CREWS Report Series 98 - 19

Process Centred Approach for Developing Tool Support of Situated Methods

C. Souveyet, M. Tawbi

Université Paris 1 - Sorbonne

C.R.I 90, rue de Tolbiac. 75013 Paris - France

email {souveyet, tawbi}@univ-paris1.fr

Appeared in the proceedings of the Ninth International DEXA Conference on Database and Expert Systems Applications, University of Vienna, Austria, 24-28 August, 1998.

Process centred approach for developing tool support of situated methods

Carine Souveyet, Mustapha Tawbi
email {souveyet, tawbi}@univ-paris1.fr
Centre de Recherche en Informatique (CRI), University of Paris1 Panthéon Sorbonne,
90 rue de Tolbiac 75013 Paris, France, tel + 33 1 40 77 46 34 (46 04).

abstract :
A reuse based approach to construct software tool supporting a specific method built « on the fly » is presented in this paper. Its originality is to apply a process view on the reusable component which is either method chunk or software chunk. Our process view leads to construct method and software tool by assembling chunks. Chunks which are either method or software ones can be assembled in different ways depending on the situation of the project at hand. One of these ways is called a method or a software path. Our aim is to support the construction of a specific method path and its use in a CASE environment. The paper focuses on (i) the description of a process centred approach for software tool development (ii) its use in the development of the tool « l'écritoire » and (iii) the support of software engineer through a set of guidelines.

Keywords : Software Engineering, Reuse, Process, Method Engineering

1 Introduction

Reuse is nowadays a way of working which applies in many situations. Reuse was initially introduced by the software engineering community[Krueger 92], [Meyer 91] [Booch 87] in order to avoid repetitions in coding. As a technique, reuse was introduced in object oriented programming languages through concepts such as class, class template and inheritance.

However, reuse has crossed the boundaries of programming and demonstrates its effectiveness in most of the steps of the system development life cycle. In fact it is argued that the effectiveness of reuse is increased in the early stages of the software development. Following this line reuse is introduced in domain analysis [Priéto-Diaz 87a,b] [Priéto-Diaz 90] with the purpose of capturing and storing domain knowledge in a domain knowledge base. Reuse of this domain knowledge facilitates eliciting the requirements imposed to the system to be developed.

Reuse based approaches have been recently introduced in the Information System Engineering (ISE) community [IFIP 96] [Harmsen 94] [Plihon 98] to construct methods « on the fly » by assembling method chunks stored in a method base. The aim in this case is to avoid the use of a method which is inappropriate to the situation of the project at hand. Reuse is combined with capitalisation of chunks of good practice and constitutes a means to benefit from past experiences in a given project.

In the context of the CREWS[1] project, we have defined a collection of method chunks to support elicitation of requirement engineering (RE) by coupling goal modelling and scenario authoring [Rolland 98a] [Rolland 98b]. These chunks can be assembled in different ways depending on the situation of the RE project at hand. We will refer to one of this way as a method path. The aim of the project is complementarily to support the construction of a specific path and its use in a CASE environment. In order to fulfil this objective we developed a process centred approach for software tool development which is the subject of this paper. We associate to each method chunk a software chunk and by analogy with the notion of method path we view the software tool as a path among software chunks. For each specific situation, one of the predefined path can be reused or a new one can be constructed by assembling the predefined software chunks. A software chunk is a module which is composed of two parts : a body which codes the method guidelines and an interface which defines the adaptation parameters. We shall see that its use requires two types of parameter : the method related ones to ensure method chunks (MC) coupling and the technical ones to ensure the interoperability of the tool with the technical environment in which it functions.

The notions of a method chunk and method path are presented in section 2 whereas the corresponding notions of software chunk and software path are developed in section 3. Section 4 presents the current implementation of « L'écritoire », the prototype implementing the CREWS situated method. In section 5, we propose guidelines to help the software engineer in implementing the software path. Finally, some conclusive remarks are made in section 6.

2 The notions of method chunk and method path

2.1 Method chunk

A *method chunk* is a reusable component embedding guidelines to achieve a specific intention (or goal). As shown in figure 1, a method chunk has an interface and a body.

The chunk interface is a pair <situation, intention>. The situation is the part of the product required as a pre-requisite for the fulfilment of the intention. The intention is the goal to be achieved in that particular situation. For example the following interface <situation=(*Goal* : *g*) intention=*Elicit scenario using CREWS authoring guidelines*> expresses the fact that a *Goal* is a required product part for achieving the intention to elicit a *Scenario.* The interface <situation=(*Scenario : s),* intention=*verify scenario by checking the vocabulary*> characterises a method chunk to guide the *verification of a scenario by checking the vocabulary* which is applicable in a situation where a *scenario* has been already elicited. As shown by these two examples, the interface of a method chunk situates its context of application.

[1] This partly funded by the Basic Research Action CREWS (ESPRIT n°21.903). CREWS stands for Cooperative Requirement Engineering With Scenarios.

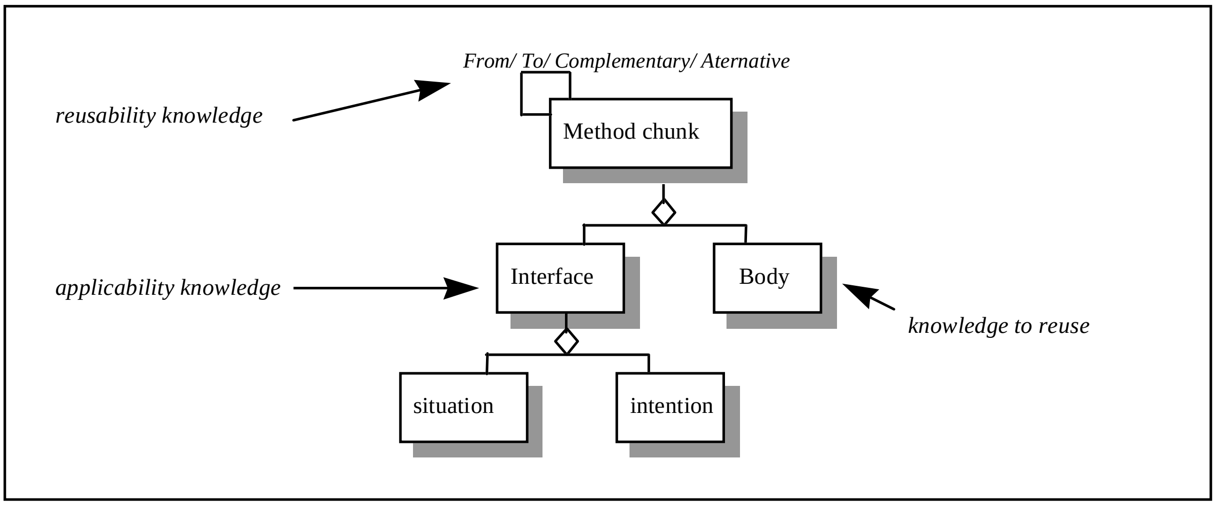


figure 1 : Overview of knowledge embedded into a method chunk.

The body chunk contains the guidelines to be applied when the chunk actually used. For capturing a large range of guidelines, the body is either formal or informal. The former implies to define guidelines as proposed by the Nature process formalism [Rolland 94] [Plihon 95]. Detailed examples of such a description can be found in [Rolland 96], [Rolland 98 c]. The latter describes guidelines in natural language.

In addition to facilitate the selection of a chunk when constructing a situated path a method chunk includes contextual information. These are expressed (see figure 1) by three types of relationships *from/to, alternative and complementary*.
The *from/to* relationships indicates chunks which can be used before or after a given chunk. For example, <situation=(*Scenario : s*), intention=*verify scenario by using linguistic completion rules*> may be preceded by a chunk analysing the semantics of a scenario content. The interface of the chunk is as follows :
< situation=(*Scenario : s*), intention=*analyse scenario semantics with a linguistic case grammar*>. Therefore, a *from* relationship is used to connect the first chunk(*verify*) to the second one (*analyse*) and a *to* relationship is used to related the « *analyse* » chunk to the « *verify* » chunk.

The *alternative* and *complementary* relationships allow to classify method chunks achieving the same goal with different manners. For example, « *elicit scenario in full prose* », « *elicit scenario with template* » and « *elicit scenario by using CREWS authoring guidelines* » are three chunks with the same goal (*write a scenario*) but several manners to achieve the goal. The first chunk supports a free text approach whereas the second constraints its user a template text approach and the last one promotes an intermediate approach with authoring guidelines.

A *complementary* relationship is used between two chunks when their ways of achieving the same goal are complementary. For example, « *Verify scenario by checking the scenario vocabulary* », « *verify scenario by using linguistic completion rules* » and « *verify scenario by disambiguating anaphoric references* » are complementary manners to verifying a scenario. These chunks can be used in the same method in order to achieve scenario verification. Consequently, *complementary way* relationships can be used between these three chunks.

2.2 Method path

A method path is based on the distinction between two types of method chunks : step chunk and flow chunk (figure 2). The former helps in the satisfaction of an intention impacting the product under development whereas the latter helps in selecting the next intention to make the process proceed.

In this view, we consider a process as being intentional. The process performance is a route between steps that we called the process flow. Steps achieve intentions and result in product transformation whereas the flow represents the sequence of intentions that have been considered [Si Said 97]. Our claim is that a method must support the engineer by providing guidelines to both help fulfilling an intention and selecting the most appropriate intention to be achieved in the next step.

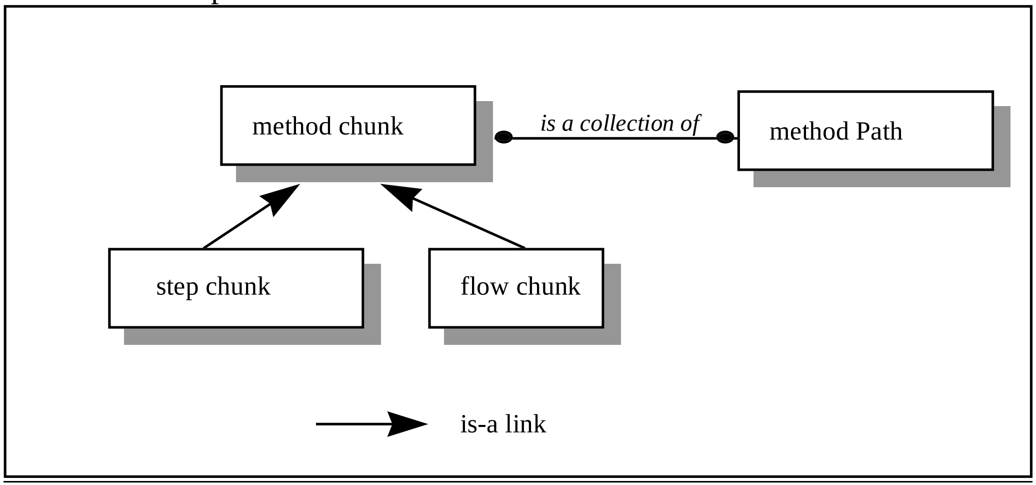


figure 2 : Method path definition.

As illustrated in the previous example a *step chunk* supports the fulfilment of an intention in a specific situation. For example, <situation=(*Goal : g),* intention=*elicit scenario using CREWS authoring guidelines*>.

A *flow chunk* guides the progression in the process. The chunk <situation=(*Scenario : s / state(s)=conceptualised),* intention= *progress by goal discovery strategy* » is a flow chunk which provides guidelines to select a way of progressing from a step where a scenario has been conceptualised. The different ways for progressing proposed by a chunk all followed the same strategy, namely a goal discovery strategy.

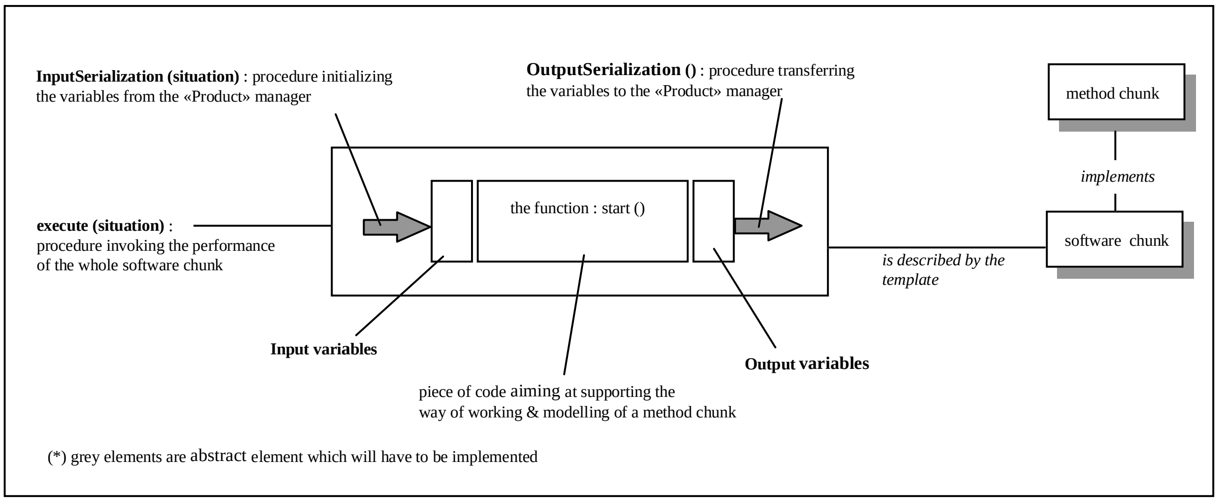
A *method path* is defined by a collection of step and flow chunks. For instance, the method path associated to the intention « Author a scenario specification » is defined by the following collection of chunks :

- step : <*(Goal :g), elicit scenario by using CREWS authoring guidelines>,*
- step : <*(Scenario :s), analyse scenario semantic with a linguistic case grammar>,*
- step *: <(Scenario : s), verify scenario by checking terminology>,*
- step *: <(Scenario : s), verify scenario by using linguistic completion rules>,*
- step *: <(Scenario : s), verify scenario by disambiguating anaphoric references>,*
- step *: <(Scenario : s), conceptualise scenario>,*
- flow *: <(Scenario s / state(s)=elicited), progress by analysing its semantic>,*
- flow *: <(Scenario s / state(s)=elicited), progress by applying a verification strategy>,*
- flow *: (Scenario s / state(s)=elicited), progress by conceptualising it*

3 The notions of Software chunk and Software path

3.1 Software chunk (SC)

A method chunk is implemented in a software chunk (see figure 3). A software chunk is reused when a new method path has been conceived and required to be implemented in a tool environment. In order to minimise the activity of reusing software chunk we propose to package its code according to a predefined template presented in figure 3.



InputSerialization (situation) : procedure initializing the variables from the «Product» manager
OutputSerialization () : procedure transferring the variables to the «Product» manager
method chunk
implements
software chunk
is described by the template
execute (situation) : procedure invoking the performance of the whole software chunk
the function : start ()
Input variables
Output variables
piece of code aiming at supporting the way of working & modelling of a method chunk
(*) grey elements are abstract element which will have to be implemented

figure 3 :template of a software chunk.

The template has been designed in order to make the software code inter-operable with different product managers. Indeed a software chunk manipulates product parts and it is important to make its code as independent as possible of a particular product manager. This independence shall ease the SC reuse in different repository managers and/or DBMS.

In order to fulfil this inter-operability goal, the template separates the body of the software chunk from its input and output variables.

Input and Output variables correspond to the product parts which are used as inputs and outputs of the chunk body. Complementarily, the *InputSerialization* and *OutputSerialization* procedures are required to map the product structure as managed by the product manager into the variables format adapted to the SC body. The *InputSerialization* procedure allows to initialise the input variables from the « product manager » whereas the *OutputSerialization* procedure aims at transferring the output variables to the « product manager ». These two procedures are abstract procedures which are not implemented in the software chunk but must be developed when the chunk is reused.

The body of the software chunk is an abstract module as it is shown in figure 3. This abstract module is defined as follows :

- the definition of variables which are local to the software chunk
- the signature and the code of the procedure *execute (<situation>)*. The situation given as input parameter is the situation of the related method chunk,

- the signature and the code of the function *start*() returning a boolean expressing if the goal of the related method chunk is achieved or not,
- the signature of the abstract procedure *InputSerialization (<situation>)*. The situation given as input parameter is the situation of the related method chunk,
- the signature of the abstract procedure *OutputSerialization* ().

Invocation of the function *execute* provokes the SC execution. This includes the execution of the *InputSerialization* procedure, the *start* function and the *OutputSerialization* procedure when the intention assigned to the method chunk has been achieved. The function start is used to return a boolean value telling if the chunk has been completely executed and thus if the intention has been achieved.

Figure 4 presents the example of the software chunk implementing the method chunk « <(Scenario : s), *conceptualise scenario* >.

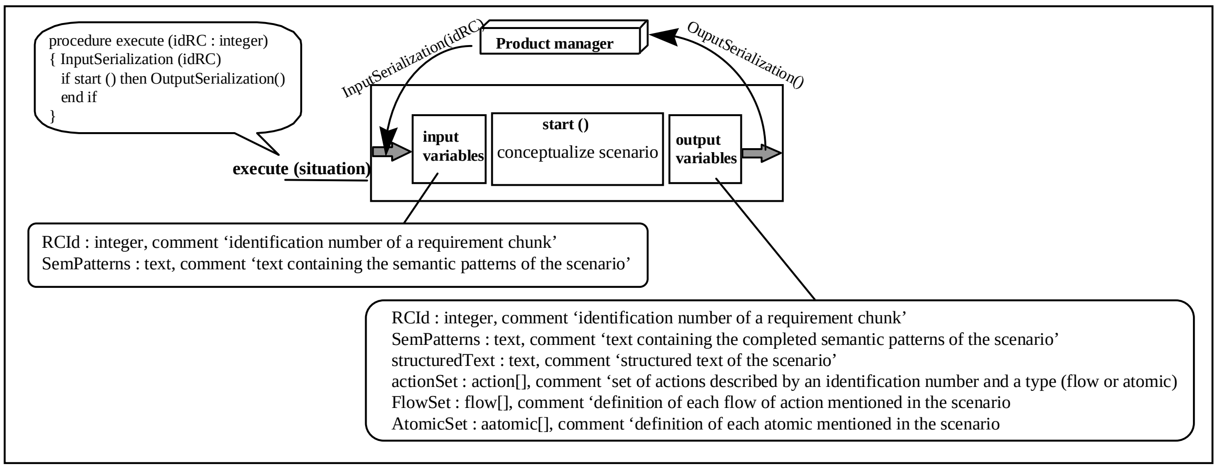

procedure execute (idRC : integer)
{ InputSerialization (idRC)
if start () then OutputSerialization()
end if
}
InputSerialization(idRC)
Product manager
OuputSerialization()
execute (situation)
input variables
start ()
conceptualize scenario
output variables
RCId : integer, comment 'identification number of a requirement chunk'
SemPatterns : text, comment 'text containing the semantic patterns of the scenario'
RCId : integer, comment 'identification number of a requirement chunk'
SemPatterns : text, comment 'text containing the completed semantic patterns of the scenario'
structuredText : text, comment 'structured text of the scenario'
actionSet : action[], comment 'set of actions described by an identification number and a type (flow or atomic)
FlowSet : flow[], comment 'definition of each flow of action mentioned in the scenario
AtomicSet : aatomic[], comment 'definition of each atomic mentioned in the scenario

figure 4 : Software chunk associated to the method chunk « conceptualise scenario ».

3.2 Software path

A software path implements a method path. Therefore it is composed of a collection of software flow chunks and software step chunks. The former define possible orderings of the latter. A software path is realised by including in every step chunk the invocation of the flow chunk which corresponds to the possible way of progressing from this step.
In order to guide the software engineer in the realisation of the path, we propose the template presented in figure 5.

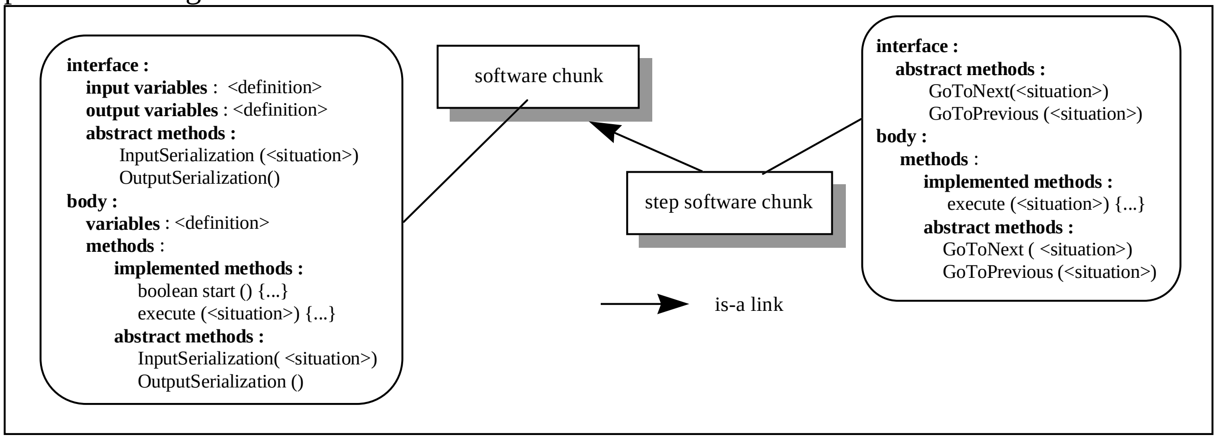


figure 5 : Definition of a step software chunk.

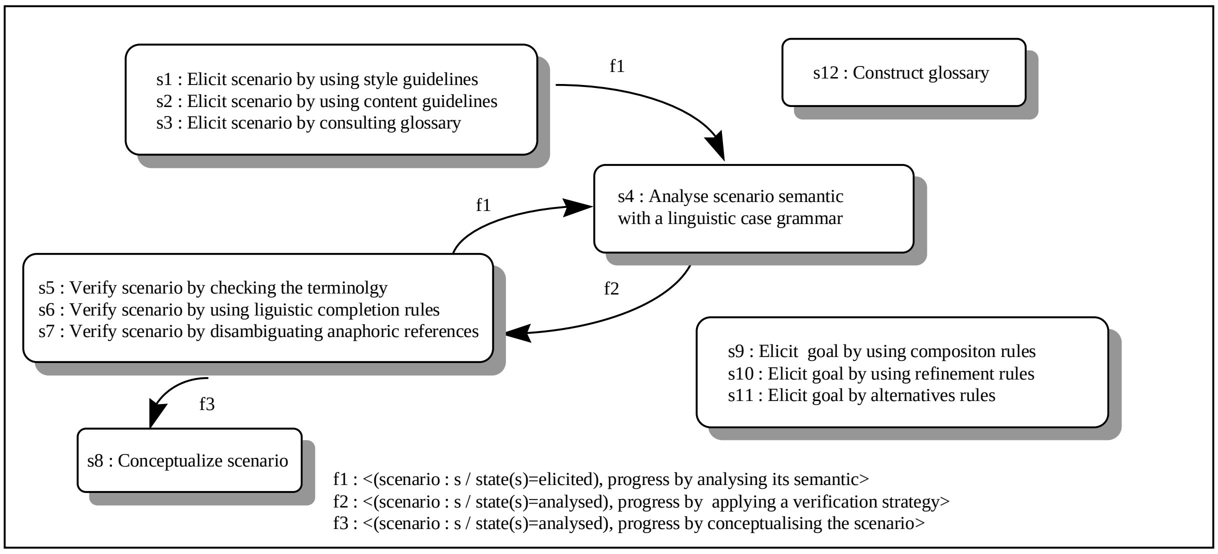
The template shows that we propose to overload the *execute* function of every step chunk. This corresponds to two abstract procedures *goToNext*() and *goToPrevious*(). These two procedures will have to be implemented by the software engineer. The *goToNext* procedure invokes the flow chunk supporting the selection of the next step intention whereas the *goToPrevious* procedure permits backtracking to the previous step. The former corresponds to the normal process proceeding. The latter occurs when the performance of the step chunk has been aborted. These two procedures are abstract because the step software chunk must be independent of what it is performed before or after it. Their implementations are made when the software chunk has to be customised according to the software path it is reused.

We are going to illustrate the use of this component-based approach for designing the software tool named « l'écritoire ».

4 Illustration of one CREWS method path in « L'écritoire » software tool

Figure 6 visualises one of the CREWS method paths which is implemented in the software prototype called « l'écritoire ».

The path is composed of twelve step chunks and three flow chunks. For sake of readability, we grouped in figure 6 step chunks corresponding to different ways of fulfilling a single intention. Thus, the path is composed of, sort to speak, 6 macro steps (in shadowed boxes). There is a predefined flow from the *elicit scenario* to *conceptualise scenario* through *analysis & verification of scenarios*. When the application engineer has initiated the process by choosing the elicit scenario intention, he will be guided through the predefined flow. Of course at any moment, he can decide to not follow the on-going flow. On the contrary the steps to *construct the glossary* and the various ways to *elicit goal* are not integrated in a predefined flow. This means that at any moment the application engineer using l'écritoire may decide to select any of these two intentions.



s1 : Elicit scenario by using style guidelines
s2 : Elicit scenario by using content guidelines
s3 : Elicit scenario by consulting glossary
f1
s12 : Construct glossary
s4 : Analyse scenario semantic with a linguistic case grammar
f1
s5 : Verify scenario by checking the terminolgy
s6 : Verify scenario by using liguistic completion rules
s7 : Verify scenario by disambiguating anaphoric references
f2
s9 : Elicit goal by using compositon rules
s10 : Elicit goal by using refinement rules
s11 : Elicit goal by alternatives rules
f3
s8 : Conceptualize scenario
f1 : <(scenario : s / state(s)=elicited), progress by analysing its semantic>
f2 : <(scenario : s / state(s)=analysed), progress by applying a verification strategy>
f3 : <(scenario : s / state(s)=analysed), progress by conceptualising the scenario>

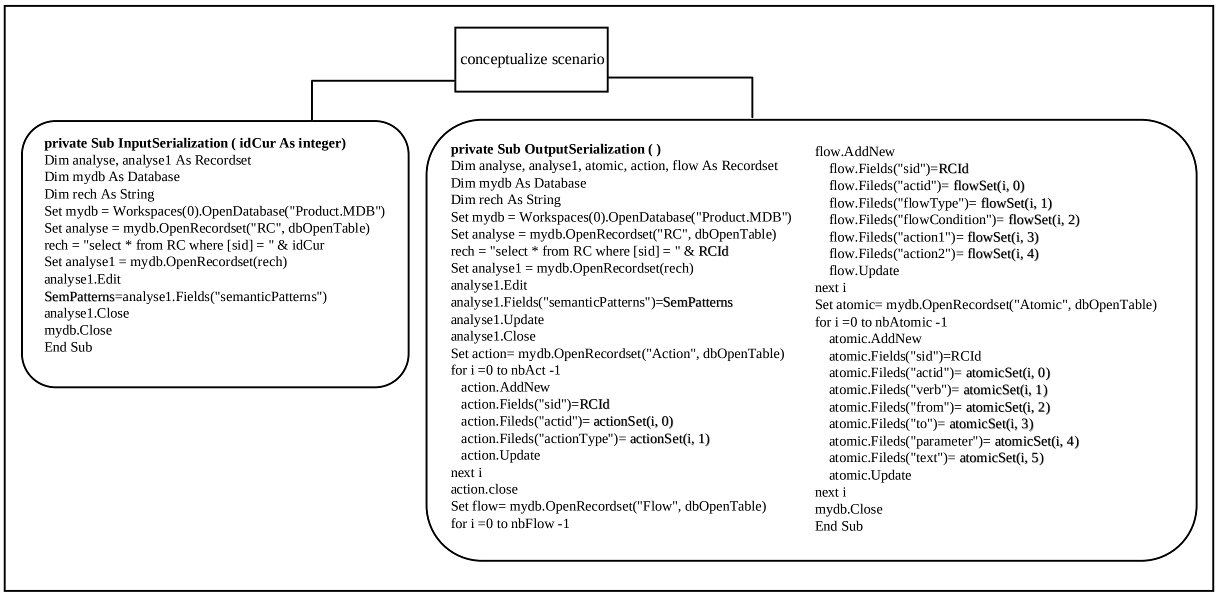
figure 6 : Example of method paths implemented in L'écritoire.

The programming environment we have chosen to explore the framework and to implement l'écritoire is Visual Basic. Every software component was designed and implemented using this programming language.

The product manager we have selected is Access DBMS because of its inter-operability with Visual Basic. The product handled by the method path shown in figure 6 is described by the following set of schema relations :

- Verb (<u>name</u>) : represents the relation storing the verbs of the glossary,
- Object (<u>name</u>) : corresponds to the objects list of the glossary,
- Actor(<u>name</u>) : is the actors list of the glossary,
- RC (<u>sid</u>, RcName, goal, manner, type, LNText, semanticPatterns, state) :allows to store information about a requirement chunk,
- Action (<u>sid, actid</u>, actionType) : permits to represent the list of actions in a scenario content of a requirement chunk,
- Flow (<u>sid, actid</u>, flowType, flowcondition, action1, action2) : explains all the flows of actions mentioned in a scenario content of a requirement chunk,
- Atomic (<u>sid, actid</u>, verb, from, to, parameter, text) : provides the list of atomic actions found in a scenario content of a requirement chunk with their definition.

The software chunks of this software path have been adapted to this product manager. It means that *InputSerialization* and *OutputSerialization* procedures of each software chunk have been implemented in order to get (resp. to transfer) product parts from (resp. to) the product manager. It implies the implementation of the *GoToNext* and *GoToPrevious* procedures of each step chunk. Figure 7 illustrates this adaptation of the software chunk related to s8 (*conceptualise scenario*). The input and output variables used in these procedures are mentioned in shadow style.

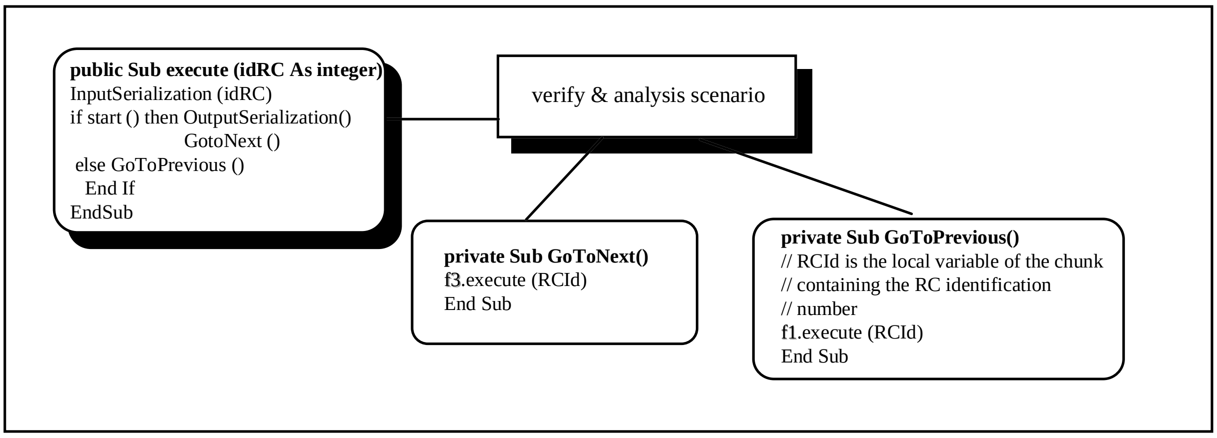


```
private Sub InputSerialization ( idCur As integer)
Dim analyse, analyse1 As Recordset
Dim mydb As Database
Dim rech As String
Set mydb = Workspaces(0).OpenDatabase("Product.MDB")
Set analyse = mydb.OpenRecordset("RC", dbOpenTable)
rech = "select * from RC where [sid] = " & idCur
Set analyse1 = mydb.OpenRecordset(rech)
analyse1.Edit
SemPatterns=analyse1.Fields("semanticPatterns")
analyse1.Close
mydb.Close
End Sub
```

```
private Sub OutputSerialization ( )
Dim analyse, analyse1, atomic, action, flow As Recordset
Dim mydb As Database
Dim rech As String
Set mydb = Workspaces(0).OpenDatabase("Product.MDB")
Set analyse = mydb.OpenRecordset("RC", dbOpenTable)
rech = "select * from RC where [sid] = " & RCId
Set analyse1 = mydb.OpenRecordset(rech)
analyse1.Edit
analyse1.Fields("semanticPatterns")=SemPatterns
analyse1.Update
analyse1.Close
Set action= mydb.OpenRecordset("Action", dbOpenTable)
for i =0 to nbAct -1
   action.AddNew
   action.Fields("sid")=RCId
   action.Fileds("actid")= actionSet(i, 0)
   action.Fileds("actionType")= actionSet(i, 1)
   action.Update
next i
action.close
Set flow= mydb.OpenRecordset("Flow", dbOpenTable)
for i =0 to nbFlow -1
flow.AddNew
   flow.Fields("sid")=RCId
   flow.Fileds("actid")= flowSet(i, 0)
   flow.Fileds("flowType")= flowSet(i, 1)
   flow.Fileds("flowCondition")= flowSet(i, 2)
   flow.Fileds("action1")= flowSet(i, 3)
   flow.Fileds("action2")= flowSet(i, 4)
   flow.Update
next i
Set atomic= mydb.OpenRecordset("Atomic", dbOpenTable)
for i =0 to nbAtomic -1
   atomic.AddNew
   atomic.Fields("sid")=RCId
   atomic.Fileds("actid")= atomicSet(i, 0)
   atomic.Fileds("verb")= atomicSet(i, 1)
   atomic.Fileds("from")= atomicSet(i, 2)
   atomic.Fileds("to")= atomicSet(i, 3)
   atomic.Fileds("parameter")= atomicSet(i, 4)
   atomic.Fileds("text")= atomicSet(i, 5)
   atomic.Update
next i
mydb.Close
End Sub
```

Figure 7 : Illustration of the technical adaptation of the « *conceptualise scenario* »chunk.

In addition each step chunk of this software path has been connected to the flow chunk helping to progress from it or to backtrack to its previous step. The adaptation of the step chunk to the method path is illustrated in figure 8 by showing the code of the *GoToPrevious* and *GoToNext* procedures. The flow chunks invoked are mentioned in a shadow style. As a reminder, we give the code of the *execute* procedure which is similar to all the steps.

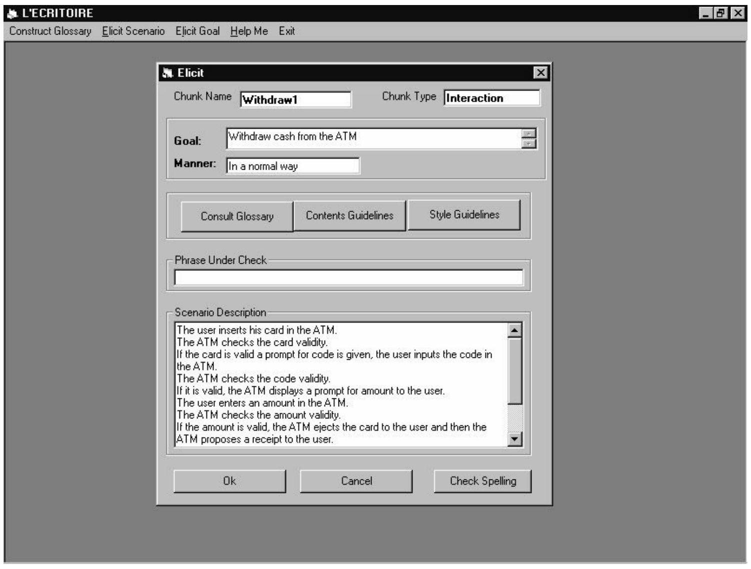


```
public Sub execute (idRC As integer)
InputSerialization (idRC)
if start () then OutputSerialization()
                 GotoNext ()
 else GoToPrevious ()
   End If
EndSub
```

```
private Sub GoToNext()
f3.execute (RCId)
End Sub
```

```
private Sub GoToPrevious()
// RCId is the local variable of the chunk
// containing the RC identification
// number
f1.execute (RCId)
End Sub
```

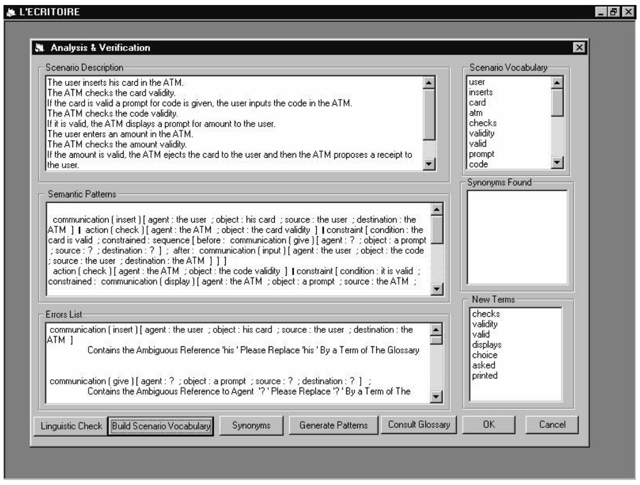
Figure 9, 10 and 11 illustrate three steps namely to *elicit, to analyse and to conceptualise a scenario*. In figure 9, the three ways of eliciting a scenario implemented in s1,s2 and s3 by the corresponding button. The figure shows an excerpt of a textual scenario produced by the application engineer using the style guidelines. When the application engineer pushes the OK button, the f1 flow chunk is executed and the window presented in figure 9 is displayed.



Chunk Name: Withdraw1
Chunk Type: Interaction
Goal: Withdraw cash from the ATM
Manner: In a normal way
Phrase Under Check
Scenario Description
The user inserts his card in the ATM.
The ATM checks the card validity.
If the card is valid a prompt for code is given, the user inputs the code in the ATM.
The ATM checks the code validity.
If it is valid, the ATM displays a prompt for amount to the user.
The user enters an amount in the ATM.
The ATM checks the amount validity.
If the amount is valid, the ATM ejects the card to the user and then the ATM proposes a receipt to the user.

figure 9 : The *elicit scenario* chunk.

The scenario elicited in the previous step appears in the upper text field of the window. In order to perform the s4 chunk, the application engineer has to pushed the button « linguistic check ». This trigger the analysis performed using a linguistic case grammar. The result appears in the two text fields as shown in figure 10.They correspond to (a) the instantiated case patterns and (b) the errors detected by the semantic analysis. By pressing the OK button, the verify scenario step is suggested by the f2 flow chunk. Let assume that the verification has been performed and that following f3 flow chunk the « *conceptualise* » step is triggered as presented in figure 11.



```
The user inserts his card in the ATM.
The ATM checks the card validity.
If the card is valid a prompt for code is given, the user inputs the code in the ATM.
The ATM checks the code validity.
If it is valid, the ATM displays a prompt for amount to the user.
The user enters an amount in the ATM.
The ATM checks the amount validity.
If the amount is valid, the ATM ejects the card to the user and then the ATM proposes a receipt to
the user.
```

```
user
inserts
card
atm
checks
validity
valid
prompt
code
```

```
 communication ( insert ) [ agent : the user  ; object : his card  ; source : the user  ; destination : the
ATM  ] | action ( check ) [ agent : the ATM  ; object : the card validity  ] | constraint [ condition : the
card is valid  ; constrained : sequence [ before :  communication ( give ) [ agent : ?  ; object : a prompt
 ; source : ?  ; destination : ?  ]  ;  after :  communication ( input ) [ agent : the user  ; object : the code
; source : the user  ; destination : the ATM  ]  ]  ]
 action ( check ) [ agent : the ATM  ; object : the code validity  ] | constraint [ condition : it is valid  ;
constrained :  communication ( display ) [ agent : the ATM  ; object : a prompt  ; source : the ATM  ;
```

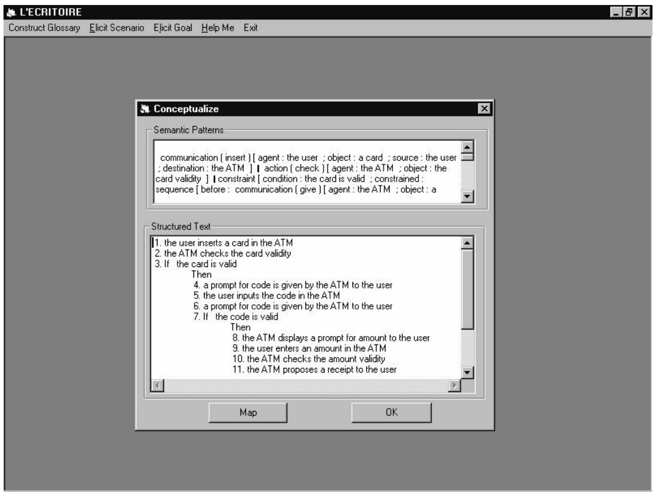
```
communication ( insert ) [ agent : the user  ; object : his card  ; source : the user  ; destination : the
ATM  ]
            Contains the Ambiguous Reference 'his ' Please Replace 'his ' By a Term of The Glossary


communication ( give ) [ agent : ?  ; object : a prompt  ; source : ?  ; destination : ?  ]  ;
            Contains the Ambiguous Reference to Agent  '? ' Please Replace '? ' By a Term of The
```

```
checks
validity
valid
displays
choice
asked
printed
```

Figure 10 : the *analysis* step chunk.

A shown in figure 11 the conceptualisation step generates a formalised scenario description presented in an indented format. This is achieved by pushing the « map » button. This formalisation uses the semantic patterns instances generated in the previous step.

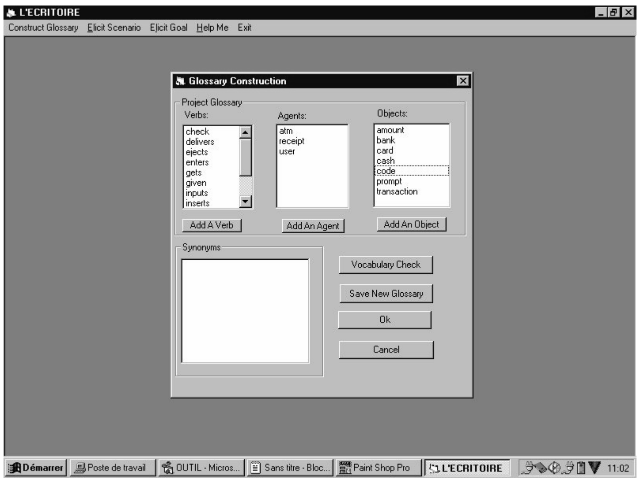


```
 communication ( insert ) [ agent : the user  ; object : a card  ; source : the user
 ; destination : the ATM  ] | action ( check ) [ agent : the ATM  ; object : the
card validity  ] | constraint [ condition : the card is valid  ; constrained :
sequence [ before :  communication ( give ) [ agent : the ATM  ; object : a
```

```
1. the user inserts a card in the ATM
2. the ATM checks the card validity
3. If   the card is valid
           Then
            4. a prompt for code is given by the ATM to the user
            5. the user inputs the code in the ATM
            6. a prompt for code is given by the ATM to the user
            7. If   the code is valid
                       Then
                        8. the ATM displays a prompt for amount to the user
                        9. the user enters an amount in the ATM
                        10. the ATM checks the amount validity
                        11. the ATM proposes a receipt to the user
```

Figure 11 : the *conceptualise* chunk.

Pushing the OK button of the window shown in figure 11 ends the predefined flow initiated by elicit scenario. Thus, the tool presents the main menu which has shown in figure 12 comprises the three top level intentions of the path namely elicit scenario, construct glossary and elicit goal.



Verbs:	Agents:	Objects:
check	atm	amount
delivers	receipt	bank
ejects	user	card
enters		cash
gets		code
given		prompt
inputs		transaction
inserts		

Figure 12 : The main menu of « L'écritoire ».

The development of the software tool « l'écritoire » has followed a specific process that we describe in next section. These guidelines help software developers to use our reuse centric approach to build a software tool.

5 Guidelines supporting a software path definition from a method path

The purpose of this section is to help the method engineer to build his software path from a method path definition. As a reminder, a method path is a collection of method chunks which are either steps or flow chunks. Therefore, the software path of a given method path is built by selecting the software chunks implementing the method chunks belonging to this method chunk.
The purpose of the software tool development is to reuse the software chunk associated to each method chunk used in this road map. The reuse of software chunk requires their adaptation to the environment in which they must be invoked. Therefore, guidelines are provided to support the method engineer in his task.

<situation=*(MethodPath : m)* ,intention= develop the CASE tool supporting m> can be viewed as a "design by reuse" process composed of 8 steps :

step 1 : <situation=(*MethodPath : m*), intention=select product management policy > :

- The development of the product management policy leads to (i) define the product, (ii) select the product manager. This could be repository-oriented, database-oriented or file-oriented, and finally (iii) implement the product schema into the selected manager.

step 2 : <situation=(*MethodPath : m* , *ProductManager : p*, *SoftwarePath : s*), intention=optimise the steps implementation>

- group step chunks specified into the method path having « complementary or alternative» relationships (see figure 6 for « l'écritoire »).
- adapt software path to this grouping.

step 3 : <situation=(*MethodPath : m* , *ProductManager : p*, *SoftwarePath : s*), intention=optimise the flows implementation>

- group flow chunks specified into the method path if it is necessary.

step 4 : <situation=(*MethodPath : m* , *ProductManager : p*, *SoftwarePath : s*), intention=implement the *InputSerialization* & *OutputSerialization* procedures for each software chunk mentioned in s>

- Implement the *InputSerialization* and the *OutputSerialization* procedures of each software chunk mentioned in s.

- The InputSerialization procedure must take into account the decision taken in step 1. The input parameter of this function is the situation on which the chunk should be applied. The extracted information is stored locally into input variables of the chunk.

- The OuputSerialization procedure transfer created or updated information (stored in output variables) to the product manager.

step 5 : <situation=(*MethodPath : m* , *ProductManager : p, SoftwarePath : s*), intention=implement the GoToNext and GoToPrevious procedures for each step software chunk mentioned in s >

step 6 : < situation= (*SoftwarePath : s / for each i in s, state (i)=implemented*), intention= Verify the vivacity of the software chunks>

- For each flow chunk or step chunk not invoked by at least one step chunk (resp. one flow chunk), we determine if it is a mistake or if it is a head of a path which will be invoked by the application engineer. If it is the case, this flow chunk (resp. a step chunk) becomes a menu option in the CASE tool.

step 7 : < situation= (*SoftwarePath : s / for each i in s, state (i)=verified*), intention= construct the Main Window of the CASE tool>
- Each root of a path which is either a flow or a step chunk becomes a menu option. Because this chunk has a situation as input, a dialogue box should be implemented to support the application engineer in the selection of the situation on which he wants to apply this chunk. Therefore, GUI event handler of this dialogue box must invoke the *execute* procedure of the chunk with the corresponding situation.

step 8 : < situation= (*SoftwareApplication : a*), intention= optimise the code>
- Avoid to execute a procedure or a function without code or with only one instruction.
- If the situation transferred from one step to another is always the same, avoid to initialise the following step from the product manager but from variables shared between these two steps.

6 Conclusion

The paper has presented a component-based approach for developing software tool supporting a method built « on the fly ». The main characteristic of this approach is to be a reuse-centric approach for constructing the method and its software tool. It means that the method as well as the software tool are constructed by assembling method chunks (resp. software chunks). The originality of this component-based approach is to propose a process centred framework which considers process as being intentional. And the process performance in this view is a route between steps that we called the process flow. Steps achieve intentions and result in product transformation whereas the flow represents the sequence of intentions that have been considered. This novelty approach is applied in the both domains method engineering and software engineering. The former aims at building methods « on the fly » whereas the latter leads to build a software tool by reusing, assembling and adapting software chunks.
This approach has been experimented in the CREWS project by developing one of the CREWS method paths in the software tool « l'écritoire ». The experimentation has led that the process centred approach can be generalised to built faster software tool of a specific method path. But it also shown that Visual Basic is not the best programming environment for supporting the definition of abstract modules, their adaptation and their reuse. We will shift to

a more appropriate programming environment such as Java because it manages abstract classes, classes, inheritance and its portability is a strong criteria to take into account.
In addition, a design for reuse will be also defined in order to support method engineer as well as software engineer in the method base improvement.

References

[Booch 87] G. Booch : « Software components with ADA : structures, tools and sub-systems », Benjamin Cummings, 1987.
[Harmsen 94] F. Harmsen, S. Brinkkemper, H. Oei : « Situational Method Engineering for Information System Project Approaches », Proc of the conference on Methods and Associated Tools for Information Systems Life Cycle, eds A.A. Verrijn-Stuart and T.W. Olle, pub. North Holland, IFIP WG. 8.1, 1994.
[IFIP 96] Proc. Of the Conference on Method Engineering , IFIP WG 8.1, ed. Chapman & all, Atlanta 1996.
[Plihon 95] V. Plihon, C. Rolland : « Modelling Ways of Working », Proc. Of the 7th International Conference on Advanced Information Systems Engineering, CAiSE'95, Springer Verlag 95.
[Plihon 98] V Plihon, J. Ralyte, A. Benjamen, N.A.M. Maiden, A. Sutcliffe, E. Dubois, P. Heymans : « A reuse-Oriented Approach for the Construction of Scenario Based Methods »,ICSP'98 (submitted).
[Priéto-Diaz 87 a] R. Priéto-Diaz : « Domain Analysis for reusability », Proc. Of COMPSAC'97, Tokyo, Japan, 1987.
[Priéto-Diaz 87 b] R. Priéto-Diaz , P. Freeman : « Classifying software for reusability », IEEE software, vol 4, n°1, january 1987.
[Priéto-Diaz 90] R. Priéto-Diaz: Domain Analysis : an introduction », ACM SIGSOFT software engineering, vol 15, n°2, april 1990.
[Rolland 94] C Rolland, G. Grosz : « A General Framework for describing the requirement engineering Process », IEEE conference on Systems, Man & Cybernetics, CSMC'94, San Antonio, Texas, 1994.
[Rolland 95] C Rolland, C. Souveyet, M.Moreno : « An approach for defining Ways of Working », Information Systems Journal, Vol 20, N°4, pp337-359,1995.
[Rolland 96] C Rolland, V. Plihon : « Using generic chunks to generate process model fragments », Proc. Of the 2nd Conference on Requirements Engineering, ICRE'96, Colorado Springs, 1996.
[Rolland 98 a] C. Rolland, C. Ben Achour : « Guiding the construction of textual use case specifications », accepted to Data Knowledge Engineering Journal.
[Rolland 98 b] C Rolland, C. Souveyet, C. Ben Achour : « Guiding », submitted to TSE journal.
[Rolland 98 c] C Rolland, V. Plihon, J. Ralyte : « Specifying the reuse context of Scenario Method chunks », paper accepted to the Conference on Advanced System Engineering (CAiSE'98), 1998.
[Si Said 97] S. Si Said : « Guidance for requirements engineering process », Proc. Of the 8th conference on Databases & Expert Systems Applications, DEXA'97, Toulouse, 1-5 September 1997.