user then simply selects the relevant information within this graphic widget and modifies it.

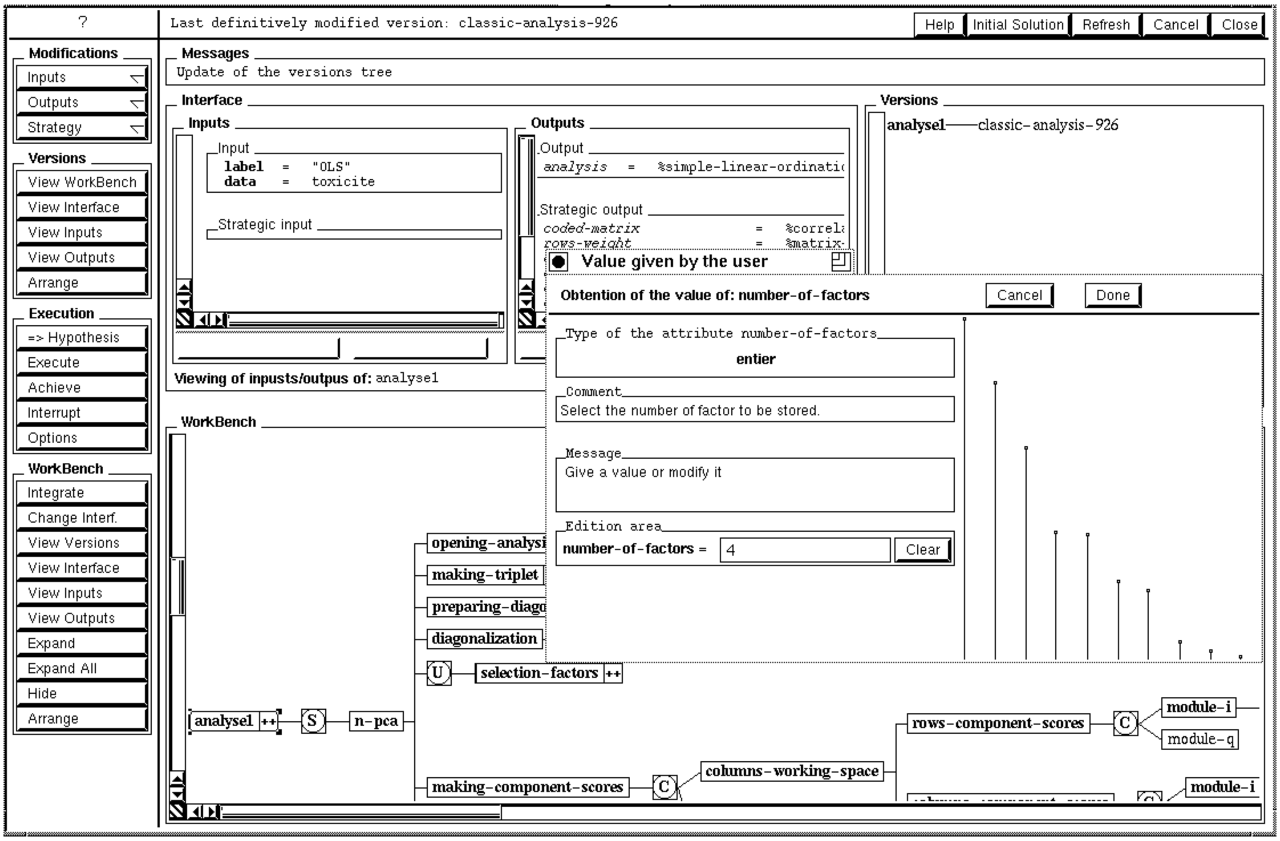

Figure 8 - the user interface: The user interface is structured in three main parts, showing the problem definition (upper left: *Interface*), the existing versions (upper right: *Versions*), and the current task decomposition tree, that represents the actual state of the problem solving process (lower part: WorkBench). The buttons on the left of the screen allow via *Modifications* of the problem definition (*Inputs/Outputs*) and of the solving *Strategy* to define new tasks, to access the different existing *Versions*, to manage the problem solving Execution process, and to control the information displayed in the *WorkBench*. This hardcopy shows the problem solving process for the task named *analyse1*, that is first specialized and then decomposed into several subtasks. The subtask actually solved is *selection- factors*. The problem definition part of the screen shows the input/output entities of *analyse1*; the *Versions* part of the screen indicates that *classic-analysis-926* has been created modifying the initial version of the problem solving process *analyse1*. The problem solving process itself is discussed in section 7.3.

7 Problem solving with SLOT, an example

This chapter presents SLOT, a cooperative problem solving environment in multivariate exploratory data analysis, developed using SCARP. The domain of data analysis generates numerous and complex problem solving strategies that may be decomposed in several elementary modules. Continually new problems are analysed and new solutions and corresponding software programs developed. Being developed for a very specific problem, these programs are often very specialized and difficult to use for other objectives. The user lacks information on how to apply them. To support the correct use of the existing methods, numerous applications were already developped, such as expert systems (e.g. MUSE [Damb&86]), decision support systems (e.g DANEX [Gaul&89]), or other environments (DINDE [Oxfo&88]). They allow to support statistic analysis in several aspects, but they do not integrate cooperative facilities such as those provided by SCARP. As indicated above, developing a cooperative problem solving environment in an application domain with SCARP consists in filling SCARP's knowledge base with domain specific knowledge. After a short introduction of multivariate analysis, the SLOT knowledge base is therefore presented. Then, an example session with SLOT illustrates the cooperative problem solving process. It shows on one hand the autonomous work of the task engine, and on the other hand the cooperation with the user.

7.1 The application domain

Multivariate analysis allows to exploit big volumes of complex data. The mass of data is generally too important for direct interpretation. Multivariate analysis permits to point out their most characteristic tendencies. It searches for a structure or homogenous groups within the data. Before doing an analysis, a concrete problem is formulated. Depending on this problem the data to gather are chosen. They represent values obtained from a set of individuals for a set of quantitative or qualitative variables. These data may be coded and exploited in different ways, e.g. directly via tables whose rows correspond to the different individuals and whose columns to the different variables. Another possibility are tables crossing two variables and representing the frequency of different value-combinations within the studied population. An example of multivariate analysis application is shown in **figure 9**.

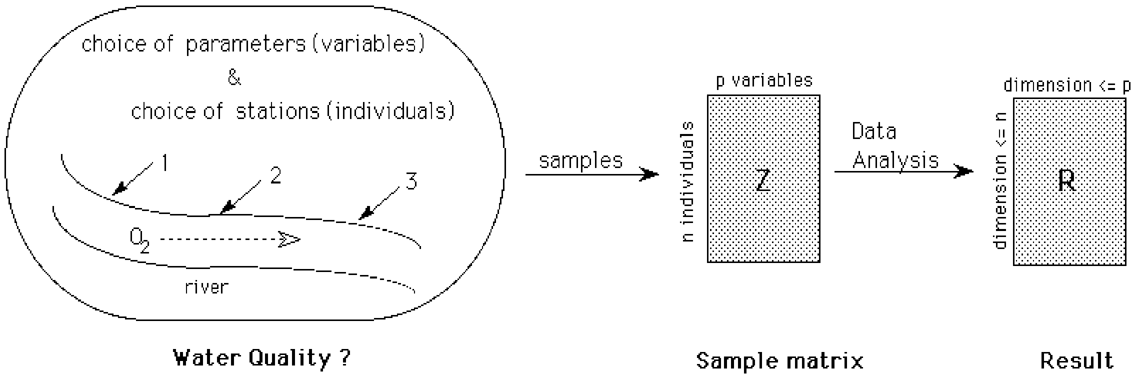

Figure 9. A sample problem of data analysis: This analysis aims at evaluating the water quality of a river depending on the place and the characteristics of the riverside. First the *n individuals*, the places where to take the samples, and the *p variables* (quantity of oxygen, etc.) to mesure are to be chosen. The samples are gathered, represented by a matrix with n rows and p columns, and then analyzed using simple linear ordination. This analysis synthesizes new characters combining the initial ones (individuals / variables). The result is a matrix in a space with less dimensions that is easier to interpret.

The data to analyze are thus represented in tables or matrices. Such a table is difficult to evaluate because of its multivariate nature. The objective is to search for a representation in a less dimensioned space. Therefore new characters are synthesized combining the initial ones (individuals / variables). This is done applying factorial and linear methods. Different statistical analysis exist that are more or less adapted to different experimental objectives and different data types. Principal component analysis is usually chosen to analyze matrices crossing individuals with quantitative variables, correspondence analysis for contingency tables, and multiple correspondence analysis for modality arrays, tables crossing individuals with qualitative variables.

For all of them, the first general step consists in coding the data. This transformation of the raw data (e.g. columns or rows centering of the initial matrix) is depending on the type of information represented by the lines and columns of the matrix and the objectives of the user. The following steps are the determination of the matrix to diagonalize, the diagonalization of the matrix (generation of an orthonormed basis), the calculation of the factorial coordinates, and finally their interpretation (supported by graphical technics). Thus, the elementary work consists in matrix calculus. Besides, graphical methods exist to support the interpretation of the analysis results.

The following paragraph exposes the knowledge formalization that was done for SLOT in multivariate analysis. It explains what kind of knowledge is necessary and how it is represented using Shirka / SCARP schemes. First, the entities of the knowledge base, then the tasks and methods are described.

7.2 Knowledge representation: entities, tasks, and methods

Two essential types of entities are handled within SLOT: *matrix* and *analysis*. Multivariate analysis consists essentially of matrix calculus. In consequence, the *matrix* class has been introduced. An analysis is not finished with the end of the calculation. It is later on interpreted and evaluated in different ways. It is therefore useful to regroup its essential characteristics in a separate class, named *analysis*. This class furthermore allows to add supplementary information related to the interpretation and evaluation phase.

As cited above, the *matrix* class represents the matrices used or produced during an analysis. The slots of the general *matrix* class regroup general information concerning a matrix. These are on one hand practical information such as the location and the name of the file containing the data, and on the other hand mathematic information such as the number of the matrix' rows and colums. As indicated before, the choice of the analysis type to execute depends on the characteristics of the initial data to be analysed. Therefore the subclasses of the *matrix* class allow to precise these characteristics, e.g. the class *samples-variables-array* indicates that the matrix rows represent individuals (or samples) and its columns variables (cf. **figure 2**).

The *analysis* class is used to collect – during and after an analysis – the information necessary for its further interpretation and evaluation. Two subclasses are defined for this class, *simple-linear- ordination-analysis* and *multiple-linear-ordination-analysis.* The first one regroups relevant information about a simple linear ordination analysis. It refers to the initial *data*, to intermediate results such as the *coded matrix*, to parameters chosen by the user that condition the analysis such as the number of *retained factors*, and to final results such as the *rows* and *columns* component scores. Furthermore, it adds supplementary information interesting for the interpretation of the analysis, its total inertia for instance (**figure 10**). The second one, *multiple-linear-ordination-analysis*, regroups relevant information about studies of couples of data tables, e.g connected analysis like PCA/PCA, inter-battery analysis, COA/MCA/ ... . These statistical analyses are not detailed here.

```
{simple-linear-ordination-analysis
        ako                              =              analysis;
        data                             $a             matrix;
        coded-matrix                     $a             matrix;
        retained-factors                 $a             integer
        total-inertia                    $a             real
                                         $if-needed     ... ;
        total-stored-inertia             $a             real
                                         $if-needed     ... ;
        factorial-rows-coordinates       $a             matrix;
        factorial-columns-coordidnates   $a             matrix;
        factorial-rows-map               $a             scatterplot-matrix;
        factorial-columns-map            $a             scatterplot-matrix;
...}
```

Figure 10. Simple Linear Ordination Analysis. This entity regroups relevant information about a *simple linear ordination analysis.* It contains the inital *data*, all the (intermediate) results obtained, and parameters used: the initial (raw) *data matrix*, the *coded matrix*, the *retained factors* etc. The slots refers in most cases to an instance of the *matrix* class. Other slots allow a posteriori to evaluate the analysis, e.g. the slots concerning its inertia. Their value might, if needed, be inferred from other values.

In SLOT, tasks are defined at different levels of complexity. At the highest level preordination, ordination and postordination are defined. Preordination allows to precise the concrete characteristics of the data contained in a data file. This task corresponds in fact to a classification of the corresponding matrix inside the class-hierarchy (cf. **figure 2**). Ordination is a complex task regrouping via its specialization different types of multiple linear ordination and of simple linear ordination analysis (**figure 11**).

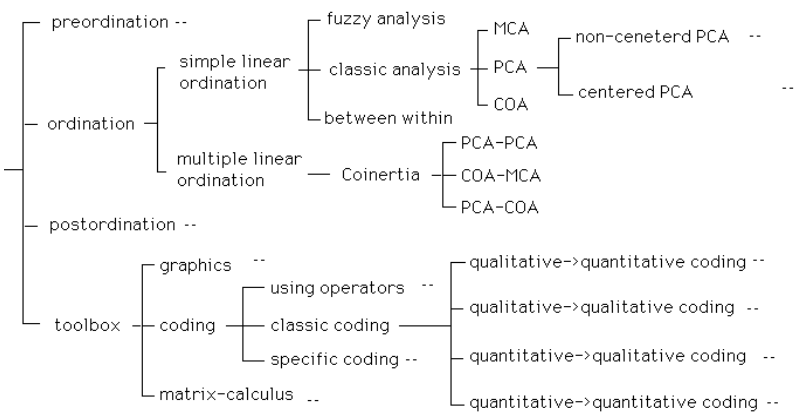

Figure 11. Simplified part of the task hierarchy in the SLOT knowledge base. This hierarchy describes in the upper part high level tasks such as preordination, ordination, and postordination, and in the lower part the tasks used as their immediate subtasks. The different tasks are described at different abstraction levels. At each abstraction level its description, especially the description of its input / output entities and of the solving strategy, is getting more and more precise. An example of a task description is shown in **figure 4**.

Postordination tasks correspond to the final interpretation and evaluation of the analysis. In the toolbox, at a lower complexity level, the subtasks of ordination are defined. They concern the coding step, matrix calculus, and graphical representations.

The methods defined in SLOT concern essentially elementary matrix calculus and calculus necessary for graphical representations. Examples are the addition of two matrices for elementary matrix calculus, and the calculation of a sticks-curve for graphic representations.

7.3 A problem solving session

This paragraph describes first interesting parts of the problem solving process corresponding to the screen shown in **figure 8** and gives then a short example for the creation of a new task within SLOT.

In the beginning of a session, the user sets general options concerning the control of the problem solving process. He decides for example to control the specialization phase. Then, the user chooses between the tasks represented in the knowledge base which one to execute. He is in this phase guided by the system that exploits the different task hierarchies defined by the system's knowledge base designer. The user chooses here for example to do a *classic-analysis*. An instance is created for the corresponding class, named *analyse1* (cf. **figure 8**) ; the user indicates that the *data* to analyse are represented by the matrix *toxitice*. The chosen task has different specializations (cf. **figure 11**). Via the interaction window (**figure 7**), the user chooses to do a *pca* (cf. **figure 5**). After the *opening-analysis* step, *making-triplet* is to be executed in order to *code* the initial data matrix and to calculate the rows- and columns-weighting-matrices. The following subtasks, *preparing- diagonalization* and *diagonalization*, are then executed. Then in the user-task *selection-factors*, the user has to select the number-of-factors to retain for the rest of the analysis. This is done via an interaction window, such as the one in the lower right of **figure 8**. The eigenvalues-sticks-curve, shown integrated in the right part of this interaction window, is in fact calculated by *visualize-curve* (cf. **figure 4**).

The selection of the number of factors to retain is an important decision. For this reason the user might afterwards want to modify it. This is easily achieved, selecting the corresponding U-node with the mouse. The corresponding interaction window reappears and the user can modify the *number-of-factors* previously chosen (**figure 8**). Once this modification validated by the user, it is taken into accout by the cooperative dialogue handler that interrupts the task engine if it was active, creates the corresponding new version copying the problem solving process preceeding *selection- factors*, and reactivates the task engine in order to finish solving this new task version. The version *classic-analysis-926* for example was created this way.

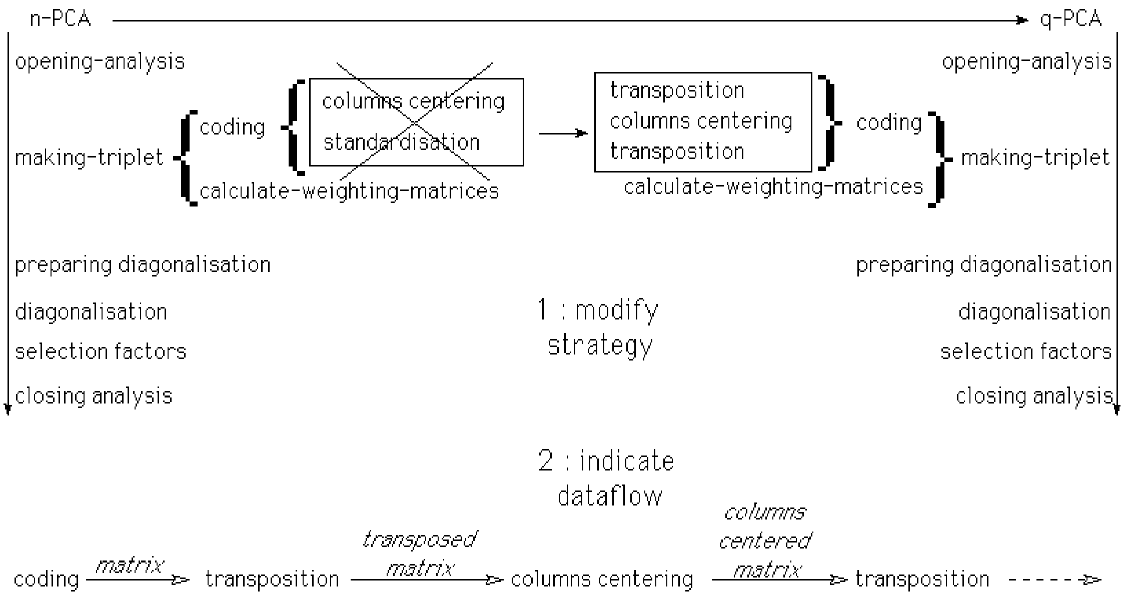

Figure 12. New task creation. A task corresponding to *q-PCA* (rows centering PCA) is created out of the task defining *n-PCA* in the SLOT knowledge base. *Q-PCA* has the same input/output entities as *n-PCA*, but a slightly different solving strategy. Therefore the task creation consists only in modifying this strategy (1) and then in indicating the new, supplementary data flow (2). The difference between *n-PCA* and *q-PCA* concerns the subtask *making-triplet*, consisting itself in *coding* and *calculate-wheighting-matrices*. The strategy of the *coding* step is modified in order to obtain *q-PCA* (1). Then an exemplary execution of the new task is done ; the systems asks the user to indicate the data flows concerning the new subtasks (2). This data flow is then stored with the new task.

The creation of a new task can concern in general the definition of its input / output entities, the problem solving strategy, and the data-flow between the task and the subtasks. In the following, only a very simple example for task creation is given. Supposing that the SLOT knowledge base only contains the strategy for normalized (n-)PCA, this example shows how it is possible to add the task q-PCA (rows centering PCA). The set of input / output entities is the same for both of them, but their strategy is different. The difference lies in the coding step. For n-PCA, this step consists in *columns centering* and afterwards in *standardization* ; for q-PCA in a *transposition* of the matrix and in *columns centering* of the transposed matrix. Thus the creation of the q-PCA task consists first in eliminating the *standardization* step and in introducing the *transposition* step before the columns centering. All the modifications are done interactively. Then, an exemplary execution allows the user to indicate the data flow, especially concerning the new subtask *transposition* (**figure 12**). After this sample execution the user can validate the new task ; it is then added to the knowledge base.

8 Conclusion

The main characteristics of our system are its genericity, cooperativeness and customizability to the users needs. It is generic because the problem solving environment shell is not domain dependent. Developing a domain specific problem solving environment consists in formalizing and structuring the domain specific knowledge in the above presented, system-specific way. The knowledge modeling to be done concerns first of all the tasks, i.e. possible problems and the associated solving strategies in the application domain. Their analysis leads to the identification of interaction points with the user, the user-tasks, of elementary tasks and methods, corresponding to the leaves of task decomposition, and of the entities, corresponding to the objects manipulated by tasks and methods. [Jean&91] describes an application of SCARP to offshore structures vibration analysis.

The system is cooperative because the control of the problem solving process can be dynamically shared between the system and the user. Even when the system has the control of the problem solving process the user can intervene at any moment. This is the great difference between our system and traditional expert system shells. The user can always decide to interrupt the problem solving process and to modify it via a specific dialogue.

A user intervention might be "soft", that is concern only a specific decision (a given parameter value, a specialization or a strategic choice). This results in the generation of different versions of the problem solving process, that are managed by the system. But a user intervention might also be "hard", that is concern the modification of the whole problem solving strategy. Then, the user explores new problem solving strategies. These strategies may afterwards be stored as "macros", i.e. new tasks, in the system's knowledge base. This results in an introduction of supplementary classes into the task hierarchy.

These new classes are placed in a separate class hierarchy yet. They are accessible for the user and can be reexecuted. But, as they are not integrated inside the hierarchy corresponding to the problem they solve, they will not be taken into account within the specialization phase of the problem solving process. Therefore, an adequate placement in the corresponding class hierarchy has to be found, depending on the input / output entities of the task and on its problem solving strategy. The identification of this placement can be supported by classification and plan recognition techniques, but shall also be found in cooperation with the user.

Problem solving environments developed with the presented shell are not dedicated to a specific user group. Beginners might limit the use of the system's cooperative facilities, while experts might fully exploit them. Global parameters indicate which problem solving phases are controled by the system and which ones by the user. These parameters are easily accessible and can be interactively modified during a problem solving session.

Bibliography

[Allp89] Allport D., *A computational Architecture for cooperative systems*, International Conference on Knowledge Based Computer Systems, Bombay, India, 1989, Springer, Lecture Notes in Artificial Intelligence 4440 , pp 3-18.

[Chan&92] Chandrasekaran B., Johnson T. R., Smith J. W., *Task Structure Analysis for Knowledge Modeling*, Communications of the ACM, September 92, pp 124-137.

[Damb&86] Dambroise E., Massotte P., *MUSE: an expert system in statistics*, In: De Antonif, Lauvo N., Rizzi A. (Eds)., Proceedings in computational statistics, COMPSTAT'86, Physica-Verlag, pp 271-276.

[Fike&85] Fikes R., Kehler T., *The Role of Frame-based Representation in Reasonning*, Communications of the ACM, september 1985, Vol. 28, N¡ 9

[Fisc90] Fischer G., *Communication requirements for cooperative problem solving systems*, Information Systems, Vol. 15, No. 1, 1990 pp 21-36.

[Gall&91] Gallopoulos E., Houstis E., Rice J. R., *Future Research Directions in Problem Solving Environments for Computational Science*, Report of the Workshop "Research Directions in integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science", April 11-12 1991, Washington, D.C., CSRD Tech. Report 1259, University of Illinois, Urbana

[Gree&92] Greef H. P., Breuker J. A., *Analysing system-user cooperation in KADS*, Knowledge Acquisition (1992), 4, pp. 89-108

[Jean&91] Jean-Marie F., Rousseau B., Rechenmann F., Prévosto M., *Embedding knowledge whithin scientific computing codes.* An application to offshore structures vibration analysis, EUROCAIPEP, Rueil- Malmaison, september 1991.

[Kant88] Kant E., *Interactive Problem Solving*, IEEE Expert, 1988, Vol. 3, No. 4, pp 36-49.

[Oxfo&88] Oxford R. W., Peters S. C., *DINDE: towards more sophisticated software environments for statistics*, SIAM Journal of Scientific and Statistical Computing, V. 9, No.1, pp. 191-211

[Rech&89] Rechenmann F., Uvietta P., *Shirka: an object-centered knowledge bases management system*, Workshop "Intelligence Artificielle en Simulation Numérique et Symbolique", SCS Europe et Laboratoire de Biométrie, Génétique et Biologie des Populations, Lyon, march 22-24 1989.

[Rech91] Rechenmann F., *Modeling tasks and methods in a knowledge-based PDE solver*, IMACS'91, june 22- 26, 1991, Dublin

[Rech&92] Rechenmann F., Rousseau B., *A development shell for knowledge-based systems in scientific computing*, in Houstis E.N., Rice J.R. (eds), Expert Systems for Numerical Computing, Elsevier Science Publishers 1992, pp. 157-173.

[Smit&84] Smith R. G., Lafue G.M.E., Schoen E., Vestal S.C., *Declarative Task Description as a User-Interface Structuring Mechanism*, IEEE Computer, September 1984, pp. 29-38.

[Stol91] Stolze M., *Task Level Frameworks for cooperative expert system design*, Artificial Intelligence Communications, Vol. 4, No. 2/3, 1991

[Will&93] Willamowski J., Chevenet F., *Modeling methodological knowledge in data analysis*, Expert Systems and their Applications. Avignon 93, may 93, Vol. 1, pp. 211-220