PROMENADE: A Modular Approach to Software Process Modelling and Enaction

Xavier Franch¹, Josep M. Ribó²

¹ Universitat Politècnica de Catalunya (UPC).
² Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
tranch@ist.upc.es
² Universitat de Lleida
P. Victor Siurana 1, 25005 Lleida (Catalunya, Spain)
josempa@eup.udl.es

Abstract. We present in this report PROMENADE, an approach for supporting software process modelling. PROMENADE integrates: UML in order to define the elements that are involved in the process; and precedence relationships, triggers and exceptions, to state the dynamic part, i.e., the ordering of the tasks to be executed during the process. The dynamic part yields a graph as the model of the process, with different types of edges that reflect some different kind of relationships between tasks. PROMENADE offers also some constructs aimed at supporting modularity at the process level, enhancing thus reuse, enlargement, combination and refinement of models. As a particular framework, we develop the examples of the report in the component programming field.

1 Introduction

A model for a software development process (i.e., a software process model [DKW99], SPM for short) is a description of this process expressed in some process modelling language (PML). The process can be viewed as the cooperation of many tasks (e.g.: requirements elicitation, component testing) that use and develop some documents (specification, test plan) with the help of some tools (CASE-tools, debuggers) and using some resources (data bases, computer networks). Tasks involve many agents (people, hardware media) which play specific roles (programmer, manager) and which coordinate through some communication media (e-mail, fax).

Hence, the definition of a SPM must state all the elements just mentioned, and also the way in which this model must be executed (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 elements that take part in the SPM. On the other hand, the dynamic part consists of a description of the way in which the model is enacted. So, it mainly focuses on questions like what and how must be done to develop a piece of the product. The systematic description of both parts not only helps in understanding software
development, but also allows the construction of systems for supporting automation of the process up to an acceptable level.

Many different approaches to this challenge currently exist; see [FKN94] and [DKW99] for a survey. Some of them have drawn a special attention within the scientific community; we think about APEL [DEA98], E3 [JPL99], EPPOS [CLM+95], MERLIN [RS97] or SPADE [BNGL94], just to name a few. Although most of them support a lot of helpful properties in software development (e.g., process evolution, versioning and support to enaction), they seem to lack at least partially in supporting:

- **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 [Chr94], most of the reported environments seem not to support it. Modularity at the process model is as important as in product level, aiding at building, understanding, maintaining and reusing software models.

- **Simplicity** in both the process of model construction and also the description of the resulting model, while keeping enactability and a high degree of expressivity. 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 [ABE197]. This tradeoff between simplicity and expressivity is a basic property in order to make these approaches useful for developing real applications.

- **Formalisation** of the elements taking part in the SPMs, and also of the notion of correctness of a model enactment. The existence of such formal basis would provide a well-established foundation to reason about model enactment.

For these and other reasons (e.g., integration of different paradigms), a second generation of PMLs is actually coming into existence [WBG+98], as JIL [SO97] and others. Intended to be part of this generation, we present here a PML 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; we call the language PROMENADE too. Concerning the static part (section 2), we describe process elements by means of UML diagrams [RJB99], which provides a modular and formal description using a widespread notation; simplicity thus is achieved by means of the UML graphical notation itself. About the dynamic part, we formulate our approach by combining two complementary paradigms for reactive and proactive control: precedence relationships between tasks, exceptions and triggers (section 3); and by defining mechanisms that enhance modularity (section 4).

## 2 The Static Part

The static part is built upon three kinds of information, which yield to several complementary UML class diagrams built with the help of the Rational Rose application. First, the individual information of the classes themselves, including constraints. Second, a class hierarchy which integrates all the documents by means of inheritance (the UML generalization relationship). Last, other association relationships between classes, both standard (aggregation) and user-defined.
2.1 Classes and their members

In our O.O approach, a generalization hierarchy of classes is the natural way to represent the many concepts involved in process modelling. Classes are characterized by many attributes and support many methods. Valid value attributes are stated through class invariants, while methods will be specified through pre and post conditions.

Table 1 summarizes those classes. As heirs of a Type superclass, all of them share a few common attributes, such as identifier. We show here some relevant attributes and methods. We do not refer to neither structural methods (i.e., those methods that simply set the value of a single attribute) nor attributes that are directly related with the associations defined in 2.2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Some instances</th>
<th>Attributes</th>
<th>Some methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>Any container of information involved in the software development process</td>
<td>A specification; a component implementation; an e-mail</td>
<td>Link to contents; relevant dates; version; status</td>
<td>Document updating with or without new version creation</td>
</tr>
<tr>
<td>Communication</td>
<td>Any document used for people communication; can be stored in a computer or not</td>
<td>A fax; an e-mail; human voice</td>
<td>Link (if any) to contents; transmission date; status</td>
<td>Send and Read</td>
</tr>
<tr>
<td>Task</td>
<td>Any action performed during the software development process</td>
<td>Specification; component implementation; error-reporting</td>
<td>Precondition; status; success condition; deadline (if any)</td>
<td>Changes of task status (see 3.1)</td>
</tr>
<tr>
<td>Agent</td>
<td>Any entity playing an active part in the software development process</td>
<td>Myself; my SunSparc workstation; a compiler</td>
<td>Profile; location; (for humans) skills</td>
<td>Just structural ones</td>
</tr>
<tr>
<td>Tool</td>
<td>Any agent implemented through a software tool</td>
<td>A compiler; a navigator</td>
<td>Root directory for the tool; binary file location</td>
<td>Just structural ones</td>
</tr>
<tr>
<td>Resource</td>
<td>Any help to be used during software development process</td>
<td>An online tutorial on Java applets; a Web site</td>
<td>Location; platform requirements</td>
<td>Access</td>
</tr>
<tr>
<td>Role</td>
<td>Any part to be played during the software development process</td>
<td>Programmer; manager</td>
<td>Tasks for which the role is responsible</td>
<td>Just structural ones</td>
</tr>
</tbody>
</table>

These classes are put together in a natural way by defining some generalization relationships between them (see fig. 1; all the figures of this section are generated using Rational Rose). The intermediate class Element appears to distinguish the classes presented here from the metaclasses presented in section 2.3.
The default hierarchy may be extended by adding new classes in a classical manner, and this allows the creation of concrete models given different criteria. For instance, we can specialize the hierarchy for a particular framework, as component programming (as defined in [FBBR97]); see Fig. 2 for an excerpt. Also, the hierarchy may be specialized to deal with a particular step of software development, as requirements elicitation or component implementation. Different class hierarchies may coexist in a process database, which can be combined or enlarged later.

Fig. 1. Default generalization hierarchy

Fig. 2. Extending the hierarchy for the component programming framework (an excerpt)
2.2 Association relationships

It is clear that the classes presented above must be related beyond the generalization relationship, and this is done by means of the UML association relationships (*associations* for short). The associations are semantic links between classes that result in connections between their instances. Using them, we can state which documents are manipulated by which tools, which tasks use which resources, and so on.

Fig. 3 summarizes a few important associations bound to the default classes. Most of them are straightforward. In particular, we highlight the existence of an association (in fact, an UML *aggregation*) from tasks to tasks that catches the concept of task decomposition; this issue will become crucial in the dynamic part of the model.

![Diagram showing associations between classes](image)

Fig. 3. Some associations between classes

In addition to them, new relationships may be defined when extending the model. A quick look at fig. 4 reveals some of them for the component programming framework: binding component specifications with implementations, compilers with component implementations, etc.

2.3 The metaprocess

An important feature of PMLs must be the ability to reason about themselves in an uniform way, i.e., considering a SPM as a product to be managed using the very PML. This is known as *reflection*. We deal with this characteristic in a twofold manner: defining a metalevel and introducing a class *Model* in the hierarchy.
The metalevel appears in the static part of the PML when introducing (meta)classes enough to model the many facets of software process modelling. These metaclasses are integrated in the hierarchy (see fig. 5). To catch the flavour, table 2 describe two metaclasses that are of interest in the static part of the PML, Class and Attribute; similar ones for methods, parameters, associations, and others (including elements that appear in the dynamic part, such as precedences and triggers) are defined.

Unlike other approaches, PROMENADE considers the SPM itself as a class in the hierarchy, heir of Metaelement, which makes it easy the reuse and combination of SPMs. The class (fig. 6) has mainly two different kinds of attributes: on the one hand, the hierarchy of the model itself (attributes Hdoc to Hrol), which defines the layout of the elements that are involved in the model; on the other hand, the instances of the classes in this hierarchy (attributes agents to roles), i.e., the tasks, documents, roles,
etc., that configure a particular enactment of the model. The class invariant, which is put in a UML note with stereotype "invariant", takes care that this two kinds of attributes are properly coupled. Fig. 6 shows the highlight of the UML class.

<table>
<thead>
<tr>
<th>Table 2. Some predefined classes in the metalevel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td><strong>Agent; Task; Specification</strong></td>
</tr>
<tr>
<td><strong>TaskStatus; RoleIdentifier</strong></td>
</tr>
</tbody>
</table>

Fig. 6. The Model class

3 The Dynamic Part: The Intramodel level

The dynamic part is in charge of establishing the order in which process tasks can be enacted. In our approach, we combine two different paradigms of behaviour specification. On the one hand, we establish precedence relationships to schedule properly the enactment of the model; on the other hand, we allow the definition of triggers and exceptions that can interrupt at any moment this default behaviour. Both elements are modelled in the PML using different mechanisms, which layout is defined in the generalization hierarchy as metaclasses.

3.1 Tasks insights

First of all, let's take a more detailed look inside tasks. As a first relevant property, tasks can be atomic or composite. In the first case, there is an attribute that specifies the location where the code (a program, a shell script, or the creation of a tool session) of the task is. When it is composite, an alternative attribute contains the refinement of
the task. Task refinement is itself a (sub)model, consisting thus of a static part, which can extend the generalization hierarchy; and a dynamic part, which can contain more composite tasks, refined in other parts of the model.

Another important characteristic of tasks is the declaration of the involved documents as attributes, which can be seen as parameters (or messages) of the task. An important part of precedence specification will be the coupling of the parameters of the tasks involved in the precedence. For instance, the task `ImplementComponent` has as (output) parameter a document of class `Implementation`, and such a document acts also as (input) parameter of the task `TestComponent`; the precedence will be on charge of coupling both parameters.

A last common characteristic to all the (instances of the) tasks is that they evolve through a set of states following some transitions. The states are summarized in table 3. Note that a state can be reached either by the execution of a method bound to the `Task` class (which is invoked by another task instance), by the evaluation result of some predicate, or by an external input; in the table, we say that the requirements of a task are fulfilled when both its precondition is satisfied and all the precedences concerning the enactment of that task hold.

<table>
<thead>
<tr>
<th>Table 3. Task states</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>Idle</td>
</tr>
<tr>
<td>ReadyToStart</td>
</tr>
<tr>
<td>Active</td>
</tr>
<tr>
<td>Interrupted</td>
</tr>
<tr>
<td>ReadyToEnd</td>
</tr>
<tr>
<td>CompletedSucc</td>
</tr>
<tr>
<td>Completed</td>
</tr>
<tr>
<td>UnSucc</td>
</tr>
<tr>
<td>WaitTo</td>
</tr>
<tr>
<td>Reexecute</td>
</tr>
</tbody>
</table>
Fig. 7 shows the transitions between the states. Of course, the ultimate responsible of a transition being effective is the enactment engine, which controls the evolution of the whole process enactment. Note that there are two valid final states. With an extra attribute, the process engineer can decide whether a task may end in an unsuccessful state, allowing CompletedUnsucc to really be a final state, or not, in which case this state is no longer a final one and then the transition from CompletedUnsucc to Idle must be done.

![Task state transition diagram](image)

**Fig. 7.** Task state transition diagram

### 3.2 Precedences

Precedences are the means for representing proactive control specification in PROMENADE, i.e., specification of the execution order of process activities. Many other approaches, as JIL, implement this kind of control in an imperative way, but we prefer the declarative style, which seems to fit better to the flexible nature of process modelling.

The layout of precedences is defined through an UML metaclass which owns as main attributes:

- Type of precedence. In many different situations, the concrete way to relate tasks may vary:
• Strong precedences. There is a strong precedence from task \( s \) to task \( t \) if the completion of \( s \) is necessary in order to enact \( t \).

This is the usual kind of precedence when considering the same abstraction level. A typical example (fig. 8) is a strong precedence from the Specify Component task to the ImplementComponent one.

• Weak precedences. There is a weak precedence from task \( s \) to task \( t \) if \( s \) must be started before \( t \) and \( s \) must end before \( t \). Note, however, that provided that both conditions are satisfied, \( s \) and \( t \) can be enacted at the same time.

A typical situation (fig. 8) could be a weak precedence from EditImplementation to CompileImplementation: compilation of a new implementation version can take place from the very moment in which the implementation is modified, without forcing the implementation to be complete; on the other hand, it is necessary for the compilation task to finish after the editing one.

• Synchronizing precedences. We distinguish two different types. There is a start synchronizing precedence from task \( s \) to task \( t \) if \( s \) must be started before \( t \). There is an end synchronizing precedence from task \( s \) to task \( t \) if \( s \) must end before \( t \). Note that weak precedences are the sum of start and end synchronizing precedences.

A start synchronizing precedence can be stated from the DevelopProgram task to the UpdateComponentLibrary task; updating may take place as soon as components start to be delivered, and may even finish before ending program development.

• Document coupling. This makes the association of parameters of the involved tasks. When no ambiguity arises, documents of the same type are coupled by default.

<table>
<thead>
<tr>
<th>precedence</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>from SpecifyComponent</td>
<td>from EditImplementation</td>
</tr>
<tr>
<td>to ImplementComponent: strong</td>
<td>to CompileImplementation: weak</td>
</tr>
</tbody>
</table>

**Fig. 8.** Examples of precedences

### 3.3 Groupings

Groupings are an alternative way to introduce precedences between tasks. This construct deals with the situation in which many tasks should be considered as a whole: a grouping \( G \) of a set of \( n \) tasks establishes that, once a task from \( G \) is enacted, the process must complete all of them before starting any other task (outside \( G \)).

Note that grouping does not state anything about the order of execution of the tasks in \( G \); this can be done using precedences. Thus, the grouping does not oblige neither
to complete a task before starting others of g nor the other way round (for instance, the n tasks may be active simultaneously if precedences allow this situation).

Fig. 9 shows two examples of groupings. The one at the top forces functional and non-functional specification of a component to be performed as a whole. The other states that all the functional specifications of imported components must take place indivisibly. Notice the use of the user-defined `ImportedBy` association; the predefined `Self` construct refers to the current instance of the task. Also we remark that task parameters appear explicitly when they become essential for stating things in the model.

```plaintext
grouping [FuncSpecifyComponent, NonFuncSpecifyComponent]

for all M: Component in ImportedBy(Self::spec_comp)
  {FuncSpecifyComponent(M)}
```

**Fig. 9.** Examples of groupings

It is easy to prove that groupings can be expressed in terms of precedences. However, we keep them in the language for understandability purposes.

### 3.4 Dynamic precedences

Precedence relationships are part of the model, hence they must be introduced at model definition time. Some of them, however, may only be determined at enactment time, when the model is instantiated in a particular context.

Consider for instance the following precedence relationship: *in order to validate a functional specification of a component, it is required to validate every individual operation of that component.* It is clear that we do not know in advance how many operations a component will have.

To cope with this limitation, we allow the definition of *dynamic* precedence relationships. A dynamic precedence stands for an unknown number of "normal" precedence relationships; the concrete number will be fixed during enactment, depending on the value of an expressions given in terms of one or more attributes. For instance, fig. 10 shows the PROME:NADE code for the previous sentence; `spec_comp` is an attribute of the task (of class `SpecComp`, i.e. specifications of components).

```plaintext
precedence dynamic
  for all op: Operation in Operations(Self::spec_comp)
  from SpecifyComponentOperation(Self::spec_comp, op)
  to ValidateSpecification(Self::spec_comp):
  strong
```

**Fig. 10.** An example of dynamic precedence
3.5 Triggers. Exceptions

Up to now we have focused on those constructs available for defining the proactive part of the process. However, in order to define models complete enough, we need to add constructs for their reactive part, making thus the model sensitive to internal changes and external stimuli. This is the goal of triggers.

A trigger consists of three parts:

- Definition. A trigger must be classified as external or internal. In the first case, it is raised explicitly by the process engineer using the process assistant tool (e.g., because a specification should be revised due to changes in requirements). In the second case, the trigger is signaled because a predicate is fulfilled; the predicate involves attributes and associations from the static part (e.g., document version number, deadline date, responsible name) and some standard functions (e.g., \textit{CurrentDate}).

- Destination. The task or tasks that must handle the trigger. If the task is not active (or interrupted), state transitions enough should be made to enter this state; if the appropriate instance of the task does not currently exist, it must be (re)created. Of course, reexecution of a task will impact on other tasks due to precedences. Two particular destination tasks are \textit{Self} and \textit{Parent} (the last one, only for internal triggers, refers to the supertask of the task that caused the action of the trigger).

- Behaviour. The actions to be taken when the trigger is raised. This is to say, what happens with the tasks currently enacted, and the documents currently manipulated. The behaviour can involve methods of tasks; in fact, the whole behaviour can be just a method call.

In fig. 11 we show three examples. The \textit{ChangeOnRequirements} trigger forces the \textit{Specify} task to be reexecuted, and the method \textit{DealWithChange} is invoked, surely in charge to make cleanup. \textit{VersionGap} is triggered when there is a difference of more than one version number in two documents; the behaviour is in the body of the trigger. Last, \textit{TeaTime} takes place at five and does not alter the state of the process.

<table>
<thead>
<tr>
<th>trigger</th>
<th>ChangeOnRequirements</th>
<th>trigger</th>
<th>VersionGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>external</td>
<td>to Specify</td>
<td>internal</td>
<td>when</td>
</tr>
<tr>
<td>method</td>
<td>Specifiy::DealWithChange</td>
<td>exist c1, c2: Component such that</td>
<td>c1::VersionNb - c2::VersionNb &gt; 1</td>
</tr>
<tr>
<td>end trigger</td>
<td></td>
<td>to Parent</td>
<td>method ...</td>
</tr>
<tr>
<td>trigger</td>
<td>TeaTime internal</td>
<td>end trigger</td>
<td></td>
</tr>
<tr>
<td>when</td>
<td>CurrentTime = 17:00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Same</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>method</td>
<td>Display(“Time for tea :)”)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end trigger</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Fig. 11.} Examples of triggers
A particular case of triggers are exceptions. An exception is an internal trigger which predicate is the negation of the success condition: it is raised when the enaction of the task ends in failure. Every task that may fail at the end, can define this special kind of exception. They are included in the very body of the task. If the task is not allowed to end in failure, the destination task of the exception will be the task itself. Otherwise, the destination task may be any other one (e.g., from TestComponent to ImplementComponent).

The main differences with respect to triggers are:
- They can include what we call feedback parameter, a document that contains the reason of the exception (e.g., a report about component test failure, or a notification of compilation errors).
- Error propagation exists from component tasks to their supertasks.

### 3.6 Precedence graphs

Precedence graphs are the underlying representation of the dynamic part of SPMs. Their usefulness is twofold: on the one hand, they serve as a graphical tool to create and visualize precedences, exceptions and triggers; on the other hand, they provide a formal model to reason about the correctness of SPMs.

In a precedence graph, nodes are tasks and edges establish some kind of relationship between them. In Table 4 we summarize the types of edges, coming from the elements already presented. The first three types correspond to strong and synchronizing precedences, while feedback precedences come from exceptions and triggers; weak precedences and groupings are not necessary since they can be defined in terms of the previous ones. External triggers appear as edges with no source. Every type of edge can also be classified as dynamic or not; in the drawings, we mark dynamic edges with a star.

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Appear in</th>
</tr>
</thead>
<tbody>
<tr>
<td>end-start</td>
<td>$s \rightarrow t$</td>
<td>Strong precedences; groupings</td>
</tr>
<tr>
<td>feedback</td>
<td>$s \rightarrow t$</td>
<td>Exceptions and triggers</td>
</tr>
<tr>
<td>start-start</td>
<td>$s \rightarrow t$</td>
<td>Start synchronizing precedences; weak precedences; groupings; subtask decomposition</td>
</tr>
<tr>
<td>end-end</td>
<td>$s \rightarrow t$</td>
<td>End synchronizing precedences; weak precedences; groupings; subtask decomposition</td>
</tr>
</tbody>
</table>

It should be noted that start-start and end-end edges appear also in the context of task decomposition: there is a start-start edge from a supertask to its initial subtasks (i.e., those with no predecessors), and an end-end edge from the final subtasks to their supertask (i.e., those with no successors). As a result, component subtasks may not start until the composite supertask itself starts, and the composite supertask may not end
until the component subtasks end. The integration of task decomposition with
dynamic elements makes the underlying model simpler.

Although graphs are bound individually to refinements, it is worth noting that
they compose naturally, yielding to a single precedence graph which rules the whole
enaction process.

Fig. 12 shows an example of precedence graph for a refinement of the Specify-
Comp task.

\[ \text{Fig. 12. A precedence graph for the SpecifyComp task} \]

4 The Dynamic Part: Modularization Aspects

We think that one of the most important features of PROMENADE is the support
provided for reusing, combining, enlarging and refining (parts of) SPMs and tasks.
This ability has become crucial for dealing with complex models, which can be thus
built from small pieces, refined stepwise of even let incomplete until enaction.

As we have already pointed out, the layout of the static part (as a generalization hi-
erarchy) is a means for supporting reuse and refinement. In the rest of this section, we
focus on three additional features provided by the PML.

4.1 Model combination

The need for combining different models appears when SPMs that involve different
elements and functionalities must be put together. Typically, this happens for models
which cover different parts of software development, as specification, design or im-
plementation: each of the corresponding SPMs manages its own documents (although
probably some of them will be shared), and they will apply their own tasks.

To be more precise, given \(n\) models \(P_1, \ldots, P_n\), we define its combination, written
\(P_1 + P_2 + \ldots + P_n\), in the following manner:
• The static part of $P_1 + P_2 + \ldots + P_n$ is the superposition of their generalisation hierarchies, together with the union of their association and aggregation relationships. Of course, hierarchies must be compatible.

Name clashes (of classes, relationships, attributes, etc.) can be avoided through renaming, which can also be used to make equivalent elements that appear with different names in the combined models.

• For the dynamic part, precedences are established from final tasks to initial tasks, without considering further decomposition of composite tasks.

Fig. 13 shows model combination applied on the specification, design, implementation (including testing) and tuning steps of whole applications. We call this Single Development, to make clear that maintenance is not included. There are strong precedences from Specification to Design, and from Design to Implementation; coupling from Implementation to Tuning is made through a weak precedence. Renaming takes place to handle the two different kinds of specification documents, the one for application specification and the other for component specification (we assume that specification of components is already named SpecComp in the Design model). Default document coupling is applied everywhere.

```
spm SingleDevelopment
  combines Specification + Design + Implementation + Tuning
  with precedences
    from Specification to Design: strong
    from Design to Implementation: strong
    from Implementation to Tuning: weak
  renaming
    Specify::Specification to Requirements
    Design::Specification to Requirements
    Implementation::Spec to SpecComp
  ...
end spm
```

Fig. 13. An example of model combination

Once combination has been made, new elements (triggers, ...) can be added to the resulting model; this is what the dotted points in the previous figure stand for.

### 4.2 Task refinement association

One of the most important drawbacks in most existing PMLs is their lack of flexibility when refining composite tasks: they fail in supporting alternative refinement of tasks to coexist in a single SPM. In our opinion, this a crucial property that PMLs should exhibit.

For instance, let’s take the task of writing a functional specification of a component. The concept of functional specification and/or the process to obtain it may vary
in different applications of the same organization and even in different places of the same application: specifications could be usually conceived to be mostly informal, but they can be forced to be formal in critical components. Thus, both the type of involved documents as well as the process of writing the specification are different.

Task refinement has been introduced in section 3.1 as the decomposition of a composite task into many component subtasks, which interact via precedences, exceptions and triggers. From now on, we assume that a composite task may have more than one refinement, each one with a different name. These refinements are to be stored in the process data base, ready to be used whenever needed. The choice of refinement is made explicit with a new construct of the language.

Coming back to the functional specification example, we can create many refinements for the FuncSpecifyComponent task: just explanations, model-oriented specification, algebraic specification, etc. When using FuncSpecifyComponent in a particular context (e.g., in the refinement of another composite task), we can write the sentence:

\[ \text{FuncSpecifyComponent refined by ModelOrientedSpecification} \]

to use the second choice of refinement

A problem may arise when considering transitivity. Obviously, any task appearing in the chosen refinement may in turn be composite, and it could also have different refinements, from which one should be chosen. This would happen when choosing Z as specification language, obtaining:

\[ \begin{align*}
\text{FuncSpecifyComponent} \\
& \quad \text{refined by ModelOrientedSpecification} \\
& \quad \text{refined by ZSpecification}
\end{align*} \]

Luckily, from our case studies it seems that not many imbrications are usually required.

A very attractive side-effect of allowing alternative refinements is its usefulness for handling incomplete models, i.e., SPMs which have some parts not entirely determined. This kind of incompleteness is nice in various contexts. In the last example, one could argue that the choice of a concrete specification approach should be made individually, depending on the nature of every component. In this case, the refinement should not be chosen at modelling time; instead, during enaction, the engineer would be required to select among the different existing options. In fact, this feature is closer to the concept of software process modelling as an assistant tool, more than a nearly fully automated tool, which seems preferable from the flexibility point of view.

As a last remark, notice that not making selection explicit alleviate the problem of transitivity mentioned above.

### 4.3 Combining different task refinements

A composite task refinement is the result of applying a strategy for carrying out the task. Sometimes, the strategy is the combination of many different tactics, and one
could wonder if this combination could be made easy just by putting together existing refinements for those tactics. Reusability is thus supported again, as well as understandability because small refinements can be bound to very concrete criteria.

For instance, consider again component specification. Among others, two different factors that impact on this task are: 1) precedence between functional and non-functional parts, and 2) precedence relationships between the specification of a component and the specification of the components it involves. Fig. 14 shows in the upper part two different task refinements; the one at the left states that functional specification should take place before non-functional one; the one at the right establishes top-down component specification. SpecifyComponent(M) stands for an instance of the SpecifyComponent task applied to the document M. So, a new refinement can be defined combining both of them, as showed in the lower part of the figure.

```
 refinement FunctBeforeNonFunct for SpecifyComponent  
    subtasks FuncSpecifyComp  
    NoFuncSpecifyComp  
    with precedences  
    from FuncSpecifyComp  
    to NoFuncSpecifyComp:  
    strong  
 end refinement

 refinement BottomUp for SpecifyComponent  
    with precedences dynamic  
    for all M: Component  
    in ImportedBy(Self::spec_comp)  
    from Self  
    to SpecifyComponent(M):  
    strong  
 end refinement

 refinement MyStrategy for SpecifyComponent  
    combines FunctBeforeNonFunct + BottomUp  
 end refinement
```

Fig. 14. Combining different task refinements

The combination of refinements, thus, overlaps the involved refinements to create a new one. As in model combination, renamings are allowed, and sometimes required to integrate things properly. Also, some compatibility conditions (involving not only the dynamic part, but also—and mainly—the static one) must hold for the combination being correct.

5 Conclusions and comparisons with other work

A process modelling language called PROMENADE, defined in the framework of the ComProLab project [FBBR97, FR98], has been presented. It consists of two different parts: the static one, to introduce that elements that play a part in the process; and the dynamic one, which establishes the order in which tasks are enacted.

The static part is designed using UML, and it integrates elements enough to define a metalevel. The dynamic part combines proactive and reactive controls by means of:
different kind of precedences between tasks; triggers for making the model sensitive to internal conditions and external stimuli; exceptions for taking actions on anomalous situations. Those elements are combined in a precedence graph, which condenses all the information that rules process enactment. To make the model more structured and to support reuse and stepwise refinements, different constructs for modularity are included in the language.

We have not presented here neither a complete definition of the language, nor formal definitions nor correctness aspects; we have preferred to introduce most of the elements of PROMENADE, with the help of a few examples. Also, despite being part of the project acronym, we have not really given any hint about enactment; in fact, the enactment process, including: assistant tool, engine, versioning, process data base management, and distributivity concerns, is just starting to be built. However, even without this enactment part, we think that the current process language is useful enough as a process model support.

We think that the most relevant features of our approach are the following:

- We use a widespread notation, UML, to define the static part, reducing thus the complexity of the language. UML has currently become a standard di facto, and this situation together with its adequacy for our purposes, clearly justifies our choice. In fact, we strongly believe that one of the aspects for which many interesting approaches to software process modelling have not encountered a better response in the software engineering community has been the fact that most of them have failed in using standard elements.

- We combine two different paradigms in the dynamic part, for managing both proactive and reactive controls. The combination of these complementary styles makes model definition simpler. Most of existing systems use only one kind of control (e.g., APEL works reactively; Merlin, proactively). Only a few of them combine both kinds of control (remarkably, JIL). Furthermore, the PROMENADE proactive control flow is declarative instead of the usual imperative approach. This means that a model written in PROMENADE does not force any particular enactment sequence but only establish certain key dependencies between tasks that should be preserved by any model enactment.

- We think that we have got a good tradeoff between simplicity and expressivity. Features like: different kinds of precedences, dynamic precedences, exception handling (including the feedback parameter), model and task combination, and alternative refinements, make the language powerful without being too difficult to manage (we think!)

- Both the definition on the static part as an inheritance relationship (which in fact is not new -just the use of UML is) and the intensive use of structuring mechanisms, makes our approach suitable for reusing, combining, adapting, enlarging and refining process models. This has been been pointed out as one of the key features of the second generation of PMLs [WBG+98]. Although the use of inheritance is a common feature in most existing PMLs, it is not so usual to find structuring mechanisms as the ones presented here in the dynamic part.
• The definition of the precedence graph gives an unified and compact view to the whole model, integrating proactive and reactive elements, as well as task decomposition, in a single framework. Its existence is a great help for us in defining the language and, especially, in defining correctness concerns and the enactment process.

• We allow incomplete models to exist, using the concept of alternative refinements. We firmly believe that incomplete models are essential to reflect the very nature of process modelling.

• By the way, in spite of not being presented in the report, we would like to remark too the ability to define both the correctness of a process model, and the correctness of a particular development with respect to a process model, which provide a powerful way to reason about process modelling. We do not know of any approach having complete results in this issue.

As future work concerning the language, we think that it mostly must focus on the management of model evolution: models are not fix in time, but they evolve to fit to changing environments, even when some applications may be enacted using them. A correct management of this feature is clearly a challenge of any modelling language. Also, we are working in the definition of an accurate data-flow model to specify better the asynchronous flow of data between tasks.

References


