A Proposal For Context-Specific Method Engineering

Colette Rolland
Université Paris 1-Sorbonne
17, rue de la Sorbonne
75231 Paris Cedex 5
rolland@masi.ibp.fr

Naveen Prakash
Delhi Institute of Technology
Kashmere Gate
110006 Delhi, India
np@dit.ernet.in

Abstract

The new emerging method engineering discipline acknowledges the need for the construction of methods tuned to specific situations of development projects. This raises at least three problems (1) the representation of method fragments in a method base, (2) the formalization of the notion of project situation and, (3) the retrieval of relevant fragments for the project situation at hand. Our contribution to the first two of these problems lies in the definition of a contextual approach which enables us to represent both method knowledge (i.e. the method base contents) and method meta-knowledge (i.e. knowledge about the potential use of method fragments) as pairs of the form <situation, decision>. This emphasizes both engineering decisions and method engineering decisions, their rationale and situations of applicability. We contribute to the third problem by proposing a tight coupling of method knowledge and method meta-knowledge in the method base. This enables the formal description of the context of use of every method fragment and shall facilitate the retrieval of relevant fragments according to the situation of the project under development. The paper presents and exemplifies the method knowledge and method meta-knowledge levels.

1 INTRODUCTION

The area of method engineering has emerged in response to an increasing feeling that methods are not well-suited (Lyytinen , 1987) to the needs of their users, the application engineers. In particular, it is necessary to change methods from one business situation (Hidding, 1994) to another. Situational method engineering (Welke, 1991) is the construction of methods which are tuned to specific situations of development projects. The Situational Method Spectrum (Harmsen, 1994) organises approaches to situational method engineering according to the degree of flexibility in meeting situational needs and places them on a scale ranging from 'low' flexibility to 'high'. At the 'low' end of this spectrum are rigid methods whereas at the 'high' end is modular method construction (Harmsen, 1994). Rigid methods are completely pre-defined and leave little scope for adapting them to the situation at hand. On the other hand, modular methods can be modified and augmented to fit a given situation.

One proposal (Harmsen, 1994) to situational method engineering looks at the situation of the project to engineer a project-specific method. A method is viewed as a collection of method fragments. A fragment can be either a product fragment or a process fragment. In the former case, it captures product related knowledge of methods whereas in the latter case, it captures activity related knowledge. Method fragments are available in a method base to be assembled together to form a method.

In this approach, the project situation is at a very global level. We believe that even after discovering the project situation, the detailed engineering of a method shall still require knowledge of which fragment can be used in which method engineering situation to achieve which objective. We view the retrieval process of method fragments as being *contextual :* a method engineer is faced to *situations* at which he looks with some *decision* in mind. Supporting the retrieval process requires that knowledge should be provided about *decisional contexts* in which fragments can be used.

Further, in this approach, the project situation is discovered during a separate step in the method engineering process and the method engineer has, thereafter, to find the applicable method fragments. The method base does not carry information about the situation in which method fragments are useful and so no support can be provided in this search task. Our view is that knowledge about the *context of use* of method fragments *shall be formalized* and *stored in the method base* together with the method fragments themselves.

Our approach to method engineering proposes a shift from global and situation based method engineering to modular and context based method engineering. It has three salient features :
- We recognise that method knowledge exists at different granularity levels and different levels of abstraction. Our approach explicitly captures both kinds of method knowledge.
- Since many decisions can be made in a given situation, our approach explicitly recognises the importance of decisions, the value of decision rationale and, additionally, tightly couples situations, decisions and rationales together into the notion of context.
- In order to provide support for retrieval from the method base, the situations and decisions for which a method fragment is applicable are explicitly available in the method base.

We propose to organise the method base at two levels, the *method knowledge level* and the *method meta-knowledge level* respectively. *Method knowledge* is represented in the method base in the form of *method chunks.* These chunks are available at different levels of granularity and at different levels of abstraction. Different granularity levels are made possible by using the NATURE contextual approach (Rolland, 1994), (Rolland, 1995) which organises method knowledge as contexts, trees, and forests of trees. Chunks can be considered to be at different levels of abstraction, the component, method construction pattern, and framework levels. It is possible for a chunk at any level of abstraction to display different granularity.

The application of the contextual approach to the representation of method knowledge has the important effect of making chunks modular. A context is defined as a pair, <situation, decision>. In other words, a method chunk is cohesive because it tells us the situation in which it is relevant and the decision that can be made in this situation. It is loosely coupled to other chunks because it can be used by the method engineer in the appropriate situation (created as a result of another method module) to satisfy his/her intention. Thus, the linear arrangement of method modules is replaced by a more dynamic one.

The *method meta-knowledge* level seeks to capture, in the method base, the situational and intentional knowledge associated with a method chunk. The contextual model of NATURE is an elegant model for representing this meta-knowledge. Since a context is defined as a pair, <situation, decision>, the contextual representation of meta-knowledge naturally captures these two kinds of meta-knowledge. Thus, modularity is extended to the meta-knowledge representation and meta-knowledge modules are tightly coupled to their peer method modules.

In the rest of this paper, we develop in detail the method knowledge and method meta-knowledge levels. Section 2 which deals with the former, presents and exemplifies the different levels of granularity and different abstraction levels which we believe, are relevant for method knowledge representation. Section 3 covers the meta-knowledge representation and illustrates through examples of queries, how method chunks can be retrieved from the method base. In section 4 we draw some conclusions and perspectives of work.

2 THE METHOD KNOWLEDGE LEVEL

Method knowledge is contained in *method chunks* represented in a uniform way but related to different methods and expressed with different granularity, at various levels of abstraction. Examples of such method chunks are (1) the OMT methodology, (2) the ER modelling approach, (3) the rules to define the key of an Entity-Type, (4) a generic outline providing a stepwise organisation of system analysis and (5) a generic set of guidelines for any concept description. The granularity is larger in (1) and (2) than in (3) above. Further, the first three are expressed at a less abstract level than the fourth and fifth examples above.

Notice that depending on its level of abstraction the method chunk will be reused as such (perhaps after some customisation) or will be instantiated before being assembled in the method under construction. Chunks of examples (1) to (3) above are directly reusable whereas each generic activity of the outline (example (4)) has to be instantiated according to the specific product of the method in hand before being used. Similarly the generic guidelines for concept description (example (5)) requires instantiation for each particular concept of the method under construction.

The examples also show that method chunks come from various methods which can have different purposes. The guidelines for an ER approach and of the OMT methodology are system engineering methods whereas the generic guidelines for concept description are part of a meta-method, i.e. a method to support the construction of methods. Our current method base which is implemented in the MENTOR environment (SiSaid, 1996) includes six traditional requirements engineering methods, namely OMT, E/R, OOA, SA/SD, O* and OOD (Plihon , 1994), the From Fuzzy to Formal method developed within the large ESPRIT project F3 (Bubenko, 1994), one meta-method, the NATURE meta-method (Plihon , 1995) and one method for method improvement, namely the learning based way-of-working to support NATURE method improvement (Prat, 1995). In short, we are concerned with engineering and re-engineering methods whose products are either computer based applications or methods themselves.

Therefore, there are two key aspects :

- a uniform representation of all types of chunks of the method base. For this, we use the NATURE modelling formalism which is based on the notions of *context, tree* and *forest.* All chunks are represented following this formalism as hierarchies of contexts called *trees*. A method is represented as a collection of trees that we refer to as a *forest.*
- a chunk classification. Chunks are classified into *component, pattern* and *framework* depending on their level of abstraction. We deal with these two aspects in the following.

2.1 Overview of method knowledge organisation

Figure 1 shows the structure of method knowledge in the method base using some binary ER-like notations. A large box represents an Entity-Type (ET) and a small box represents a binary Relationship-Type (RT) between two entity-types. The arrow head indicates the direction in which the label of the relationship-type holds. Cardinalities are also shown. For example, "Tree" and "Forest" are entity-types and are related through the "composed of" relationship-type. The direction of the RT says, "Forest composed of trees".
The notations also include the notion of an "objectified relationship-type" (Tempora, 1994). This notion is an abstraction mechanism which allows a relationship-type to be viewed, at a higher level of abstraction, as an entity-type. This applies for example, to the RT between a "Situation" and a "Decision" which is viewed as the entity-type "Context" to enable it to enter into a RT with the ET "Tree".

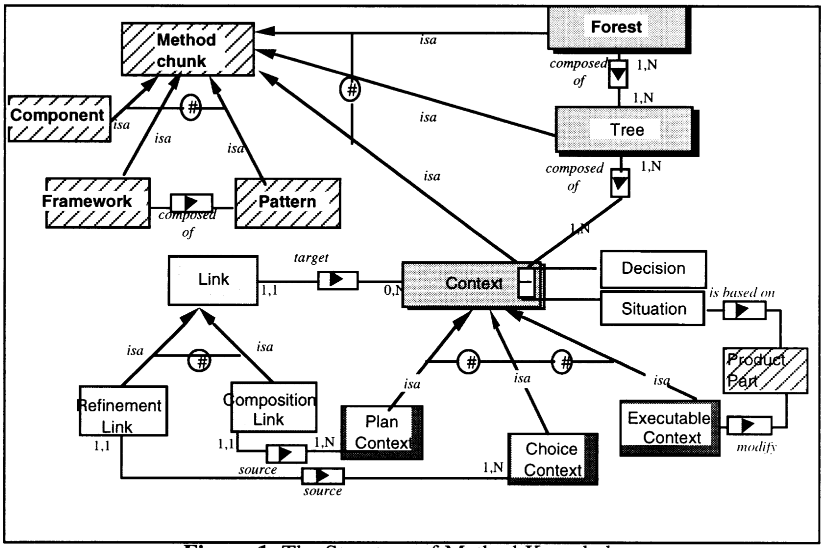


Figure 1 The Structure of Method Knowledge.

According to their type, method chunks are classified into *component*, *pattern* or *framework*. The entire OMT method description is an example of component, a set of generic guidelines for concept identification is an example of pattern, the NATURE meta-method (the method to guide the construction of any NATURE way-of-working) is an example of framework. This classification is borrowed from the Object Oriented area and shall be made precise in the next section.

According to their granularity, method chunks are classified into *forests*, *trees* or *contexts*. Within the OMT method, a single piece of knowledge such as the description of a Class in the Dictionary is modelled as a context. This context couples the decision Describe_Class_into_Dictionary to the situation a Class_has_ been_created. The set of guidelines to Identify_ a_Class is a more complex chunk of knowledge which can be modelled as a tree to compose the context for Identifying_the_Class and the one for Validating_The_Class. The OMT methodology itself can be represented as a forest of trees, with a tree for each model, namely the Object Model, the Dynamic Model and the Functional Model.

Figure 1 shows that these two classifications of method chunks are orthogonal. They constitute two ways of clustering the method base elements, each method chunk being represented in each of the two clusters. Besides, each cluster is a partition. Thus, a component is neither a pattern nor a framework; a forest is neither a tree nor a context. In the following, we consider the two orthogonal classifications in detail. We first consider the issue of uniform representation of chunks and thereafter their typology.

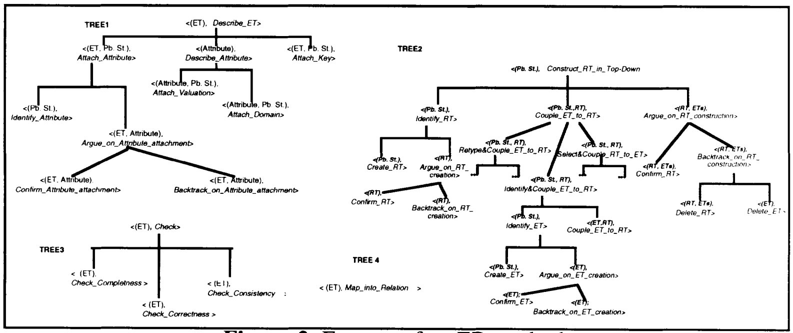
2.2 Chunk representation

The modelling formalism used to represent method knowledge has been designed with a certain view of engineering process support in mind. We believe that methods which aim at supporting engineering processes must be *contextual*. At any moment, each application or method engineer is in a subjectively perceived *situation* which is looked upon with some specific *intention of decision* in mind. The NATURE modelling formalism (Rolland, 1994), (Rolland, 1995) makes the notions of situation, decision, as well as *context* explicit.

As shown in Figure 1 the notion of *context* constitutes the basic building block of our method modelling approach. Contexts can be linked repeatedly in a hierarchical manner to define *trees*. A tree is composed of *contexts* and *links*. As shown in Figure 1, links are of two kinds : *refinement links* which allow the refinement of a large-grained context into finer ones and *composition links* for the decomposition of a context into component contexts.

A *method* is represented as a collection of hierarchies of contexts that we refer to as a *forest*. This reflects our view of methodological support being based on disconnected prescriptions, i.e. context trees which are not linearly sequenced but which can be dynamically combined according to the situation at hand.

Contexts, trees and forests are the three kinds of method chunks stored in the method base. They might be looked upon as structured modules of knowledge for supporting decision making in engineering processes. Figure 2 presents a partial method for defining an ER diagram made of a forest composed of four trees. Each tree is related to a specific issue : Entity-Type (ET) description (tree 1), Relationship-Type (RT) construction (tree 2), ET checking (tree 3) and ET mapping (tree 4). They will be progressively explained in the remaining of this section.



TREE1
<(ET), Describe_ET>
<(ET, Pb. St.), Attach_Attribute>
<(Attribute), Describe_Attribute>
<(ET, Pb. St.), Attach_Key>
<(Attribute, Pb. St.), Attach_Valuation>
<(Attribute, Pb. St.), Attach_Domain>
<(Pb. St.), Identify_Attribute>
<(ET, Attribute), Argue_on_Attribute_attachment>
<(ET, Attribute), Confirm_Attribute_attachment>
<(ET, Attribute), Backtrack_on_Attribute_attachment>
TREE3
<(ET), Check>
< (ET), Check_Completness >
< (ET), Check_Consistency >
< (ET), Check_Correctness >
TREE2
<(Pb. St.), Construct_RT_in_Top-Down
<(Pb. St.), Identify_RT>
<(Pb. St.,RT), Couple_ET_to_RT>
<(RT, ETs), Argue_on_RT_construction>
<(Pb. St.), Create_RT>
<(RT), Argue_on_RT_creation>
<(Pb. St., RT), Retype&Couple_ET_to_RT>
<(Pb. St., RT), Select&Couple_RT_to_ET>
<(RT, ETs), Confirm_RT>
<(RT, ETs), Backtrack_on_RT_construction>
<(RT), Confirm_RT>
<(RT), Backtrack_on_RT_creation>
<(Pb. St., RT), Identify&Couple_ET_to_RT>
<(RT, ETs), Delete_RT>
<(ET), Delete_ET>
<(Pb. St.), Identify_ET>
<(ET,RT), Couple_ET_to_RT>
<(Pb. St.), Create_ET>
<(ET), Argue_on_ET_creation>
<(ET); Confirm_ET>
<(ET); Backtrack_on_ET_creation>
TREE 4
< (ET), Map_into_Relation >

Figure 2 Excerpt of an ER method.

Contexts are defined as couples <situation, decision>, where the *decision* part represents the choice an engineer can make at a moment in the engineering process and the *situation* is defined as the part of the product it makes sense to make a decision on. Notice that our notion of method chunk strongly couples the process part to the product part. Details about this coupling and the product meta-model can be found in (Schmitt, 1993) and (Schmitt, 1995). A decision corresponds to an intention, a goal that the engineer wants to fulfil. Finally contexts can be of three *types*, namely executable, choice or plan. Each type of context plays a specific role in a tree. We now consider each of these types of contexts in turn.

Executable Context

An *executable context* corresponds to a decision which is directly applicable through actions which induce a transformation of the product under development. In tree2 of Figure 2, the context<*(Pb. St.), Create-RT*> is an executable context, (Pb. St. is the abbreviation of Problem Statement). The intention to *Create a Relationship-Type* is immediately applicable through the performance of an action for creating a new Relationship-Type (RT) in the specification under development. Executable contexts are the atomic blocks of our methods. They are often the leaves of trees. Non atomic contexts are built over contexts using refinement or composition links.

Choice Context

When building a product, an engineer may have several alternative ways to solve an issue. For this purpose we introduce the second type of context, namely the *choice context.* The execution of such a context consists of choosing one of its alternatives, i.e. selecting a context representing a particular strategy for the resolution of the issue raised by the context. For example in Figure 2, the context *<(RT, ETs); Argue-on-RT-construction>* is a Choice context introducing two alternatives to a Relationship-Type construction, namely to confirm its creation (*<(RT, ETs); Confirm-RT>*) or to withdraw it (*<(RT, ETs); Backtrack-on-RT-construction>*). Arguments are defined to support or object to the various alternatives of a choice context. For example, the *backtracking-on-RT-construction* is the right decision to make either when the relationship between entities is not sensible or when it is not relevant. Description arguments play an important role in the process model. They help in capturing heuristics followed by the application engineer in choosing the appropriate problem solving strategy.
Finally, it is important to notice that the alternatives of a choice context are contexts too. In the previous example, the first alternative is an executable context and the second one is a plan context. But they could be choice contexts introducing what is referred to as a refinement link between contexts.
As illustrated in Figure 2, we use a graphical notation for contexts and trees. For the sake of conciseness, we will also use a textual notation which, in the case of choice contexts, is based on the OR logical connector (denoted ”) between alternatives (denoted alt_i). Thus, the textual notation for a choice context (CC) is : $CC = alt_1$ ”alt_2 ”... ” alt_n. For instance, the textual notation for the *<(RT, ETs), Argue-on-RT-construction>* context is *<(RT, ETs), Confirm-RT>* ” *<(RT, ETs), Backtrack-on-RT-construction>.*

Plan Context

In order to represent situations requiring a set of decisions to be made for fulfilling a certain intention (for instance to *Construct a RT* in the ER methodology) the modelling formalism includes a third type of context called the plan context. A plan context can be looked upon as a macro issue which is decomposed into sub-issues, each of which corresponds to a sub decision associated to a component situation of the macro one. Components of a plan context are also contexts but related through a *composition link.*
In Figure 2, the context *<(Pb. St.), Construct-RT>* is a plan context composed of three component contexts, namely *<(Pb. St.), Identify-RT>*, *<(Pb. St., RT), Couple-ET-to-RT>*, *<(RT, ETs), Argue-on-RT-construction>*. This means that, when constructing an RT, the method engineer has first to identify the RT, then couple it to all the ETs participating in it, and finally, argue on the construction of the RT. Component contexts of a plan context can be organised in a sequential, iterative and/or parallel manner.
In the textual notation of plan contexts, the sequence is represented by a "•", iteration is denoted by "*" and parallelism by the shuffle symbol "⧢". Therefore, the textual representation of the plan depicted in Figure 3 is *<(Pb. St.), Construct-RT>* = *<(Pb. St.), Identify-RT>* • *<(Pb. St., RT), Couple-ET-To-RT>** • *<(RT, ETs), Argue-on-RT-construction>.*

It can be seen that the uniform representation looks upon chunks as modules. These modules are cohesive : they are contextual and can be used in specific situations to carry out specific intentions. This ensures a tight coupling between the product and process aspects of methods and a chunk represents both these aspects in a unified way. Besides being cohesive, the uniform representation provides loose coupling through the graft (Schwer, 1995) operation. By a use of this operation it is possible to break away from the strict, artificial linearity of methods and couple chunks together more dynamically.

2.3 Chunk typology

Looking to method chunks as reusable elements leads us to classify them into three categories namely, *components, patterns* and *frameworks.*

Early method engineering approaches (Welke, 1991), (Harmsen, 1994) assume that method construction is an assembly process of methods fragments. These method fragments are method specific and can be product or process parts of existing methodologies. One can draw an analogy between such method fragments and reusable classes in object oriented approaches. We refer to these as *method components* or simply *components.*

However, our belief is in the existence of a corpus of both, generic *method knowledge* and generic *method construction knowledge* which has not been looked after, identified and described yet. Our proposal is to develop a *domain analysis* approach to identify objects, rules and constraints which are

(a) common among different (but similar) methods

(b) common among different (but similar) ways of method construction

and to formalize them as method chunks.

In this way, method engineering can use the results of method domain analysis and save a significant amount of time as demonstrated in other domains (Arango, 1989). If we assume that the degree of similarity which exists in the construction of methods which belong to the same area is similar to the equivalent degree in system requirements engineering, then method domain analysis can result in significant overall productivity improvement in method construction. Indeed, Jones (Jones, 1984) indicates that only 15% of the requirements for a new system are unique to the system; the remaining 85% comes from the requirements of existing similar ones.

We introduce the notion of *framework* to model commonalties among methods and the notion of a *method construction pattern, pattern* for short, to capture generic laws governing the construction of different but similar methods. A framework is a method chunk which formalizes, in a more abstract way than a component does, knowledge which is common among several methods. A pattern models a common behaviour in method construction. It is generic in the sense that it is used by a typical method engineer in every method construction process. It is more abstract than a component or a framework. Both terms have been chosen by analogy to reuse approaches in the object oriented area. Patterns are there defined as solutions to generic problems which arise in many applications (Gamma, 1993), (Pree, 1995) whereas a framework is application domain dependent (Wirfs-Brock, 1990), (Johnson, 1988).

All examples provided in the previous paragraph belong to the class of method components. In the following, we exemplify the notions of framework and pattern within the NATURE meta-method.

Method Construction Pattern

There can be many different kinds of construction patterns, like the *Identify, Describe, Construct* and *Define patterns.* A more detailed presentation of these patterns can be found in (Rolland, 1996). These patterns already form a part of our method base. In addition, we are in the process of defining additional patterns for *checking* and *refinement.* In this section, we illustrate the notion of a pattern through the *Describe* pattern. Its genericity is brought out by applying it to OMT and ER approaches.

We carried out a domain analysis and tried to identify common patterns of behaviour that method engineers shall exhibit when instantiating the concepts of the NATURE modelling formalism in order to construct methods. This leads us to two major conclusions :

- there exist generic laws underlying method construction; these laws are generic in the sense that they can be applied to the construction of many methods. Using these laws, we generate for instance, six of the traditional analysis methodologies, OMT, OOA, SA/SD, ER, O* and OOD. These laws can be encapsulated in method construction patterns and made available in a library, the method base. Patterns are method chunks and therefore, like any other method chunk, can be expressed using the NATURE modelling formalism : patterns are trees of contexts.

- the structure of the product generated by the method is a key constructional factor and, therefore, the main parameter of the generic laws. We have calculated, for example, that varying the typology of concepts used to represent the OM product structure of OMT can lead to 13000 different ways of engineering it. This demonstrates, in some way, the genericity of the patterns and partly provides a measure of their effectiveness.

Let us take for instance, the example of any description of a concept in a schema - whatever the product under construction is (e.g. ER, OMT, etc.) - such a description follows the pattern shown Figure 3. The pattern identifies discriminant criteria which are, in this case, types of concepts. For instance, we make the distinction between *constructional concepts* - which participate to the structuration of the product - and *definitional concepts* - which only contribute to the definition of the constructional concepts (Prakash , 1994). For instance, in an ER model, an Entity-Type (ET) is an example of a constructional concept whereas a Domain is a definitional concept. Definitional concepts are further refined into *Properties, Prop*, *Constraint,* Const, or *Cd-Concept,* Cd (concept for the definition of another concept). In an ER model, the Valuation of an Attribute is a Property, the Key of an Entity-Type is a Constraint and the Attribute could be regarded as a definitional concept participating in the description of the constructional concept Entity-Type. Besides, from the point of view of their granularity, concepts are classified into *atomic concepts* - they are stand-alone concepts and *compound concepts* - they are built upon other concepts. In an ER model, a Relationship-Type will be a compound concept while a Domain is an atomic concept.

It is important to notice that the classification of concepts of a given model (let say the ER model) is not unique but specific to each method based on this model. For instance, a particular ER based method can consider the concept Attribute as a Cd-concept participating in the definition of Entity-types whereas it could be a constructional concept in another method. Therefore, different topologies of concepts can be derived for the same model from the above classification.

<(C), *Describe_C*> = ⧢ (<(C, Pb.St.), *Attach_Cd*>* • <(Cd), *Describe_Cd*>* ,
<(C, Pb.St.), *Attach_Prop*>*) • <(C, Pb.St.), *Attach_Const*>*)
<(C, Pb.St.), *Attach_Cd*> = <(Pb.St.), *Identify_Cd*> • <(C, Cd), *Argue_on_Cd_attachment*>
∪ <(Pb.St.), *Identify_Cd*>
<(C, Cd), *Argue_on_Cd_attachment*> = <(C, Cd), *Confirm_Cd_attachment*> ∪
<(C, Cd), *Backtrack_on_Cd_attachment*>
⧢ : shuffle • : sequence ∪ : alternative

Figure 3 The Pattern for concept description.

The pattern in Figure 3 is a plan-context stating that the description of a concept C requires the attachment of all its definitional concepts (<(C, Pb.St.), *Attach_Cd*>*) followed by their description (<(Cd), *Describe_Cd*>*), the attachment of its properties (<(C, Pb.St.), *Attach_Prop*>*), and constraints (<(C, Pb.St.), *Attach_Const*>*). The shuffle symbol indicates that the attachment of properties and definitional concepts can be made in any order.

The generic method construction pattern is based on the recursive description of a concept (since Cd "is-a" C, Describe_Cd "is_a" Describe_C). This allows us to deal with compound definitional concepts which have themselves to be described. Attachment may be with

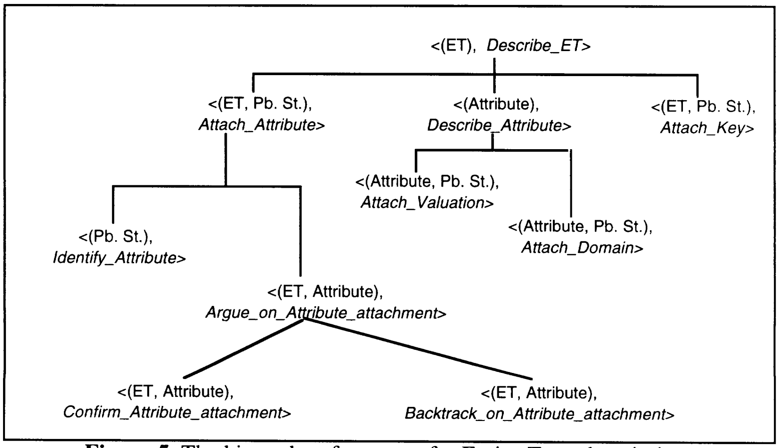
(<(Pb.St.), *Identify_Cd>* • <(C, Cd), *Argue_on_ Cd_ attachment>*) or without argumentation (<(Pb.St.), *Identify_Cd>*).

When applying the pattern to each concept of the model in use, the method engineer has first, to instantiate Cd, Prop and Const for this concept and secondly to choose an order for the various attachments or to leave open the option of their intertwining. This results in a plan-context which is the methodological guideline for describing a concept. The previous actions must be repeated for each component context of the plan referring to a compound definitional concept . This will result in a tree structure as a methodological guideline. Figure 4 gives the tree generated by the application of this pattern to the concept of Entity-Type.

<(ET), *Describe_ET>* = (<(ET, Pb.St.), *Attach_Attribute>** • <(Attribute), *Describe_Attribute>**) • <(ET, Pb.St.), *Attach_Key>*
<(ET, Pb.St.), *Attach_Attribute>* = <(Pb. St.), *Identify_Attribute>* • <(ET, Attribute), *Argue_on_ Attribute_attachment>*
<(ET, Attribute), *Argue_on_Attribute_attachment>* = <(ET, Attribute), *Confirm_Attribute _attachment>* ∪ <(ET, Attribute), *Backtrack_on_Attribute_attachment>*
<(Attribute), *Describe_Attribute>* = <(Attribute, Pb.St.), *Attach_Valuation>* • <(Attribute, Pb.St.), *Attach_Domain>*

Figure 4 Instantiating the *Describe* pattern on the concept of Entity-Type.

The corresponding graphical representation is shown in Figure 5. It corresponds also to tree 1 in Figure2. The method engineer has chosen to sequentially order the attachment of Attributes to Entity-types, their description and the Key constraint definition. This corresponds to the first instantiation of the *Describe* pattern. He/she chose to argue on the attachment of Attributes to Entity-Types but not on the one of the Key. Finally, since Attribute is a compound definitional concept, the *Describe* pattern has to be applied a second time to decide in which way attributes will be described. Assume, there are two properties describing the Attribute concept namely, Valuation and Domain and no constraint. The <(Attribute), *Describe_Attribute>* context is then a plan-context with two executable component contexts, <(Attribute, Pb. St.), *Attach_ Valuation>* and <(Attribute, Pb.St.), *Attach_Domain>* .


<(ET), Describe_ET>
<(ET, Pb. St.), Attach_Attribute>
<(Attribute), Describe_Attribute>
<(ET, Pb. St.), Attach_Key>
<(Attribute, Pb. St.), Attach_Valuation>
<(Attribute, Pb. St.), Attach_Domain>
<(Pb. St.), Identify_Attribute>
<(ET, Attribute), Argue_on_Attribute_attachment>
<(ET, Attribute), Confirm_Attribute_attachment>
<(ET, Attribute), Backtrack_on_Attribute_attachment>
Figure 5 The hierarchy of contexts for Entity-Type description.

For constructing an ER method, the *Describe* pattern will be applied to every constructional concept of the model e.g. Entity-Type and Relationship-Type. But it can be applied in a similar way to generate description guidelines for any other concept in any other method. As an illustration, Figure 6 shows, in a textual form, the tree generated for the description of the concept Association of the OMT methodology.

<(Association), *Describe_*Association> =
    <(Association, Pb.St, *Attach_Attribute_to-Association*>* • <(Attribute), *Describe_Attribute*>*,
    • <(Association, Pb.St), *Attach_Multiplicity_to_Association*>*

<(Association, Pb.St), *Attach_Attribute_to_Association*> =
    (<(Pb.St), *Identifiy_Attribute*>
        • <(Association, Attribute), *Argue_on_Attachment_of_Attribute_to_Association*>)

<(Attribute), *Describe_Attribute*> =
    (<(Attribute, Pb.St), *Attach_Domain_to_Attribute*>
    • <(Attribute, Pb.St), *Attach_Valuation_to_Attribute*>

<(Association, Attribute), *Argue_on_Attachment_of_Attribute_to_Association*> =
    <(Association, Attribute), *Confim_Attachment_of_Attribute_to_Association*>
    " <(Association, Attribute), *Backtrack_on_Attribute_Attachment*>

Figure 6 Instantiating the *Describe* pattern on the concept of Association.

An association is a constructional concept with two definitional concepts: its attributes and its multiplicity. Therefore, the description of an association is guided by a plan consisting of the attachment of attributes to the association (<(Association, Pb.St, *Attach_Attribute_to_Association*>*), their description (<(Attribute), *Describe_ Attribute*>*) followed by the attachment of the multiplicity constraint related to each of its roles (<(Association, Pb.St), *Attach_Multiplicity_to_Association*>*). Notice that the shuffle has no influence since an association does not have properties and has only one Cd-concept, attribute. Consequently there is only one possible ordering of contexts in the plan.
The importance of patterns lies in their genericity, in their ability to be used systematically in constructing a large number of methods. Chunks which are patterns make available this genericity to method engineers.

Method framework

Frameworks have already proved their efficiency in software design (Johnson, 1991), (Johnson, 1988), (Wirfs-Brock, 1990) and particularly in interface development (Krasner, 1988), (Wilson, 1991), (Weinand, 1989). The design and improvement of frameworks (Wirfs-Brock, 1990), (Johnson, 1991) have been studied and approaches developed. In information systems design, a number of conceptual frameworks have been proposed (Olle , 1988), (Pohl , 1993), (Krogstie , 1995a) and even merged (Krogstie , 1995b). We believe that method frameworks can play a role in method construction and should be integrated in method bases. The way we understand the concept of framework is close to the notion of outline in (Harmsen, 1994) and road map. Let us take, as an example, the NATURE meta-method.

In the NATURE project we initially addressed the problem of constructing methods by providing a process meta-model (the NATURE formalism presented in the previous section), under which methods can be generated by instantiating the meta-model. This provides a way by which method engineers can define, in a systematic manner, the desired methods. However, this does not obviate the need for a prescribed method for the method engineer, the meta-method, which could be followed for method construction.

When developing the meta-method, it was possible to develop yet another formalism for representing it. However, by extending the NATURE contextual approach to the meta-method, it was possible to represent it in contextual terms, apply the modelling formalism to the meta-method, and treat it as just any other method. The NATURE method, once prescribed, can be used within the MENTOR environment which provides guidance to any process which is in accordance with it. Therefore, since the meta-method is just another process, we can extend the full guidance capability to it.

The meta-method is encapsulated in a framework depicted in Figure 7. We organise the construction of a specific method as a plan composed of three components to respectively, find the basic blocks of the method under construction (<(Method Statements), *Find_Basic_Blocks*>) then, to assemble the basic blocks into trees (<(Product Structure, Basic Blocks), *Build _Trees*>) and, finally, to describe each context of the forest in detail (<(Contexts), *Specify_Forest*>).

The generation of basic blocks assumes the existence of the product structure for the method under construction as well as of the relevant types of decision for decision making. Therefore, the meta-method suggests a plan where the constructional factors are defined first (<(Method Statements), *Build_Product_Structure*>), then the decision types are identified <(Method Statements), *Identify_Types_ of_Decision*> and finally, the basic blocks are generated (<(Product Structure), *Generate_ Basic_Blocks*>).

The gathering of basic blocks in trees asks for a definition of the approaches. Therefore, the context <(Product Structure, Basic Blocks), *Build_Trees*> is further defined in the meta-method as a plan with two components, to choose the approaches <(Product Structure), *Choose_Approaches*>, and to achieve the gathering of basic blocks to generate the method trees <(Product Structure, Basic Blocks), *Generate_Trees*>.

The formal specification of the method consists of repeating the component (<(Context), *Specify_Context*>) to specify contexts.

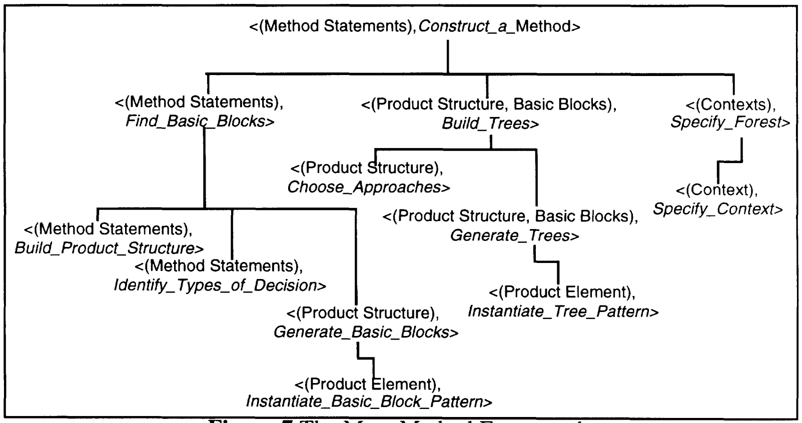

<(Method Statements), *Construct_a_*Method>
<(Method Statements), *Find_Basic_Blocks*>
<(Product Structure, Basic Blocks), *Build_Trees*>
<(Contexts), *Specify_Forest*>
<(Product Structure), *Choose_Approaches*>
<(Context), *Specify_Context*>
<(Method Statements), *Build_Product_Structure*>
<(Product Structure, Basic Blocks), *Generate_Trees*>
<(Method Statements), *Identify_Types_of_Decision*>
<(Product Element), *Instantiate_Tree_Pattern*>
<(Product Structure), *Generate_Basic_Blocks*>
<(Product Element), *Instantiate_Basic_Block_Pattern*>

Figure 7 The Meta-Method Framework.

Frameworks often use patterns (Krasner, 1988) (this justified the composed-of relationship in Figure 1). As shown in Figure 7 our proposal is to support the generation of basic blocks by method construction patterns. Therefore, the application of the framework calls for the instantiation of the appropriate patterns. Generic patterns are classified into two : those which build basic blocks for each product element of the product structure (<(Product Element), *Instantiate_Basic_Block_Pattern*>) and those which relate these blocks together (<*(*Product Element*), Instantiate_Tree_Pattern*>).

3 THE METHOD META-KNOWLEDGE LEVEL

Assuming that method chunks for components, patterns and frameworks exist, the question now is "how to deliver the relevant method chunks to the user?" We shall look at this question in two ways : first, by describing the semantic contents of the method base in a manner which eases the retrieval of chunks meeting the requirements of the user and, secondly, by providing query facilities.

Description of method knowledge is knowledge about method knowledge i.e. *method meta-knowledge*. We use the notion of *descriptor* (De Antonellis, 1991) as a means to describe method chunks. A descriptor plays for a method chunk the same role as a meta-class does for a class. The concept of descriptor is similar to the one of faceted classification schema (Prieto-Diaz, 1987) developed in the context of software reuse.

The knowledge that should be provided by the descriptor aims at facilitating the use of the method base. If we keep in mind that the knowledge base shall facilitate the construction of a method suitable for a specific project, then the descriptor must help in categorizing the situation in which a method chunk is relevant for a certain purpose. The method engineer, as a user of the method base, is faced to a certain *situation* which he/she looks upon with a certain *intention* in mind. He/she is placed in a certain *context* that he/she should be able to formulate as a query to the method base. Consequently, the descriptor shall also be organised in a contextual fashion. This means that a descriptor must categorize the situation in which the chunk can be used and describe the intention of its use.

Thus, we propose to extend our contextual approach to the representation of method meta-knowledge. A descriptor can be seen as a *meta-context* which links the situation in which a method chunk is relevant to the intention the chunk allows to fulfil. The situation part refers to the characteristics of the projects in the development of which the chunk can be used as part of the project method. The intention part refers to the engineering intention(s) that could be fulfilled when using the chunk.

A descriptor shall naturally be associated to every method chunk, irrespective of its granularity. However gathering of chunks into other chunks such as forests might be justified because these chunks have a unique context of use and, therefore, a unique descriptor. We consider it is the role of the method base administrator to reorganize the initial method chunks in the light of their description. This justifies a classification of forests (see 3.1) into *methods* (forests which represent methods) and *groups* (forests which gather trees having the same use). Similarly, for facilitating the use of the method base, the administrator can build hierarchies of descriptors associated with hierarchies of forests. This shall permit a hierarchical search in the method base by refining progressively the characterisation of the project situation and/or the method engineering intention.

In the next section we consider the method meta-knowledge in detail. In the subsequent section we shall illustrate its use by introducing the query language through examples.

3.1 Method meta-knowledge organisation

Figure 8 depicts the way the meta-knowledge is organized using the same ER like notations that have been used for the knowledge part in Figure 1. For the sake of readability, the representation of the knowledge part has been restricted to the elements required to understand the links with the meta-knowledge part.

As introduced before, a method chunk is placed and described in the context of its potential use in specific projects. A method chunk is said to be relevant *for* (relationship-type *for* in Figure 8) a certain *situation* (entity-type *situation*) *to* (relationship-type *to*) achieve a certain *intention* (entity-type *intention*). For example, Chunk c is applicable *for* high risk project (characterization of the situation) *to* a reengineering purpose (intention).

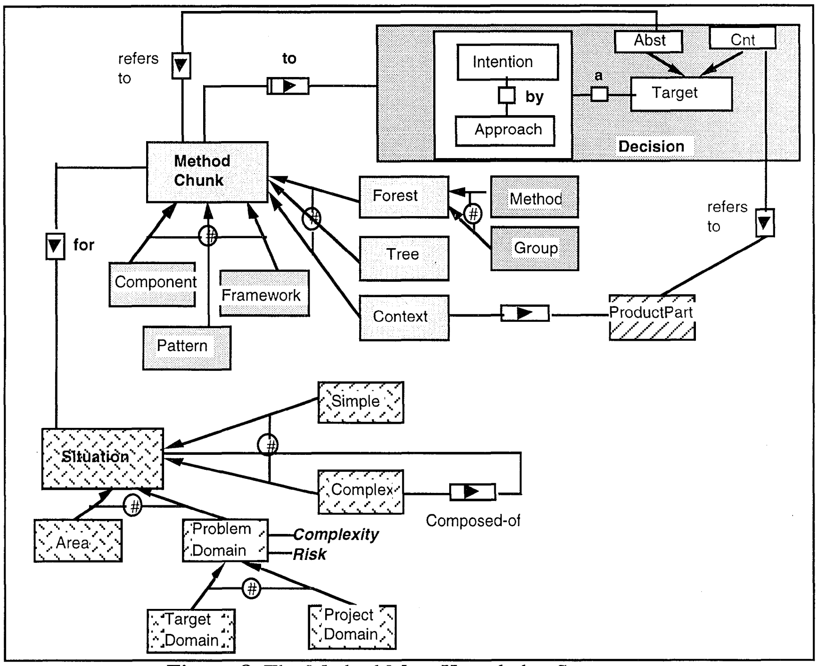


Figure 8 The Method Meta Knowledge Structure.

The NATURE contextual modelling approach refines the concept of *Decision* into *Intention* and *Approach* (Rolland, 1995). The intention is the goal to be achieved, the approach is the way to reach the goal. Further a Decision has a *Target* which is expressed in terms of *Product Parts.* As shown in Figure 8, all these aspects are relevant in a descriptor as they are discriminant criteria for the selection of method chunks. The intentional part of the descriptor can then express that a chunk is relevant *to* (relationship-type *to* in Figure 8) achieve *a* (relationship-type *a* in Figure 8) targeted intention *by* a given approach. Therefore the previous example can be refined in the following way : Chunk c is applicable *for* high risk project *to* reengineer (intention) *a* business process (target) *by* domain based approach (approach).

Targets in descriptors are either abstract targets -which refer to type level i.e. to one of the types of method chunks- or concrete targets -which refer to instances of method chunks. This is modelled in Figure 8 by sub-typing *Target* into two, *Concrete Target (Cnt)* and *Abstract Target (Abst)* A concrete target refers to a Product Part whereas an abstract target refers to a Method Chunk (relationship-types *refers to* in Figure 8). For example, the descriptor of the forest (a group) composed of the collection of constructional patterns such as *Describe, Identify, Define etc.* has generate as *intention* and an *abstract target* which is a tree. On the other hand, the descriptor of the OMT forest (a method) describes the *intention* of the forest as requirements engineering and the *concrete target* as an analysis schema. This classification is helpful to

formulate recursive queries whose outputs are abstract targets i.e. method chunks that can, in turn, be subjects of embedded queries. This shall be exemplified in the following section.
The situation part of a descriptor aims at providing the means to evaluate the adequacy of the method chunk to the situation of project at hand while the intention part tries to ensure that the goal of the project matches the goal of the method chunk. As shown in Figure 8 we propose to characterise the situation in two ways : (a) by the *Area* (entity-type *Area*) of the project and, (b) by the *complexity* and *risk* (properties of the entity-type *Problem Domain)* as two situational factors evaluated for both the *Project Domain* (entity-type *Project Domain*) and the *Target Domain* (entity-type *Target Domain*).

Our characterisation of the Problem Domain is based on the results achieved by the EUROMETHOD project (Franckson, 1994). The term Target Domain refers to the system to be engineered or re-engineered and its evaluation comprises two parts, one for the information system and another one for the computer system. Each of the two parts is further refined into detailed factors such as *size* of target domain, *heterogeneity of actors*, *complexity of data*, *complexity of target technology* etc. There are four aspects of the Project to evaluate : *tasks*, *structure*, *actors* and *technology*. Complexity measure has three values, simple (S), moderate (M) and complex (C); similarly the risk values are low (L), moderate (M) and high (H).

It is clear that all these situational factors cannot be evaluated for every descriptor; some of them may not even be meaningful for some descriptor. Thus, in order to leave the required level of flexibility and freedom to the method base administrator the *Situation* is specialized into *Simple Situation* (entity-type *Simple Situation* in Figure 8) and *Complex Situation* (entity-type *Complex Situation* in Figure 8) - a situation composed of situations. This allows us to describe a Context for <u>geopolitical game</u> *Area* and <u>C</u> *complexity of Target Domain.size* and <u>C</u> *complexity of Target Domain.heterogeneity of actors* and <u>M</u> *risk of Project.actors.*

A method chunk descriptor includes instances of elements introduced in Figure 8 as parts of the situation and decision which form its context of use. As a brief summing up of this section, we can point out that the NATURE contextual approach provides an elegant frame to model the meta-knowledge supporting situational method engineering. The key notion of a context is used here to describe the type of project, the problem domain situation, as well as the engineering purpose for which a method chunk can be used. Besides the same frame is applicable to concrete components (method fragments) and abstract patterns and frameworks. In the next section we exemplify the query mechanism to access the method base.

3.2 Using meta-knowledge to retrieve method knowledge

Access to method knowledge is through the meta-knowledge i.e. the descriptors. Based on the structure of the descriptor presented in the previous section, we propose a query language which is exemplified in this section. Navigation in this language is through the structure of a descriptor and is based on key words corresponding to entity-type names and relationship-type names.

In order to query the method base, the user has access to dictionaries which contain the values of the entity-types which are parts of the descriptor structure. He can access, for instance, the *intentions* which have chunks corresponding to them in the method base. Similarly, access can be obtained to chunks according to the known *areas*. Dictionaries are organized like thesaurus, in a hierarchical manner in which, for instance, the intention *engineering* is refined into *requirements engineering*, *design* and *implementation*. Requirements engineering could itself be refined into *requirements representation*, *specification* and *agreement*. Dictionaries are automatically updated in an interactive way at the time the method base administrator adds, removes or changes descriptors.

Below are some examples of queries focusing on the decision part of the descriptor. Key words are bold letters and comments are put into brackets in the query itself

```
Select forest with name (property of the ET) = 'OMT'

Select tree to specify(intention)
        a schema (target) with name = 'OM'

Select method chunk
to requirements engineering•representation.(sub-intention)
        by reuse (approach)
and to requirements engineering•agreement (sub-intention)
        by cooperation (approach)
```

With is used to introduce a selection based on an Entity-Type (ET) property. **To** is the name of the Relationship-Type (RT) which links the ET Method Chunk to the ET Decision (Figure 8). It is used for expressing a selection based on decision values. **By** and **a** play a similar role. The notation • is the dotted convention of query languages. In the last example it is used to manipulate sub-intentions (representation is a sub-intention of requirements engineering).

```
Select framework
        to engineer (intention)
        a method (target)
                to requirements engineering (target intention)
                by domain reuse (target approach)

Select pattern
        to generate (intention)
        a tree (target)
                to construct (target intention)
                a schema (sub-target)
```

These two queries aim at selecting abstract method chunks; they have abstract targets (method and tree) which are subjects of further selection just as any other method chunk can be. This leads to queries with an embedded form where a **to-a-by** clause is embedded in another **to-a-by** selection clause.

The remaining examples focus on selections based on the situation part of a descriptor. The queries could complement some of the previous ones.

```
Select method.
        for real time (area)

Select framework
        for business system (area)
                with risk-of project domain•structure = H
                and with risk-of project domain•actors = H

Select method
        for information system (area)
        with complexity-of target domain•size = C
        and with complexity-of target domain•technology= C
        and with risk-of project domain•tasks = H
```

The key word **for** is the name of the relationship-type which links the ET method Chunk to the ET Situation. It introduces the selection part based on the project and domain situation. The notation • allows the user to characterise the complexity and risk of both the project and domain using the subdivisions proposed.

The query language is currently under development in the MENTOR environment. It is being implemented on top of the O2 (O2, 1993) query language.

4 CONCLUSION

The essence of the contextual method engineering approach developed in this paper lies in the emphasis on decision making in specific situations in a rational manner. At the centre of this approach is the notion of a context, which is applied both to method knowledge and method meta-knowledge.

The contextual approach advocates a move from method fragments to method modules. These modules are both cohesive and loosely coupled. Cohesiveness results in modules which can be used in specific situations in specific ways whereas loose coupling makes it possible to move away from a rigid, linear modular sequence to more dynamic module interaction.

Method knowledge at different levels of abstraction and granularity is all expressed in the same representation, that of a chunk modelled with the NATURE contextual formalism. This can been gainfully exploited to enact any chunk, whatever its abstraction or granularity, thanks to the enactment mechanism of the MENTOR environment which guides any process modelled in the terms of this formalism.

Future work shall concentrate on the discovery of new patterns and the implementation of the query language.

5 REFERENCES

Arango G (1989), "*Domain analysis : from art to engineering discipline*", Proc. 5th Int. Workshop on Software Specification and Design, IEEE Computer Society Press, San Diego

Bubenko J., Rolland C., Loucopoulos P., DeAntonellis V. (1994), "*Facilitating "Fuzzy to Formal" Requirements Modelling*", IEEE 1st Int. Conference on Requirements Engineering, ICRE'94, pp 154-158

De Antonellis V., Pernici B., Samarati P. (1991) "*F-ORM METHOD : A methodology for reusing specifications*", in Object Oriented Approach in Information Systems, Van Assche F., Moulin B., Rolland C. (eds), North Holland

Franckson M. (1994), "*The Euromethod deliverable model and its contribution to the objectves of Euromethod*", Proc. IFIP-TC8 Int. Conf. on Methods and Tools for the Information Systems Life Cycle, Verrijn-Stuart and Olle (eds), North-Holland, pp131-149

Gamma E., Helm R., Johnson R., Vlissides J. (1993), "*Design patterns : Abstraction and Reuse of Object-Oriented Design*", Proc. of the ECOOP'93 Conf., Sringer Verlag

Harmsen F et al (1994), "*Situational method engineering for informational system project approaches*", in Method and Associated Tools for the Information Systems Life Cycle, Verrijn-Stuart and Olle (eds.), North Holland, pp169-194

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

Jones T.C. (1984), "*Reusability in programming : a survey of the state of the art*", IEEE Transactions on Software Engineering,SE Vol 10, No1

Johnson R. E., Foote B. (1988), "*Designing reusable classes*", Journal of Object-Oriented Programming, Vol 1, No3

Johnson R.E., Russo F. (1991), "*Reusing object-oriented design*", Technical report UIUCDCS 91-1696, May 1991, University of Illinois

Krasner G.E, Pope S.T (1988), "*A cookbook for using the Model-View Controller user interface in Smalltalk-80*", Journal of Object-Oriented Programming, Vol 1, No3

Krogstie J., Lindland O.I., Sindre G. (1995 a),"*Defining quality aspects for conceptual models*", in E.D. Falkenberg et al., editor, Information Systems Concepts, Proc. ISCO3, Marburg, Germany, North Holland

Krogstie J., Lindland O.I., Sindre G, (1995 b), *"Towards a Depeer Understanding of Quality in Requirements Engineering"* in Advanced Information Systems Engineering, CAISE'95, Iivari J. and Lyytinen K. (eds), Springer Verlag

Lyytinen K. (1987), *"Different perspectives on information systems : problems and solutions"*, ACM Computing Surveys, Vol 19, No1

O2 (1993) *"The O2 User Manual December"*,

Olle T. W., J. Hagelstein, I. MacDonald, C. Rolland, F. Van Assche, A. A. Verrijn-Stuart, (1988), *"Information Systems Methodologies : A Framework for Understanding"*, Addison Wesley

Plihon V. (1994), *"The OMT, The OOA, The SA/SD, The E/R, The O*, The OOD Methodology"* NATURE Deliverable DP2

Plihon V., Rolland C. (1995), *"Modelling Ways-of-Working"*, Proc 7th Int. Conf. on Advanced Information Systems Engineering, CAISE'95, Springer Verlag

Pohl K. (1993), *"The Three Dimensions of Requirement Engineering"*, 5th Int. Conf. on Advanced Information Systems Engineering, Paris, France, June 1993

Prakash N. (1994), *"A Process View of Methodologies"*, 6th Int. Conf. on Advanced Information Systems Engineering, CAISE'94, Springer Verlag

N. Prat (1995), *"Using learning techniques for process model improvement"*, Internal report, CRI (Centre de Recherche en Informatique), University of Paris-Sorbonne

Pree W. (1995), *"Design Patterns for Object-Oriented Software Development"*, Addison Wesley

Prieto-Diaz R., Freeman (1987), P., *"Classifying software for reusability"*, IEEE Software, Vol. 4, No. 1

Rolland C. (1994), *"A Contextual Approach to modeling the Requirements Engineering Process"*, SEKE'94, 6th International Conference on Software Engineering and Knowledge Engineering, Vilnius, Lithuania

Rolland C., Souveyet C., Moreno M. (Rolland, 1995), *"An Approach for Defining Ways-Of-Working"*, Information Systems, Vol 20, No4, pp337-359

Rolland C., Plihon V. (1996), *"Using generic chunks to generate process models fragments"* in Proc.of 2nd IEEE Int. Conf. on Requirements Engineering", ICRE'96, Colorado Spring

Tempora (1994), Tempora ESPRIT project : final report

Schmitt J.R. (1993), *"Product Modeling in Requirements Engineering Process Modeling"*, IFIP TC8 Int. Conf. on Information Systems Development Process, Prakash., Pernici and Rolland (eds) North Holland

Schmitt J.R. (1995), *"Méta-modélisation des démarches d'analyse"*, Phd thesis, University of Paris6 Jussieu

Schwer S., Rolland C. (1995), *"Theoretical formalization of the process meta-modelling approach"*, internal CRI report 95-08, University of Paris 1, France.

Si-Said S., Rolland C., Grosz G. (1996), *"MENTOR : A Computer Aided Requirements Engineering Environment"*, in Proc 8th Int. Conf. on Advanced Information Systems Engineering (CAISE'96), Springer Verlag .

Weinand A., Gamma E., Marty R. (1989), *"Design and implementation of ET++, a seamless oject-oriented applcation framework"*, Journal of Structured Programming, Vol 10, No2, pp63-87

Welke R, and Kumar K. (1991), *"Method engineering : a proposal for situation-specific methodology construction"*, in Systems Analysis and Design : A Research Agenda, Cotterman and Senn(eds), Wiley

Wilson D.A, Rosenstein L.S., Shafer D. (1991), *"Programming with MacApp"*, Addison-Wesley

Wirfs-Brock J., Johnson R. (1990), *"Surveying current research in Object-Oriented Design"*, Communications of ACM, Vol 33, No9

6 BIOGRAPHY

Colette Rolland is currently Professor of Computer Science in the Department of Mathematics and Informatics at the University of Paris-1 Panthéon/Sorbonne. Her research interests lie in the areas of information modelling, databases, temporal data modelling, object-oriented analysis and design, requirements engineering, design methodologies, development process modelling and CASE tools. She has extensive experience in participating to national and european research projects under the ESPRIT programme (projects TODOS, BUSINESS CLASS, F3, NATURE, TOOBIS, ELKD and CREWS) and conducting co-operative projects with industry. She is the French representative in IFIP TC8 on "Information Systems" and chairperson of the IFIP Working Group WG8.1.

Naveen Prakash is currently Professor of Computer Science and Dean of the Department of Information Technology in the Delhi Institute of Technology. His research interests are in object oriented analysis, methods, CASE and META-CASE tools and Method Engineering. He had an extensive professional experience in research and development as the R&D Director of CMC, one of the largest software houses in India. He is member of the IFIP Working Group WG8.1 and the co-ordinator of the group's activities in Asia.