SOFTWARE PROCESS AND ASSISTANCE MODELLING : THE CASE OF THE UPSSA TOOL

T. Khammaci

Institut de Recherche en Informatique de Nantes (IRIN)
3, Rue du Maréchal Joffre
44041 Nantes Cedex 01
France
Tel : (33) 40-30-60-56
Fax : (33) 40-30-60-53
e-mail : khammaci@iut-nantes.univ-nantes.fr

Abstract : *Recent works on the software process suggest that the software life cycle should be made through an enactable software process model. This model defines constraints to be respected by a software process during project development and initiatives to be taken by the environment. This paper presents UPSSA : a tool based on a software process model which is an abstraction of objects, tools, and policies. Thus, it encompasses an object model to describe project databases, an operator model to describe at abstract levels the tools' effects, and characteristics for expressing the policies. Design and implementation are also considered to ezhibit how a software process model described in our terms can be used for assisting intelligently the software developer by controlling the software process.*

Key-words : *Software Engineering, Software Process Modelling, Assistance Modelling, Object Model, Operator Model, Characteristics.*

1 Introduction

Currently, considerable attention is devoted in the field of software process modelling. The major efforts are described in [15] [8] [7]. Before presenting the concepts for software process and assistance, let us introduce definitions of some key terms used by the software process community that allow one to etablish a conceptual framework for process modelling.

A *model* is an abstract representation of reality that excludes much of the world's infinite detail [7]. This idea of the model means that detail which does not influence the relevant behavior of the model is eliminated. A *software process* consists of the set of activities performed during software development, including their scheduling and the object manipulated [5]. This idea of a software process means that there is one software process for each software system developed. A *software process model* is an abstract description of an actual or proposed process that represents selected process elements that are considered important to the purpose of the model and can be enacted by a human or a machine [7]. A language suitable for describing software process models is called a *software process modeling language* [5].

The software process model describes long-term activities which are complex and evolutive. It describes not only the quality of the software products, the usual tool and the policies to follow, but also how to help their users; this is why they are called models for assisted software process. In this framework, we classify the assistance into five kinds:

controlling, taking initiatives, helping and guidance, explaining and observing. (1) Controlling: depending on the current state of the development, an activity can or cannot be performed and when a user invokes an operation to perform an activity, then the environment controls whether the activity is allowed or not. (2) Taking initiatives: the environment may decide on its own to perform some operations without human interactions. (3) Helping and to guidance: a user can expect the environment to help and to guide him in his work by offering some facilities. (4) Explaining: explanation helps the user in understanding the software process at particular instants; for example, when the environment takes an initiative, it should explain why it decided to take it. (5) Observing: the software process model must give a user means to describe what, when and how to observe measurement and historical information about the software development process.

This paper is organized as follows. Section 2 outlines related work in software process and assistance modelling. Section 3 defines objectives and requirements for software process and assistance modelling. In section 4, we present the UPSSA (Using Pre-postconditions for Simulating Software Assistant) software process model. Using an activity example of software development, section 5 shows how we structure and formalize the knowledge necessary to interpret the UPSSA software process model. Section 6 outlines the architecture of the UPSSA tool we have developed to validate our proposals with regard to software process and assistance modelling. Finally, section 7 concludes with some perspectives.

2 Related work

Considerable work has been undertaken in the field of software and assistance modelling. Some of the first papers explicitly addressing software processes as a subject of interest are [16] [20]. Since then, several workshops [15] [8] and conferences [9] [21] have been devoted to software processes.

Curtis [7] has distinguished five approaches for representing software process information: (1) the programming models, (2) the functional models, (3) the plan-based models, (4) the Petri-net models and (5) the quantitative models. The key idea of the *programming models* approach [20] is a complete algorithm description of a software process by means of a formal language. This description is considered as a specification of how software process is to be managed in the software development environment by users and tools. Several on-going projects have been influenced by this idea, resulting in the construction of some experimented process software engineering environments as, for example, ARCADIA-APPL/A [26]. This process software environment has extended the ADA language with new capabilities for supporting software processes. The main drawback with this approach is that no algorithm of a particular software process can be completely predescribed in advance. In the *functional models* approach, which is illustrated by the Hierarchical and Functional Software Process [14], a software process is represented as a collection of process elements with input and output attributes. Specifically, a process is defined as a set of mathematical functions depicting relationships among inputs (such as specifications) and outputs (such as source codes). Furthermore, each of these functions is hierarchically decomposable into process subelements, where the input and output attributes of a parent process element must be satisfied by it's children attributes. This decomposition is continued until it produces process steps that can be mapped to

external tool invocation or manual operations. The main drawback with this approach is that it difficult to identify *a priori* the communications points between concurrent processes. The *plan-based models* approach [11] use planning paradigms that are emerged from artificial intelligence to design the software process. Thus, the GRAPPLE language constructs process models from two components: a set of process steps and a set of constraints on how those steps can be selected, ordered and applied. The main drawback with this approach is the integration of the multiple sources of knowledge representation and an analysis capability for deriving the properties of processes. In the *Petri-net models* approach [22], a software project is viewed as a structure of roles and their interactions. This structure is represented and executed by Petri-net language. Thus, role interaction nets aid the structured tasks that is those that can be planned from known dependencies. The main drawback with this approach is that it is difficult to use the model of coordination among role structures if an organization has not been able to establish a basic description of their process. Finally, the *quantitative models* approach [1] provides a quantitative representation of many behavioral observations of software projects.

Among several kinds of process languages that the software process community has been using to model the software process into process centered software engineering environments, Belkhatir [2] proposes three languages that are the most representative : the even-condition-action, the rule-based and procedural languages. In the *even-condition-action (trigger) language*, software processes are modeled by a set of event-condition-action rules which are interpreted by a trigger mechanisms tightly connected with a software database. ALF/MASP [3] is one of the few practical systems that has experimented with this approach. Other examples are: ARCADIA/APPL-A [26], DARWIN/LAW [18], TEMPO [2]. In ALF/MASP, triggers are used to control communication among parallel tasks by capturing changes on database objects. The tasks are modeled by pre- and post-conditions enveloping foreign tools and managed by a specialized expert system shell connected to PCTE+ [10]. APPL/A has extended ADA language with programmable trigger upon relations. The automation of the software process is done by these triggers. In DARWIN/LAW, an event happens when a message is sent from one object to another one, and control is passed to Darwin. Then, DARWIN enforces the appropriate law (a PROLOG rule set) which may intervene between message flow, possibly changing the message before it is delivered and/or rerouting it to a different destination or cancelling it. Unlike ARCADIA/APPLA and ALF/MASP, the TEMPO trigger can be attached to both entities and relationships to envelop methods. In the *rule-based language* the software process can be described by using logical declarations allowing one to specify what the user wants rather than a detailled specification of how the results are to be obtained [27]. Using this behavioral approach, various prototypes have been built, e.g., MARVEL [13], TEMPO [2]. MARVEL enacts the development process of a project by automatically firing user-specified rules that encapsulate development activities. Also, controlled automation in MARVEL is provided through opportunistic processing employed backward chaining and forward chaining. TEMPO uses rules in order to envelop method execution as well as to take initiatives when possible. In the *procedural language* approach, for example the APPL/A language [26] allows processes to be modeled in an extension of ADA that enables explicit representation of programmable, persistent relations. It supports multiple representational paradigms, e.g., procedural and declarative.

In another way, work in the software process focuses on the use of database modeling in the software life-cycle. Consequently, a number of research projects have been undertaken

[23] [17] [3]. Thus, a first approach relies on an object base and a rule base. The *object base* is the repository of all the information needed for the development of a software project. The *rules* specify the precondition of an activity and its postcondition. They are used as the basis for reasoning during the effective software development [13] [4]. Another approach provides predefined structures for the object and the knowledge bases [23] [17].

3 Software process and assistance modelling

3.1 Objectives

In the following, we introduce some objectives expected from software process and assistance modelling.

- Understanding the development process
- Reasoning to improve the development process, to verify it, and to verify what it produces
- Controlling the development process and the activities
- Measuring the development for a better understanding
- Evaluating the process itself, the activities, and the persons from a quantitative as well as a qualitative viewpoint
- Managing the activities and their various resources
- Integrating the different kinds of activity which are performed during the software process, like a development activity and a project management activity
- Guiding the development process as well as its definition
- Assisting the various activities
- Evolving of the model, the software products

3.2 Requirements

In this section, we define some requirements and functionalities which must be fullfilled by software process and assistance modelling.

- Life-cycle covering
- Description of objects, properties, and relationships at different level of granularity
- Description of allowed transformation on objects
- View mechanisms
- Structural and organizational descriptions

- Task and constraint descriptions
- Type and level of assistance specification
- Formality
- Readability and understandability
- Partial or total re-usability
- Modularity and evolutionary

4 Concepts of the UPSSA process model

A software process and assistance model is an abstraction of objects, tools and policies. Therefore, we propose to model it with an object model for describing the variety of objects, with an operator model for describing the tools and with characteristics for expressing the policies.

4.1 The object model

The aim of the object model is to provide a description of the object structures by means of object properties and relationships among objects at different levels of granularity. As already noted, software development involves various kinds of objects [24]. These include the design, specification, coding, and debugging of computer programs, as well as, the creation, maintenance, and reuse of modules and versions. Also. the software environment deals with objects involving the management and the control of an overall software development project,e.g., cost calculations and scheduling dependencies. Thus, the object model should enable the description of all these objects. Further, this description should be easily refined to satisfy the different users' needs.

So, we propose to use a model which is based on the entity-relationship data model [6] extended with ISA-Relationships. Objects are classified according to their respectives types. Using the entity-relationship concepts, an object-type is described as an *entity-type*. A list of attributes describe each entity-type. An *attribute* is viewed as a pair (attribute-name:attribute-domain). Formally, an entity-type can be viewed as a record-type whose fields are the attributes of the entity-type and attribute is defined as a function which maps the entity-type into the attribute-domain. In this framework, the value of an object is the values of its attributes plus the content of the object, if any. For example, one can define an entity-type called Module with (module-name:string, creation-date:date, modification-date:date) as its related attributes. Thus, the value of an occurrence of Module is its name, the dates when created and when modified.

Links among objects are described by means of relationships. Relationships are also typed. A *relationship-type* is specified providing: the name of the relationship, the list of the related entity-types, the cardinalities, that is the miminal and maximal number a given object may be linked to other ones within relationship and the attributes, if any, which describe the relationship. For instance, a relationship called Authorship may hold between the entity-types Module and Person. An attribute named date-writing may be attached to

the relationship. A (1,1) cardinality associated with Module indicates that every module must have a single author. In this framework, the value of a relationship is defined by a value for each related entity-type plus a value of each attribute which describes the relationship. A particular relationship is distinguished: the ISA-Relationship. This kind of of relationship enhances the original data model and enables a more "sophisticated" description of the object model [12]. Moreover, it assumes the availability of inheritance mechanisms to make the subtypes of a given type T inherit attributes of T. For instance, an entity-type Source-Module may have Pascal-Source-module, C-Source-module and Ada-Source-module as subtypes.

4.2 The operator model

The object model describes the structures that should be produced or those on which activities should be performed. An abstraction of these activities is described by the operator model. Even operators may be typed in the sense that an operator-type (e.g. edit) defines a class of operators, while an operator (e.g. edit using Emacs or edit using ed) is considered as a member of a given class. This operator-typing mechanism eases the addition of new tools to the environment.

An operator type is defined by its name, the precondition which must be fullfilled before activating any operator of this type, the postcondition which is assumed to be true at the end of the operator's execution, and the signature, i.e., the object types needed as inputs to activate any operator of the type (domain) and the object types produced by these operators (range).

Example: Compile-module
Signature: s:source-module ⟶ o:object-module ∨ e:error
Pre: edit(s) is true
Post: (object-of(s)=o xor comp-error(s)=e)

The definition of an operator enables the linking of a tool to an operator-type. An additionnal feature of an operator indicates whether it may be processed by the system alone, or if it needs any user's cooperation. The operator is then respectively considered to be *automatic* or *non automatic*. This information is included in the description of the operators. It helps the system in taking the initiative to activate an operator without a user's stimulus.

4.3 The characteristics

The characteristics of a software process model are a set of constraints that are satisfied during the activation of this model. They may serve as integrity constraints that must be satisfied. They can also serve as a basis for reasoning mechanisms [4] activated by the system when a characteristic vilolation is detected. Using the characteristic and the preconditions and postconditions of the operator, the aim of the reasoning is to find a way to make the characteristic true, i.e., to find a sequence of actions that should enable the system to enforce the characteristic. If this task fails, the system should undo the actions that lead to the characteristic violation. Thus, we say that the characteristics define the stable state of the system.

Example: compiled(x) $\Longrightarrow$ edited (x)

5 Interpretation of the UPSSA process model

The object model, operator model and characteristics introduced above constitute the basic knowledge of the UPSSA tool. We integrate parametric knowledge with the basic knowledge in order to interpret the UPSSA model. UPSSA reacts to the user's initiative by first, controlling its appropriateness. This control is performed on the basis of the control information included in the UPSSA process model and stored in the objects base. Whenever a user command (i.e., the execution of an operator) successfully passes through the control phase, it is performed. If it is not, plan generation is performed to fulfill the operation preconditions.

Furthermore, planning generation and execution facilities can be activated by the UPSSA whenever it notices a characteristic violation. Plan generation is performed thanks to operator's pre and postconditions. If plan generation task fails, the consequences of the user's command are undone.

Using an activity example of software development, we show in the following, how we structure and formalize the knowledge necessary to interpret the UPSSA software process model.

5.1 An illustration on Ada compilation

As an activity of software development step, we take the example of separated-compilation in ADA language. All Ada program units generally have a similar two-part structure, consisting in a specification and a body. The specification identifies the information visible to the user (the interface), while the body contains the unit implementation details which can be logically and textually hidden from the user. Ada enables submitting the text of a program in one or more compilations. Thus, the dependencies explicitly defined among units require that they must be compiled in a certain order. Basically, the rule is that a given unit must be compiled before it can be visible to another unit. In particular, the specification of a subprogram, package or task must be compiled before the corresponding body. So, orderings rules are constraints which must be satisfied during operators activation and they define a partial order which allows the etablishment of the possible recompilations consecutive to the recompilation of a unit.

5.2 The basic and parametric knowledge

Modeling and representing the knowledge

Using the concepts of the UPSSA software process model, a part of the previous text can be modeled as follows :

- Entity types : specification-unit, body-unit, subunit and library-unit.
- Relationship types : use, realize, is-included and isa.
- Operators types : compile-specification, compile-body, compile-subunit, modify-specification, modify-body and modify-subunit.

- Characteristics : orderings constraints which must be satisfied during operators activation.

Formalizing and describing the basic knowledge

To formalize the knowledge, we use a typed first order predicate calculus language similar to the one defined in [25]. The language symbols are extracted from the object model and the operator model. More precisely, we associate a symbol *type* for each entity type; a symbol *constant* for each entity instance, a symbol *n-ary predicate* for each n-ary relationship between the entities. By combining these symbols with logical connectors and quantifiers, we build the language formulae for describing the knowledge set. Thus, by considering the Ada compilation example we have :

- For the entity types, we associate the type symbols SPECIFICATION-UNIT, BODY-UNIT and SUB-UNIT. In the same way, we associate typed constants for the instances. For example, if obj_1 is an instance of specification-unit entity type, we write SPECIFICATION-UNIT:obj_1 or simply SPECIFICATION-UNIT(obj_1).
- For the relationship types, we associate the predicate symbols USE, REALIZE, IS-INCLUDED and ISA. These relationships only express static aspect, therefore we introduce other binary relationships : is-compiled-specification-in, is-compiled-body-in and is-compiled-subunit-in to express dynamic relations and the types compiled-specification, compiled-body and compiled-subunit. Similarly, we introduce the binary predicates IS-COMPILED-SPECIFICATION-IN, IS-COMPILED-BODY-IN and IS-COMPILED-SUBUNIT-IN, then the type symbols COMPILED-SPECIFICATION, COMPILED-BODY and COMPILED-SUBUNIT as symbols of the language description.
- For the operators, we associate the functional symbols ***compile***1(O1 : SPECIFICATION-UNIT), ***compile***2(O2 :BODY-UNIT), ***compile***3(O3 :SUB-UNIT), ***modify***1(O4 :SPECIFICATION-UNIT), ***modify***2(O5 :BODY-UNIT) and ***modify***3(O6 :SUB-UNIT). Since an operator is described by a pre-postcondition, we associate with each functional symbol two formulae of the language which represent operator pre-postconditions. We do not quantify universally the free variables, formulae are built from : typed constants which indicate instance types, binary predicates which indicate static and dynamic relationships, logical connectors $\wedge$ and $\Rightarrow$, and the existential quantifier $\exists$.
- Finally, the characteristics are described by means of formulae. It should be noted that, like the characteristics, the pre-postconditions contribute to maintain a safety state of the software under development. This is why we enhance the pre-postconditions by the characteristics in order to maintain this safety state.

Let us illustrate this formalization process on an operator. Figure 1 shows the description of the operator ***compile***1 which addresses a specification unit.

The knowledge presented above is enhanced by another kind of knowledge which is only used by the plan generation mechanism (cf. §5.2).

Strategies

There are several strategies for pruning the space states when generating a plan [19]. As examples, let us mention the depth-first strategy, the breadth-first strategy, the hierarchical refinement strategy and so on. Needless to give details on these strategies which are widely described in the literature of this topic.

Operator	*compile*1;
Signature	SPECIFICATION-UNIT $\mapsto$ COMPILED-SPECIFICATION:
Sorts	X, Z: SPECIFICATION-UNIT; T, Y: COMPILED-SPECIFICATION;
Pre	USE$(X, Z) \Rightarrow \exists\, T$ such that IS-COMPILED-SPECIFICATION-IN (Z, T);
Post	IS-COMPILED-SPECIFICATION-IN (X, Y);
Comments	This operator describes the compilation of a given specification unit. Because the latter uses another specification unit, we express that this unit must be compiled before the initial given specification unit.

Figure 1: Description of the operator compile1

Heuristics

In addition to strategies, heuristics are commonly used to prune the space states. In fact, heuristics serve to select the state to be explored and the operator to be used. In most cases, heuristics are closely related to the considered application. The next section gives examples of heuristics which have been used.

5.3 The plan generation mechanism

As mentioned earlier, maintaining the safety of a software under development can be a major problem. Suppose that the software initially is in a safety state. The characteristics can be violated during the development. In this case, our goal is to find a sequence of actions to fulfill the characteristics and restore the software at hand in a safety state. On the other hand, an operator cannot be activatable if its preconditions do not hold. Once again, our goal is to find a sequence of actions to fulfill the operator preconditions.

Let us turn now to the plan generation mechanism and consider :

- K, the set of the formulae which represent the knowledge about the objects and their relations.
- $OP = \{op_1, \ldots, op_n\}$, the set of the available operators. We denote by $pre_{op_i(arg)}$ (resp. $post_{op_i(arg)}$), the set of formulae that represent the preconditions (resp. the postconditions) of the operator op_i having arg as a given argument.

For example, suppose that we have the objects su_1, su_2 et su_3 of specification-unit type; the object cs_3 of compiled-specification type; the relations use(su_1,su_2), use(su_1,su_3) and is-compiled-specification-in(su_3,cs_3). Let us consider the operators *compile*(su_3) and *compile*(su_1). By adopting the notations described in §3.2, we have the following formulae:

$$K = \left\{ \begin{array}{l} \text{specification-unit}(su_1), \\ \text{specification-unit}(su_2), \\ \text{specification-unit}(su_3), \\ \text{compiled-specification}(cs_3), \\ \text{use}(su_1, su_2), \\ \text{use}(su_1, su_3), \\ \text{is-compiled-specification-in}(su_3, cs_3) \end{array} \right\}$$

$$pre_{compile(su_3)} = \{\ \emptyset\ \}$$

$$post_{compile(su_3)} = \{ \text{ compiled-specification}(cs_3) \wedge \text{is-compiled-specification-in}(su_3, cs_3) \ \}.$$

$$pre_{compile(su_1)} = \left\{ \begin{array}{c} \text{specification-unit}(su_1) \wedge \text{specification-unit}(X) \wedge \text{use}(su_1, X) \\ \Rightarrow \\ \text{compiled-specification}(Y) \wedge \text{is-compiled-specification-in}(X, Y) \end{array} \right\}$$

$$post_{compile(su_1)} = \{ \text{ compiled-specification}(X) \wedge \text{is-compiled-specification-in}(su_1, X) \ \}$$

In the following, we suppose that the interpretation $[\![K]\!](\mathcal{U})$ (where $\mathcal{U}$ is the actual problem universe) admits a model, the so-called intented interpretation. Intuitively, this means that the formulae set of K are consistent since K admits at last one model, in particular the *intentional* model. Clearly, when the preconditions of $op_i(arg)$, a given operator, are not satisfied, the interpretation $[\![K \cup pre_{op_i(arg)}]\!](\mathcal{U})$ does not hold.

To be able to apply the operator $op_i(arg)$, we are led to build at preliminary a finite sequence of operators $op_j(a_k) \subset OP$. By activiting those operators, the intented result is to augment K in such a way that $[\![K \cup pre_{op_i(arg)}]\!](\mathcal{U})$ holds. By using the artificial intelligence terminology [19], we refer to these finite sequences as plans.

From the theoretical point of view, we can construct several sequences from OP. So, without taking either the order of the operators application nor the choice of the arguments instantiation, we have 2^n possibilities where n is the cardinal of the set OP. This is a serious problem. However, it should be noted that every sequence of actions cannot be considered as *valid* plan. More precisely, we introduce the following definition.

Definition: Let $S = < op_{j_1}(a_{k_1}), \ldots, op_{j_m}(a_{k_m}) >$ a finite sequence of operators in OP. The sequence S constitute a *valid* plan $\mathcal{P}$ if and only if : $\forall i \in [1, \ldots, m]$, we have:

(1) $[\![K \cup pre_{op_{j_i}(a_{k_i})}]\!](\mathcal{U})$ holds;

(2) $((K \cup post_{op_{j_i}(a_{k_i})}) \cap K) \neq \emptyset$

Intuitively, the first condition tells us that the preconditions of each operator in the plan are respected. The second condition insures that any operator in the plan is not "unfruitful" or "sterile" i.e., the application of an operator necessarily augments the set of factual knowledge.

Although it is possible to generate several *valid* plans for a given operator, we are an interest to only generate one plan. Thus, we use the depth-first search strategy. Figure 2 gives the plan generation procedure for a given operator $op_i(arg)$.

```
GENERATE(pre_{op_i(arg)}, op_j(a_k), K)
|  % Step 1.: The initial operator, op_i(arg), is ready to be applied?
| • If [K ∪ pre_{op_i(arg)}](U) Then Return(SUCCESS);
|  % Step 2.: The selected operator is sterile?
| • If [K ∪ pre_{op_j(a_k)}](U) ∧ post_{op_j(a_k)} ∪ K = K Then Return(FAILURE);
|  % Step 3.: Augment K, the set of formulae.
| • K ← K ∪ post_{op_j(a_k)}
|  % Step 4.: Define the candidate operators to be applied
| • Operators ← {op_l(a_m)/a_m ∈ K ∧ [K ∪ pre_{op_l(a_m)}](U)}
|  % Step 5.: Building the plan by depth-first search
| • Repeat
| |  % Step 5.1: If there exist candidate operators?
| | • IF Operators = ∅ Then Return(FAILURE)
| |  % Step 5.2: Backtracking step, to select another operator if needed
| | %          Using the heuristics -see the note below-
| | • Operator ← SELECT-ONE(Operators)
| |  % Step 5.3: Retract the selected operator from the candidate operators
| | • Operators ← Operators\Operator
| |  % Step 5.4: Step forward in the plan generation.
| | • Résultat ← GENERATE(pre_{op_i(arg)}, Operator, K ∪ post_{Operator})
| • until Result ≠ FAILURE
|  % Step 6: Return the constructed plan
| • Return (Operator · Result)
End
```

Figure 2: Plan generation procedure

In the fourth step only the operators whose preconditions are satisfied, can be considered as candidates. Also, at the second step, any "sterile" operator is rejected by the procedure. Thus, we have the following proposition :

Proposition: Let $op(arg)$ a given operator; $\mathcal{P}$ the generated plan; K' the augmented set of formulae at the end of the execution of the GENERATE procedure. The followings hold :

- $\mathcal{P}$ is valid;
- $[K' \cup pre_{op_i(arg)}](\mathcal{U})$

Although, generating only valid plans restricts the choice to select an operator and thus pruning the space states, this remains somehow insufficient. Thus, we use the function SELECT-ONE at step 5.2. This function is based on heuristics such as to find an order, eventually partial, over the set of the candidate operators and to select the "best" operator which must be considered.

Generally, heuristics strongly depend on the activity at hand and more precisely on the kind of the considered operators. In our case, where the operators concern Ada units compilation, the ordering of the candidate operators is made by successive refinements using the following heuristics:

H_1 First, classify operators that have the signature SPECIFICATION-UNIT $\mapsto$ COMPILED-SPECIFICATION, then those with the signature BODY-UNIT $\mapsto$ COMPILED-BODY and finally those with the signature SUB-UNIT $\mapsto$ COMPILED-SUBUNIT.

This heuristic allows global and partial ordering. Then, in each group of operators, we apply the following heuristic.

H_2 Let us consider the operators $op_{j_1}(a_{k_1})$ and $op_{j_2}(a_{k_2})$ which have the same signature, we classify $op_{j_1}(a_{k_1})$ firstly if:

- $\exists\sigma$ such that $[\![K \cup \sigma(pre_{op_i(arg)})]\!](\mathcal{U})$,
- $card(\sigma(pre_{op_i(arg)}) \cap post_{op_{j_1}(a_{k_1})}) > card(\sigma(pre_{op_i(arg)}) \cap post_{op_{j_2}(a_{k_2})})$

where σ is a substitution on the free variables in the initial operator. Intuitively, this heuristic captures the following criteria : in the case of two given operators, we prefer to try, at the first time, the operator whose postconditions could contribute more to reaching the initial goal; that is, to satisfy the preconditions of the initial operator.

6 Architecture of the UPSSA tool

In this section, we just describe an outline of the architecture of UPSSA tool which is written in Le-lisp. The knowledge base contains three kinds of knowledge. The objects and the operators together with the pre-postconditions constitute the *basic knowledge*. The *parametric knowledge* which contains the strategies and the heuristics set. The main data structure is a hierarchy of frames. Figure 3 gives an example of two frames representing an operator and a heuristic.

```
(deframe                                    (deframe
 #:Operator:Compile:SpecificationUnit (x)    #:Heuristic:Organize* (x y)
  (#:Precondition:SpecificationUnit x)        ((#:Pred:SpecificationUnit x)
  (#:Precondition:SpecificationUnit y)        (#:Pred:BodyUnit y) x)
  (#:Precondition:Use y)                      ((#:Pred:SpecificationUnit y)
      ⋮                                       (#:Pred:BodyUnit x) y)
  (#:Add:List                                 ((#:Pred:BodyUnit x)
   ((#:Pred:SpecificationUnit x)              (#:Pred:SubUnit y) x)
   (#:Pred:IsCompiledSpecification y))            ⋮
  )                                           ((#:Pred:SpecificationUnit x)
  (#:Delete:List () )                         (#:Pred:SubUnit y) x)
)                                           )
```

Figure 3: An example of a frame representing an operator and a heuristic.

Finally the *system knowledge* includes the information about the global state of the system itself such a set of trees representing the generated plan.

The general architecture of the tool is portrayed in Figure 4. Like most knowledge-based system, it is organized around a **knowledge base** which is composed of the three kinds of knowledge listed above and **tools** that are the *inference mechanism* and the *man-machine interface*. Three modules form the *inference mechanism*. The *generator* module which is in charge of the generation of plans. The second module, called *interpreter*, is in charge of the interpretation of the end-user commands. This module invalidates the operator effect when the knowledge base become unsafe. Finally the third module, called *interface*, enables the access to the knowledge base and the communication with the *man-machine interface* tool. This last tool is composed of two modules. The *managing* module is in charge of the interaction with the end-user (the developer). The *updating* module enables the accessing and the updating of the knowledge base and also it transmits the developer queries to the inference mechanism.

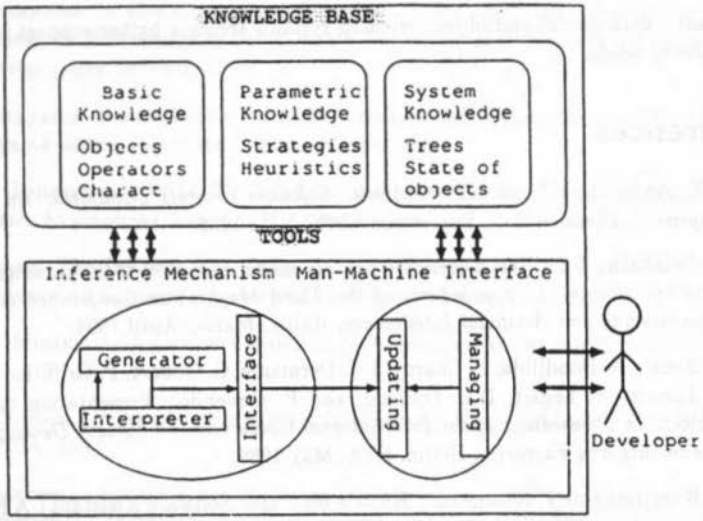

Figure 4: Architecture of the UPSSA tool

7 Conclusion

In this paper we have argued the need for software process and assistance modelling as an integrated part of a software engineering environment. This environment, called process-driven environment, provides automated support for software work based on a process model that controls some of the functionnality of the environment. The expected benefits of software process and assistance modelling are improvements in productivity and improvements in quality of the products. We discussed the objectives, requirements, and concepts for software process modelling, i.e., for capturing, describing, and using the variety of knowledge needed to build an *advanced software engineering environment logistics framework* fitted with assistance capabilities.

Beyond experimentations of the UPSSA tool to other activities, several extensions of this work can be mentioned. For example, in the formalisation of the knowledge (e.g., types, constants and operators), we have adopted a systematic way; thus, a (partial) mechanization of this formalization step is possible. Another example, let us consider the pre-postconditions expressed by the typed first order predicate calculus language. Within this formal framework and using formal techniques, it could be possible to prove some properties as the consistency. Finally, the enhancement task of the operator pre-postconditions by the characteristics is currently made manually. Once again, it could be possible to make use of transformational technics to (partially) mechanize these enhancement task.

Further, considering the variety of things to be described, like object structures, available tools, various relationships among objects, and policies induced by the tools or by a software development method, conventional database technology seems to be inadequate for software engineering needs and particularly for software process and assistance modelling In this framework, a "multiparadigm" model which incorporates both philosophies

of semantic data model and object-oriented systems seems a better support to software environment needs.

References

[1] T.K. Abdel-Hamid and S.E. Madnick. *Software Projects Dynamics: An Integrated Approach.* Prentice-Hall, Englewood Cliffs, N.J., 1991.

[2] N. Belkhatir. TEMPO: a software process model based on O.O. paradigm and objects viewpoints. In *Proceedings of the Third Maghrebian Conference on Software Engenieering and Artificial Intelligence*, Rabat, Maroc, April 1994.

[3] K. Benali, N. Boudjlida, F. Charoy, J.C. Derniame, C. Godart, P. Griffiths, V. Gruhn, P. Jamart, A. Legait, D.E. Oldfield, and F. Oquendo. Presentation of the ALF project. In *Proceedings of the International Conference on System Development Environments and Factories*, Berlin, RFA, May 1989.

[4] N. Boudjlida and T. Khammaci. Knowledge-Based Software Assistant : A Knowledge Representation Model and its Implementation. In *Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference (KBSA)*, pages 317–330, Syracuse, New York, USA, September 1990. Published as Intelligent Systems Review (Revisited Version), Vol 2, No 4, 1990.

[5] A. Brockers and V. Gruhn. Computer-Aided Verification of Software Process Model Properties. In *Proceedings of the Fifth International Conference, CAiSE'93*, pages 521–546, Paris, France, June 1993. Published as LNCS no 685.

[6] P.P. Chen. The Entity-Relationship Model: Toward an Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[7] B. Curtis, M.I. Kellner, and J. Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[8] J.C. Derniame. *Proceedings of the Second European Workshop on Software Process Modelling.* J.C. Derniame editor, Trondheim, Norway, September 1992.

[9] M. Dowson. *Proceedings of the First International Software Process Conference.* M. Dowson editor, Redondo Beach, CA, USA, October 1991.

[10] EC2. *Proceedings of the PCTE'93 Conference.* Ian Campbell, September 1993.

[11] K.E. Huff and V.R. Lesser. A Plan-based Intelligent Assistant That Supports the Software Development Process. In *Proceedings the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* pages 97–106, Boston, MA, USA, November 1988.

[12] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[13] G.E. Kaiser, P.H. Feller, and S.S. Popovitch. Intelligent Assistance for Software Development and Maintenance. *IEEE Soft*, 5(3):40–49, 1988.

[14] T. Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 343–352, 1989.

[15] T. Katayama. *Proceedings of the Sixth International Software Process Workshop.* T. Katayama editor, Hakodate, Japan, October 1990.

[16] M.M. Lehman and W.M. Turski. Essential properties of IPSEs. *ACM SIGSOFT, Software Engineering Notes*, 12(1), January 1987.

[17] B. Meyer. The Software Knowledge Base. In *Proceedings of the Eighth International Conference on Software Engineering*, London, UK, August 1985.

[18] N. H. Minsky. Law-governed systems. *Software Engineering Journal*, 6(5):285–302, 1991.

[19] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, CA, USA, 1980.

[20] L. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, CA, USA, March 1987.

[21] L. Osterweil. *Proceedings of the Second International Conference on the Software Process.* L. Osterweil editor, Berlin, Germany, February 1993.

[22] M.A. Ould and C. Roberts. Modeling iteration in the software processes. In *Proceedings of the Fourth International Software Process Workshop*, Moreonhamstead, Devon, UK, May 1988.

[23] M.H. Penedo, E. Ploedereder, and I. Thomas. Object Management Issues for Software Engineering Environments – Workshop Report –. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 226–234, Boston, MA, USA, November 1988.

[24] M.H. Penedo and E.D. Stuckle. PMDB - A Project Master Database for Software Engineering Environments. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 150–157, London, UK, August 1985.

[25] R. Reiter. Toward a Logical Reconstruction of Relational Theory. In M.L. Brodie, J. Mylopoulos, and Y. Schmidt, editors, *On Conceptual Modeling*, pages 191–238. North-Holland Publishing, 1984.

[26] S. Sutton, D. Heimbigner, and L Osterweil. Language constructs for managing change in process centered environments. In *Proceedings of the Fourth SIGSOFT Symposium on Software Development Environments*, pages 35–44, Irvine, CA, USA, December 1990. ACM Software Engineering Notes, 15(6):206-217, December 1990.

[27] L.G. Williams. A behavioral approach to software process modeling. In *Proceedings of the Fourth International Software Process Workhop*, pages 108–111, Moretonhampstead, Devon, UK, May 1988.