A Structured Approach to Software Process Modelling[1]

Xavier Franch
franch@lsi.upc.es
Universitat Politècnica de Catalunya
Jordi Girona 1-3, 08034 Barcelona
Catalonia (Spain)
FAX: 34-93-4017014. Phone: 34-93-4016965

Josep M. Ribó
josepma@eup.udl.es
Universitat de Lleida
P. Víctor Siurana 1, 25003 Lleida
Catalonia (Spain)
FAX: 34-973-702062. Phone: 34-973-702000

Abstract

Systematic formulation of software process models (SPM) is currently a challenging problem in software engineering. We present here an approach to define such models that encourages: reuse of both elements and models; modularity and incrementality in model construction; simplicity and naturality of the resulting model; and a high degree of concurrence in their enaction. In this paper we focus on model definition, distinguishing as usual its static and dynamic parts. We define the static part by means of formally defined hierarchies introducing the categories of elements that take part in SPM definition. Such hierarchies may be constructed and enlarged according to the requirements of any specific SPM. We present as an example a hierarchy for component programming that takes into account non-functional aspects of software (efficiency, etc). The dynamic part of the SPM is defined by means of precedence relationships between tasks that take part in the model. These precedence relationships are represented with precedence graphs. Development strategies are defined by encapsulating new precedence relationships in modules, that can be combined and reused.

1. Introduction

A model for a software development process [DWK97] (i.e., a *software process model*) is a description of this process expressed in some *process modelling language*. The process can be viewed as the execution in a suitable order of a set of *tasks* (e.g., requirements elicitation or module testing) intended to develop some *documents* (e.g., specification or test plan). These tasks are developed by some *agents* (e.g., people or hardware media) with the help of some *tools* (e.g., editors or debuggers) and using some *resources* (e.g., data bases or computer networks).

Hence, the definition of a software process model must state all the elements just mentioned, and also the way in which this model must be execute (*enacted*). This idea leads to the notion of *static* and *dynamic* parts of a model. The static part is given by the description of the tasks, documents, agents, tools and resources that take part in the software process model. On the other hand, the dynamic part consists of a description of the way in which software is developed; so, it mainly focuses in questions like what and how must be done to develop a piece of the model. The systematic description of both parts not only helps in understanding software development, but also makes feasible the construction of systems for supporting automation of the process up to an acceptable level, centered on the process modelling language.

Many differents approaches to such systems currently exist; see [FKN94] for a survey. Some of them have drawn a special attention within the scientific community, like EPOS [Con95], MERLIN [Jun95, RS97] or SPADE [BNF96], just to name a few of them. Although they support a lot of helpful properties in software development (e.g., process evolution, versioning, concurrency during enaction and cooperation among tasks), they seem to lack at least partially in supporting the following interesting ones:

- *Modularity* in model construction, i.e., the ability to build a model by combining several partial models using some operators. Although there are some proposals in this sense (remarkably [Chr94]), most of the reported environments seem not to support it. Modularity at the process model is as important as at the product level, aiding at building, understanding, maintaining and reusing software models.

[1] This work has been partially supported by the spanish project TIC97-1158, from the CICYT program.

- *Simplicity* in both the process of model construction and also the description of the resulting model. This property is not easily achieved in the systems we have studied so far, as we can see in the case studies appearing in [FKN94] and also [ABEL97]. Simplicity is a basic property in order to make these approaches useful for software teams developing real applications.
- *Formalisation* of the elements taking part in the software process models, and also of the notion of correctness of a model enaction. The existence of such formal basis would provide a well-established foundation to reason about model enaction.

In this paper, we present a process model language aimed at supporting these properties. The language is the kernel of our PROMENADE approach (PROcess-oriented Modellization and ENAction of software DEvelopments), currently in progress. Concerning the static part, we describe process elements by means of OOZE [AG91] classes, which provides a modular and formal description; simplicity is added by providing a graphical notation to describe class relationships. About the dynamic part, we formulate our approach by establishing precedence relationships between tasks, and by defining encapsulation mechanisms that enhance modularity; also, we provide the notion of correctness of a software development with respect to a software model. The proposal relies on a previous paper [FR97], which introduced the basis for the current dynamic part, but which lacked from a proposal for the static one.

2. The Static Part

The static part of the language is defined upon a hierarchy integrating all the items involved in software development: *Documents, Tasks, Tools, Agents* and *Resources*; also, we include a *Domain* class defining some auxiliary concepts (string, date, etc., and also many domain-specific ones). These items are encapsulated and defined formally through classes organised as a hierarchy. These classes act as classification criteria for other elements, introducing some attributes and operations that are inherited by their heirs. As shown in fig. 1, they are in turn heirs of the *Type* class, which is the root of the hierarchy. We use a plain arrow to represent inheritance.

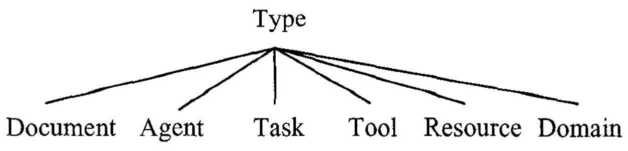

Fig. 1: The upper levels of the static hierarchy.

Although we can consider this classification enough to start software process models definition, we plan to provide many default hierarchies for different development contexts. In this paper, we will take as case study a hierarchy for dealing with *component programming* (*CP-hierarchy* for short). New elements defined in terms of classes can be added at a certain place of the hierarchy. The definition of a new element involves the definition of the attributes that every instance of the element must possess, the operations that may be applied to an instance of the element and the requirements that must hold in all the instances of the element (invariant of an element). Using the hierarchy helps in achieving reusability during software process modelling: any part of the hierarchy is always available in the definition of new models.

As mentioned above, and since one of our goals is to provide a formal framework for software process definition, it becomes essential to specify formally the elements that take part in model definition. We use the OOZE [AG91] formalism to do so. OOZE combines the widespread Z notation with some structuring mechanisms and, in particular, inheritance. So, the hierarchy drawn above can be inferred from OOZE classes. We plan to use the hierarchy as an external language for software process engineers and for documentation purposes too. Models described with OOZE have a well-defined formal meaning, which helps in determining the semantics of software processes.

As an example, we are going to develop in more detail the part of the hierarchy concerning documents and tasks.

2.1. Documents

The category of documents, known as such for being heirs of the *Document* class, defines the artifacts produced during software development. For instance, some significant documents are (see fig. 2): *SpecDoc,* compounded of *FspecDoc*, that contains the functional specification of a component and *NFSpecDoc*, for its non-functional specification; *ImplBehDoc*, compounded of *ImplDoc* and *BehDoc*, for functional and non-functional parts of implementations, respectively; *TestDoc*, to model the tests that are to be performed on some document (and that are compounded of the test code, *TestPlan*, and the test results, kept in *AvalDoc)*. The *Component* itself would be another example of a composite document since it is a product of the process of software development that contains other kind of documents as attributes. This category also includes other artifacts as *Workspace*, that keeps the environment of an agent at a given instant, *SpecLib* (a library for storing specifications) and

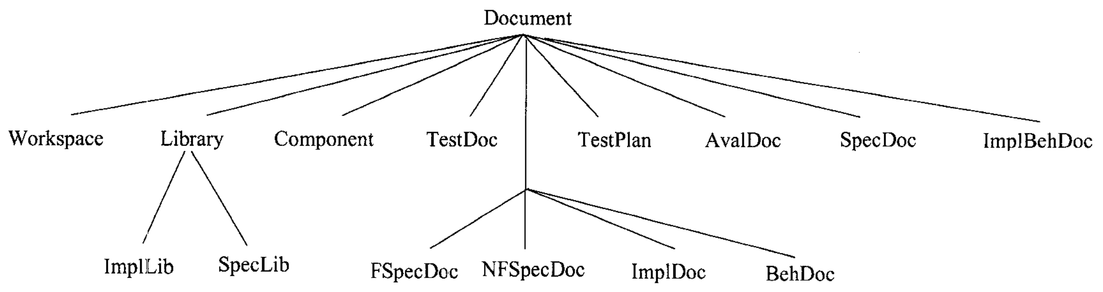

Fig. 2: The part of the CP-hierarchy concerning documents (only inheritance relationship is depicted).

ImplLib (another one for storing implementation documents).

Apart from the inheritance relationships depicted in fig. 2, other kind of relationships (for instance, *part-of* and *consists-of*, which are one inverse of the other) apply to document classes. We will present these relationships at the time they are needed.

2.1.1 The *Document* class

Document is defined with the following attributes: the document *identifier,* the dates of creation and last update on that document (creation, update), the document *status* (which may be *notComplete, complete* and *checked*) and, finally, the document's *owner*. *Document* is the superclass for all document types and it is specified in fig. 3.

```
Class Document < Type
  State
      identifier: String
      creation, update: Date
      status: StatDoc
      owner: Agent

  Init
      id?: String
      dat?: Date
      own?: Agent
    ------------------------------
      identifier'=id?
      update'=dat? ∧ creation'=dat?
      status'=notComplete
      owner'=own?
```

Fig. 3: The class Document

This specification contains some classes (like *String* and *Date*) that are defined in the *Domain* subhierarchy. Hereafter, operations for controlled attribute modification and selection are not shown.

2.1.2 The *Component* class

The *Document* subclass *Component* is defined with the following attributes: the specification document (*spdoc*), that contains the functional and non-functional specification for that component; and the implementation and behaviour document (*ibdoc*), which contains both the component implementation and the non-functional behaviour of that implementation. Notice that the attributes *identifier*, *creation*, *update*, *status* and *owner* are inherited from the class *Document.*

The specification of *Component* class in OOZE is given in fig. 4. One remarkable aspect of this specification is the class invariant which states that the status of a component is *checked* if and only if the status of all the documents it is compounded of are *checked*; this property will be usual in compounded documents. Also, we state that if both documents are finished, the last update of the implementation must be greater or equal than specification's one.

The definition of *Component* as a union of various documents brings up the topic of the existence of other kind of relationships between classes. A new kind of relationship between classes may be introduced in this case: the *consists-of* relationship. We say that a class *A* consists of classes *C1*, ..., *Cn* if and only if *A* can be defined as the *cartesian product* of *C1*, ..., *Cn*, i.e., $A = (C1 \times C2 \times \ldots \times Cn)$. We identify a *consists-of* relationship between the classes *SpecDoc, ImplBehDoc* and the class *Component* (a *Component consists-of* a *SpecDoc* and a *ImplBehDoc*).

```
Class Component < Document
  State
    spdoc: SpecDoc
    ibdoc: ImplBehDoc
  ----------------------------------------
    Status=checked ⇔
     (spdoc.status=checked ∧ ibdoc.status=checked)
    Status=checked ⇒ spdoc.update ≤ ibdoc.update

  Init
    id?: String
    dat?: Date
    own?: Agent
  ----------------------------------------
    identifier'=id?
    update'=dat? ∧ creation'=dat?
    status'=notComplete
    owner'=own?
```

Fig. 4: The class Component

2.1.3 The classes for specifications

SpecDoc consists-of the functional (*FSpecDoc*) and non-functional specification (*NFSpecDoc*) for a component. Its specification is similar to the one for *Component* and it is not shown.

The *FSpecDoc* document is defined with the following attributes: *spec* (the functional specification of a component expressed in some formalism); *limport* (a list containing the functional specification documents that must be imported in order to complete this one); and *testdocs* (which contains a list with all the test cases for the verification of the functional specification document along with the results of each test). We are not choosing here a particular formalism for the specification; different components may be specified in a different way. Specification styles may come into existence just defining their characterisation by means of new OOZE classes declared as heir of the specification one.

The specification of *FSpecDoc* document in OOZE is given in fig. 5. The class invariant establishes that *FSpecDoc* will be considered to be *checked* only after all tests planed to be run on it have finished successfully and they have been executed with the current version of the specification, which is checked using the last update date (the isolate *update* reference is applied to the current class, *FSpecDoc* in this case). The class *TestDoc* and its heirs are presented next.

```
Class FSpecDoc < Document
  State
    spec: Specification
    limport: seq String
    testdocs: seq TestDocFS
  ----------------------------------------
    Status=checked ⇔
       (∀td: TestDocFS | td∈testdocs •
              td.avald.success=true ∧
            td.avald.update < update)
  ... init and other operations
```

Fig. 5: The class FSpecDoc

The class *NFSpecDoc* is defined in a similar way. Also, implementations work the same way as specifications do and are not shown here.

2.1.4 The *TestDoc* class

This is the class that performs some kind of test (including both the test code and the test results) on the different documents (namely *FSpecDoc, NFSpecDoc, ImplDoc* and *BehDoc*). We consider a different kind of *TestDoc* class for each class of document (i.e. *TestDocFS* for testing *FSpecDoc* classes; *TestDocNFS* for testing *NFSpecDoc* classes ...). Hence we need to enlarge the type hierarchy of fig. 2 by making these new classes heirs of *TestDoc*.

Let us present, as example, the class *TestDocFS*. The attributes of this class are the following: *testeddoc* (the document which is being tested), *testpl* (the test code) and *avald* (the document containing the result of such test). Fig. 6 contains a specification of this class with the usual class invariant involving status and dates.

Test classes introduce another kind of relationship: *is-tested-in*. We say, for instance, that a *FSpecDoc is-tested-in* a *TestDocFS*. Unlike the ones presented up to now, this is a user-defined relationship, local just to a part of the hierarchy. These new relationships may be later used in the dynamic part of the model. On the other hand, the *consists-of* relationship may also be applied here since each kind of *TestDoc* class *consists-of* a *TestPlan* and an *AvalDoc*.

```
Class TestDocFS < TestDoc
  State
  testeddoc: FSpecDoc
  testpl: TestPlanFS
  avald: AvalDocFS
  ------------------------------------------
  Status=checked ⇔
      (testpl.status=checked ∧
       avald.status=checked)
  Status=checked ⇒ testpl.update ≤ avald.update
------------------------------------------
...init and other operations
```

Fig. 6: The class TestDocFS

2.1.5 The classes for libraries

The class *Library* is meant to store developed documents. Hence it will only contain documents with a *checked* status. In the default CP-hierarchy, we consider just two kinds of libraries (although it can be worthy to define other ones): *SpecLib*, to store specification documents and *ImplLib*, to store implementation documents; note that tests are part of these documents. Another kind of relationship between classes rises with libraries: *is-stored-in*. For instance, a *SpecDoc is-stored-in* a *SpecLib*, while a *ImplDoc is-stored-in* an *ImplLib*. We store in the corresponding library the *SpecDoc* as a whole (it is not allowed to store only the *FSpecDoc* or the *NFSpecDoc* for a given component). Fig. 7 shows the specification of *SpecLibrary* class. The two class invariants state, respectively, that all the documents contained in the library are in a *checked* status and that a library is self-contained (i.e. all the documents imported by a document stored in the library must be also stored in the library). Note also that the invariant allows the stored version of the specification not to be the last one.

2.1.6 The *Workspace* class

The *Workspace* class represents a document repository compounded of those documents that are visible to an agent at a given instant. This includes some documents taken from some library and some other documents which are being constructed.

The relationship between classes *is-stored-in* may also apply here. But in this case it is not compulsory to store in the workspace the pair of specification (or implementation) documents, because documents may be incomplete in the workspace. The OOZE specification is straightforward and we do not include it here.

```
Class SpecLib < Library
  State
  sl: seq SpecDoc
  ------------------------------------------
  ∀d: SpecDoc | d ∈ sl  •  d.status=checked

  ∀d: SpecDoc | d ∈ sl  •
     (∀d': SpecDoc  | d ' ∈ d.limport  •  d' ∈ sl)
------------------------------------------
.. init and other operations
```

Fig. 7: The class SpecLibrary

2.2. Tasks

A task represents an action that must be performed in the process of software development. It may be a composite action, which, in turn, will be decomposed in more simple tasks called subtasks, or an atomic one.

Since the software process model we propose is mostly task-oriented, task elements have a major importance in it. We define tasks as classes in the type-hierarchy. Tasks are specified by means of OOZE classes which attributes represent the parameters of the task. An additional parameter keeps track, at enaction time, of the task's subtasks that have been executed. The class is also provided with two methods, named respectively *begin* and *end* that are called at the starting and end instants of the task execution. Both methods perform everything needed to keep the consistency of the task (e.g. *begin* puts the task status to *active*, initalizes some task parameters, etc.; *end* calculates the *success* condition of the task, puts the task status to *complete*, etc.).

The issue of how to get the functionality of the task (i.e. in which way we describe the actions to be undertaken in order to get the task goals) is the main matter of the dynamic part of the model, developed in the next section.

We present in fig. 8 and 9 two exemples of task specification: the class *Task* which acts as superclass for all tasks, and *TestFSpec*, which performs the test of a functional specification document) with respect to some *test plan.* The parameters of this last task are the *FSpecDoc* (see fig. 5) we want to test and the *TestDoc* (in this case *TestDocFs*, see fig. 6) used to perform this test. At the beginning, the link between the specification document and the tests is created. At the end, the success condition of the task, *success'*, evaluates to true if all the single tests which is compounded of have been executed successfully. We call *SingleTestFSpec* the class of tasks that performs a single test on a functional specification document; we consider that each of these tasks includes a test *plan*, a test *result* and success condition. Finally we state that the test results stored in the evaluation document are exactly those produced by the application of *SingleTestFSpec* tasks on the actual instance of *FSpecDoc*.

```
Class Task < Type
  ── State ──────────────────────────
    status: StatTask
    success: Bool
    subtasks: P Task
    executed: seq Task
  ───────────────────────────────────
    ran executed ⊆ subtasks
    status = complete ⇒ ran executed = subtasks

  ── Init ───────────────────────────
    sb?: P Task
  ───────────────────────────────────
    status'=idle
    subtasks'=sb?

  ── begin ──────────────────────────
    status' = active
    executed' = <>

  ── end ────────────────────────────
    status'= complete
```

Fig. 8: The class Task

3. The Dynamic Part

The dynamic part of the model states (1) what must be done during model enaction (i.e. what tasks are to be executed), and (2) what constraints are to be applied in such enaction (i.e. what precedences in task executionmust be satisfied). We rely on task decomposition in order to state what a task must do (i.e. what subtasks are involved in its execution) and we define precedence relationships between tasks in order to establish precedence constraints involving task enaction

```
Class TestFSpec < Task
  ── State ──────────────────────────
    fspecld: FSpecDoc
    test: TestDocFs

  ── begin ──────────────────────────
    fspecdoc?: FSpecDoc
    testp?: TestPlanFS
  ───────────────────────────────────
    fspecld'=fspecdoc?
    test'.testpl=testp?
    test'.testeddoc=fspecdoc?
    fspecld'.status=complete

  ── end ────────────────────────────
    success'= (∀p: Code | p ∈ test.testpl.specbody •
                (∃tpart: SingleTestFSpec •
                   tpart ∈ executed ∧
                   tpart.plan=p ∧
                   tpart.success=true))
    ∀r: Result • r ∈ test.avald'.lresults ⇔
          (∃tpart: SingleTestFSpec |
            tpart∈executed •
             (tpart.result=r ∧
                tpart.success=true))
    fspecld'.status=checked ⇔ success'=true
```

Fig. 9: The class TestFSpec

3.1. Precedence relationships

Precedence relationships between tasks state the requirements that must be satisfied in order to be able to start the execution of a task. These requirements are established in terms of the tasks whose execution must have finished successfully in order to start the execution of a given task. More precisely, we say that there is a precedence relationship from task A to task B ($A \to B$) iff a requirement needed in order to initiate task B is that task A has been completed successfully (i.e. with success condition evaluating to *true*).

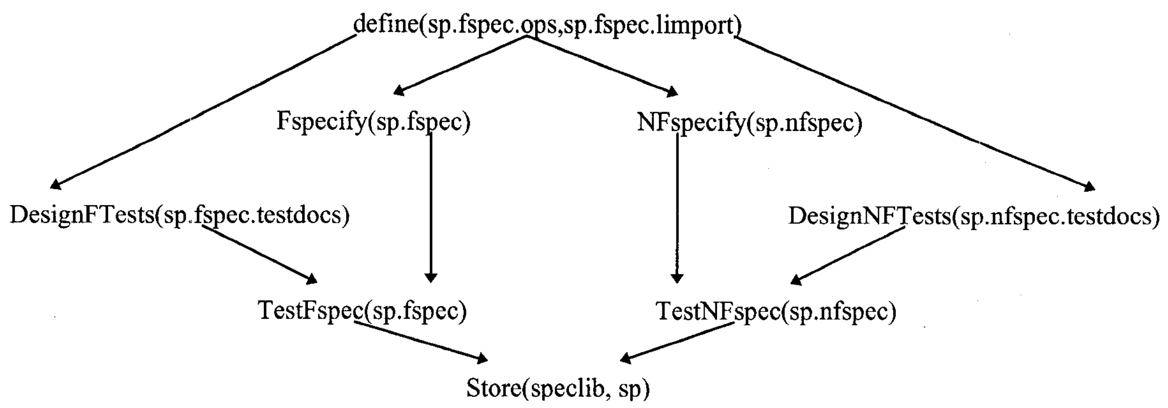
define(sp.fspec.ops,sp.fspec.limport)
Fspecify(sp.fspec)
NFspecify(sp.nfspec)
DesignFTests(sp.fspec.testdocs)
DesignNFTests(sp.nfspec.testdocs)
TestFspec(sp.fspec)
TestNFspec(sp.nfspec)
Store(speclib, sp)

Fig. 10: A possible precedence graph for developing specifications.

We can represent precedence relationships between tasks by means of precedence graphs, being their nodes tasks, and their edges precedences. Fig. 10 presents a precedence graph for developing a specification with functional and non-functional parts. We use tasks for defining the operations of the component, to create both parts, to design tests for them, for carrying the tests out and for storing the two parts in the specification library as a whole. Note that tasks appear parameterised, using the attributes introduced in the involved classes. This example graph is a default one in PROMENADE, and it could be inferred from some relationships stated at the static level, mainly having to do with dates and success conditions.

It is important to notice that by describing tasks using precedence relationships we state all the interactions that must be observed between tasks during model enaction. Apart from those interactions, the process engine is free to select any other execution order among not related tasks. This improves the concurrency of model enaction.

3.2. Development strategies

Given the modelisation of precedence relationships using graphs, we can consider a development strategy as a set of new edges binding nodes of these graphs. Sometimes, edges will relate tasks (nodes) in the same graph, to say things like "the functional specification of a component must be developed before the non-functional one"; however, in the general case, edges will involve tasks appearing in graphs bound to different modules, as in "it is necessary to specify all the components imported by a component *M* before any implementation of *M* is built". Sets of related rules are encapsulated in *strategy modules*. For instance, we show in fig. 11 three different strategy modules that add edges to the graph presented in fig. 10. The first one forces the finalization of the functional specification before starting the non-functional one. The second one implements the idea of bottom-up specification, saying that imported specifications must be finished before starting new ones. Last, a new strategy can be stated just by combining the previous ones.

```
strategy FUNCTIONAL_BEFORE_NON_FUNCTIONAL
  sp: SpecDoc
  Fspecify(sp.fspec) -> Nfspecify(sp.nfspec)
end module
```

```
strategy BOTTOM_UP_SPECIFICATION
  sp, Z: SpecDoc
  for all Z in sp.fspec.limport:
    Fspecify(Z.fspec) -> Fspecify(sp.fspec),
              NFspecify(sp.nfspec)
    NFspecify(Z.nfspec) -> NFspecify(sp.nfspec)
end module
```

```
strategy BOTTOM_UP_WITH_FUNCTIONAL_BEFORE
  combines BOTTOM_UP_SPECIFICATION,
  FUNCTIONAL_BEFORE_NON_FUNCTIONAL
end module
```

Fig. 11: Some strategies for specification development.

3.3 Modular process construction

In PROMENADE, model definition is intended to allow software process model construction in a modular and incremental way. One part of this modular construction of models relies on the reusability and extensibility of the hierarchy containing the static elements of the model. The other, and more fundamental part, deals with the process of task description.

Tasks play the role of ruling process development by appliying strategies. Constructing a software process model in a modular way consists in selecting within a *strategy library* those ones with the required functionality and combining them with some suitable precedence relationships in order to build a *description graph* for the model. This will define the model strategy.

Following this process, it is possible to combine many partial models to form the final one. These combination may be of two kinds. On the one hand, we can build precedence graphs for a subset of software documents (specifications, libraries, working context, etc.) by adding new precedences to an initial graph, by joining two graphs with the same nodes, etc. On the other hand, we can just put together some of these graphs to obtain a new one covering more documents (i.e., dealing with more software development stages); for instance, we can put together the graph of fig. 10 with other concerning implementation construction, to obtain a graph covering the whole component development process. The resulting graphs, if convenient, may be in turn, stored in a library for future reuse.

3.4. Correctness concerns

Using the precedence graphs and also the success conditions stated in the static part of the model, it is possible to formulate some correctness conditions, both concerning the model itself and also concerning a particular development process with respect to a model. This issue has been outlined in [FR97] and has been refined by incorporating the idea of three dimensional graphs, in which an axis corresponds to precedence relationships and the other to task decomposition.

Another point concerning the correctness is the matching between the static and the dynamic parts of the model. Obviously, precedence relationships between tasks and decomposition of elements into smaller parts must agree. We are currently working on the characterization of this matching.

4. Conclusions and future work

We have presented the process modelling language of the PROMENADE approach. This language addresses to many interesting properties at the process level which we think are not currently totally covered in the field. The PROMENADE approach plays a part in a more ambitious system called ComProLab [FBBR97] defined to support many different aspects of component programming.

In our proposal, the language consists of static and dynamic parts. Concerning the static part, we use a hierarchy to introduce software process elements (documents, tasks, etc.), which are encapsulated using OOZE classes and thus provided of a clear semantics. With respect to the dynamic part, we use precedence graphs as the underlying model of task enaction ordering, and we allow the definition of development strategies using modules that can be reused and combined.

We can classify and evaluate the adequacy of our approach with respect to the aspects proposed by Conradi and others in [CLJ91]. We put a star (*) on those issues still not covered but just planned to:

- *Basic process model apparatus:* Precedence relationships between tasks modelled by precedence graphs. Static part covered by class hierarchies.
- *Coverage of process entities:* products, activities, tools, agents, roles and resources. Our approach is, however, activity-oriented.
- *Coverage of software process life-cycle:* All steps can be modelled in our approach, including specification of software attributes (non-functional specification).
- *Task structuring*: We present a fully task structuring by means of task abstraction.
- *Type structuring*: Yes, in an object-oriented way. We define a default hierarchy of types that may be extended on demand.
- (*) *Customization and evolution of the process model:* It will be allowed by using meta-types.
- *Concurrence*: Supported by the fact that just precedence relationships avoid concurrent enaction of tasks.
- (*) *Configuration management:* A usual versioning system will be provided.

We think that the most interesting points of our approach are:

- ***Modularity/reusability:*** PROMENADE offers the possibility of reusing fragments of existing models under an object-oriented approach. Hence, in order to build a new model it is possible to reuse tasks, documents, roles and any other element previously generated to construct some other model, which makes the process of model construction much more simple. PROMENADE shares this feature with some other systems like EPOS and E^3.

On the other hand, some well-known systems like SPADE, ADELE or MERLIN lack this property or, at least, it is not shown explicitly how to get it in the revised literature.

What is new in PROMENADE with respect to the revised systems is its explicit ability of constructing in a modular manner new models adding some strategies to existing ones. These strategies are encapsulated in what we call strategy modules, which are presented briefly in section 3.2 and in more detail in [FR97].

- ***Simplicity/comprehensibility***: PROMENADE seems to facilitate the generation of software process models (SPMs) in an intuitive and quite simple way by means of a graphical representation (that will be translated into a formally defined language, which is currently being defined). This graphical representation is based on defining hierarchies of entities (for the static part) and new tasks by means of stating the precedence relationships that must hold between some other tasks (for the dynamic one).

On the other hand, once the SPM has been constructed, precedence graphs make it quite comprehensible. Precedence graphs (i.e. diagrams which depicts the precedence relationships between activities) are crucial in order to understand the whole process of software development. Curiously enough, most of the Process-centered Software Engineering Environments (PSEEs) we have explored do not use this kind of diagrams.

Two aspects that also help in the achievement of simplicity are the object-oriented approach we undertake and the high level constructs we provide for our system. Unlike other PSEEs, like SPADE or APPL/A, we do not require the software engineer to explain *how the SPM will be enacted* (which is usually a criptic matter) but instead, *what must be done in order to develop software*, which is clearly what the software engineer knows (and what he/she is interested in modelling).

Although the *simplicity* property should be a very important one for PSEEs, most of them fail (at least to some extent) in achieving it[2]. SPADE and APPL/A are directly enactable PSEEs that get a remarkable performance in model enaction. Since they achieve that performance on the basis of a low level construction, we think that SPM written in these systems (specially in SPADE) are difficult to write, difficult to read and difficult to understand. MERLIN and EPOS are *high level PSEEs*, but in our opinion, it is difficult to get with them fully comprehensible models: the resulting model in MERLIN is *working-context-oriented* which makes it difficult to grasp the whole development process model.

In EPOS the task sequence to be executed in the resulting SPM is not obvious and the advantage of using an AI planner is not clear. Finally, E^3 provides simplicity on the basis of a wide range of diagrams (including precedence ones), an object-oriented approach and high level constructs. This leads to a very comprehensible model but with quite simplistic constructs.

- ***Direct enactability:*** There is a usual correlation between the level of the constructs offered by the system in order to generate a SPM and the enactability of the resulting model. For instance, directly enactable systems like APPL/A and SPADE offer quite low level constructs. On the other hand, systems like E^3 and MERLIN, by far more *high-level systems*, cannot be enacted directly; they need a translation into a *lower level language*.

PROMENADE achieves a balance between the level of the language to create the model (which is clearly *high level*) and the SPM enactability, since a PROMENADE SPM is directly enactable by the application of the algorithm described in [FR97].

- ***Concurrent model enaction:*** Not only offers PROMENADE an enactable SPM but also a very natural concurrent model based on a multiagent approach (each agent is responsible for executing some tasks) with the restrictions imposed by the precedence relationships between tasks. We are currently working on these enaction aspects. An overview of them may be found in [FR97].

Concurrent models offered by some other PSEEs need synchronization between tasks (the case of APPL/A since it uses Ada tasks). In other cases (as SPADE) the concurrent model is based on a formal mechanism (Petri nets). This leads to a very efficient but low level approach. In some other cases of not directly enactable models (MERLIN, E^3), the concurrence of the final enactable model is not reported.

It is important to say that, unlike other systems like ADELE or MERLIN, we do not deal, for the moment, with concurrent accesses to documents, which is a very important research area.

- ***Formality:*** This is one of the shortcomings of most PSEEs, for only a few of them are constructed on the basis of a formal approach.

One of the goals of PROMENADE is to define a SPM established on some formal basis. This will allow the complete understanding of the specification of the elements (classes and relations) that take part in model definition; the rigorous definition of the mechanism of model enaction; the establishment of the correctness of a

[2] Although in some cases it is possible that the complexity of the model is a consequence of the inherent complexity of the problems that are to be solved.

model enaction with respect to a model definition; the accurate definition of what a SPM is and what is compounded of; the formal reasoning about the properties that hold in any part of the SPM, etc.

We achieve this property by the formal specification of classes by means of the OOZE language (see section 2), the rigorous model definition and the formalization of model enaction (which includes the notion of correctness of a software system development with respect to a SPM). Both aspects are shown in our previous work [FR97].

Some points of our approach have not been included here. In fact, the language has just been outlined. We have a catalogue of class relationships in the static part. In the dynamic one, we have a special kind of precedence relationship called *grouping* (simultaneous enaction of several tasks) that has been proved to be useful. Also the actions to be taken in case of task failure have not been shown here; they take the form of adding new kind of edges to the precedence graphs.

Other points have not been addressed yet in our work. Remarkably, we have not defined the meta-level, and our way to deal with component redevelopment is very naïve. Also absolute time must be added in our approach and combined with precedence relationships in a proper way. However, we feel that all these issues could fit in PROMENADE in a natural way.

References

[ABEL97] J. Arlow, S. Bandinelli, W. Emmerich and L. Lavazza. "Fine grained Process Modelling: An Experiment at British Airways". *Software Process Improvement and Practice.* Wiley, 1997.

[AG91] A.J. Alencar, J.A. Goguen. "OOZE: An Object-Oriented Z Environment". *European Conference on Object-oriented Programming (ECOOP'91),* Geneva (Switzerland), July 1991.

[BNF96] S. Bandinelli, E. Di Nitto, A Fuggetta. "Supporting Cooperation in the SPADE-1 Environment". *IEEE Transactions on Software Engineering,* 22(12), December 1996.

[Chr94] G. Chroust. "Partial Process Models". *Software Systems in Engineering,* PD-vol. 59, 1994.

[CLJ91] R. Conradi, C.Liu, M. L. Jaccheri. "Process Modeling Paradigms: An Evaulation". Position paper at *7th International Software Process Workshop (ISPW7),* Yountville (Ca, USA), 1991.

[Con95] R. Conradi. "PSEE architecture: EPOS process models and tools" . *Workshop on Proces-centered Software Engineering Environments Architecture,* Milano, March 1995.

[DWK97] J.C. Derniame, B. Warboys and A.B. Kaba (eds). *Software Process: Principles, Methodology, Technology.* Springer Verlag, 1997.

[FBBR97] X. Franch, X. Burgués, P. Botella, J.M. Ribó. "ComProLab: A Component Programming Laboratory".*9th International Conference on Software Engineering and Knowledge Engineering (SEKE),* Madrid (Spain), 1997.

[FKN94] A. Finkelstein, J. Kramer, B. Nuseibeh. *Software Process Modelling and Technology.* J. Wiley & sons, 1994.

[FR97] X. Franch, J. M. Ribó. "Software Process Modelling as Relationships between Tasks". *23rd EUROMICRO,* Budapest (Hungary), September 1997.

[Jun95] G. Junkermann. "A Dedicated Process Design Language based on EER-models, Statecharts and Tables". *7th International Conference on Software Engineering and Knowledge Engineering (SEKE),* Rockville, (Maryland, USA), June 1995.

[RS97] W. Reimar, W. Schafer. "Towards a Dedicated Object-Oriented Software Process Modelling Language". Workshop on Modeling Software Process and Artifacts, held at the 11th European Conference on Object-oriented programming, Jyvaskyta (Finland), June 1997.