PROMENADE: A PML Intended to Enhance Standarization, Expressiveness and Modularity in Software Process Modelling♠

Josep M. Ribó[1], Xavier Franch[2]

[1] Universitat de Lleida
C. Jaume II 69, 25001 Lleida (Catalunya, Spain)
fax: +34.973.70.27.02
josepma@eup.udl.es
[2]Universitat Politècnica de Catalunya (UPC),
c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
fax: +34.93.401.70.14
franch@lsi.upc.es

Abstract. Some limitations have been identified in current software process modelling languages (SPML) concerning *expressiveness*, *standarization*, *modularity* and *flexibility*. We present a SPML called PROMENADE which aims at improving the above-mentioned features. Expressiveness is enhanced in PROMENADE by the use of a basic catalogue of precedence relationships (instead of transitions), the ability to define derived precedences using a high-level notation and the existence of dynamic precedences intended for relationships that are not known until process enactment time. Standarization is achieved by the mapping of PROMENADE features into a slightly extended UML. PROMENADE allows also the construction of complex models by the combination of partial ones using high-level operators.

1 Introduction

A model for a software development process [DKW99] (i.e., a *software process mode, SPM*) is a description of this process expressed in some *process modelling language (PML)*. 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 executed (*enacted*). The systematic description of software processes not only helps in understanding software develop-

♠ This work is partially supported by the spanish research programme CICYT under contract TIC97-1158.

ment, but also makes feasible the construction of systems for supporting automation of the process up to an acceptable level.

During the last fifteen years, many approaches dealing with software process modelling have been proposed. As a result of this important research effort, some remarkable features have been attained: object-orientation has been introduced as a natural way to model the structural part of the process, while achieving a certain degree of reuse and modularity; the abstraction level of systems has increased, which has made them easier to use; several different paradigms have emerged leading to a wide variety of different approaches, which have shown their usefulness. Each one of these approaches proposes its own language for modelling the process. As we can see in [FKN94, DKW99], the variety of such languages is enormous, ranging from existing formalisms coming from other fields (as EER, Petri Nets, ECA-rules, state-transition diagrams and so on) to *ad hoc* proposals. Some approaches combine more than one formalism, both for covering the static and dynamic parts (i.e., the conceptual model and the behaviour, respectively) and for dealing properly with different aspects of the software process, as reactive and proactive controls, or user interface.

In spite of the notable results achieved by the software process (SP) research community, there still exist several features that current approaches fail to manage in an appropriate way, being, in our opinion, the most important ones: the lack of expressiveness and standarization of most approaches; the insufficient means to construct a SPM in a modular way (i.e. from partial SPM already contructed); and the lack of flexibility in its enactment.

The purpose of this report is to present the most important features of a PML called PROMENADE (**PRO**cess-oriented **M**odelling and **ENA**ctment of software **DE**velopments)[1] intended to be a part of the *second generation of PMLs* (as it has been named in [SO97]) and to make some contributions to the improvement of the referred issues. Remarkably, PROMENADE is based on UML notation, making it more attractive for software engineers (not another formalism to be learned).

Section 2 shows the reasons for which PROMENADE has focused on those features. Section 3 outlines some static aspects of PROMENADE (including an overview of its metamodel). Section 4 presents the statechart that controls the tasks life cycle. Section 5 shows the double decomposition of tasks in PROMENADE and how this decomposition will help in the achievement of flexibility at enactment time. Sections 6 and 7 deal with the behavioural part of a PROMENADE model: they present the notion of *precedence relationship* and provide a notation to define them. Section 8 is devoted to standarization: it presents how precedence relationships may be modelled in UML. Finally, section 9 focuses on the modular construction and reuse of SPMs.

[1] PROMENADE is defined as a part of a more general research project *ComProLab* (A Component Programming Laboratory), see [7].

2 Motivations of our approach

We devote this section to explain why we believe that the above-mentioned issues (namely *expressiveness, standarization, modularity* and *flexibility*) are not well achieved in current approaches and to outline our ideas concerning them. See [RF00] for a more detailed study of these issues, including a comparison between some relevant approaches.

Expressiveness:
The need for a flexible and expressive modelling of process control-flow, together with the ability to define new control-flow dependencies, has been recognized in the last few years in the field of workflow management (see, for instance, [JB96, JH99, RD98]), but, in our opinion, it has been somewhat neglected in the related area of SPM.

In spite of the very nature of SPs (which involve a loose control and a high degree of collaboration) most of existing approaches lead to very prescriptive models which do not conform to the way in which software is actually developed.

As it is shown in table 1, existing SPM approaches range from the low-level first generation languages to the more recent ones (APEL, E3, Little-JIL, Peace+...) that have provided some higher level mechanisms to improve comprehensibility. However, we believe that they still fail in achieving the level of expressiveness needed for a realistic SPM (i.e. in most cases the control flow constructs are very basic and unable to model complex dependencies. See section 6 for an example).

The contribution intended by PROMENADE to this matter is to model a software process by stating in a declarative way the precedence relationships existing between the different activities taking part in that process. A basic set of such precedence relationships has been identified for this purpose. PROMENADE also provides a high-level notation to define new precedence relationships derived from the basic ones and a formalism to describe precedences that are known only at enactment time.

Standarization:
The intensive SPM research that has been developed in the last few years, on the one hand, has stimulated progress in the field but, on the other hand has lead to a proliferation of languages and notations that have hampered the wide use of the software process technology within the software engineering community. One could wonder if it is time for the community to adopt a linguafranca and, in this case, which should be the chosen formalism.

An obvious candidate is UML [RJB99]. UML is becoming a standard modelling notation in the field of software engineering, both in the academy and in the industry. Since SPM can be seen as a particular domain of software engineering, it is reasonable to wonder about the adequacy of UML for expressing software processes.

As far as we know, not much research work has been carried out in this matter. Some proposals do exist [Rata] to extend the application range of UML (mainly by defining a set of stereotypes) which, in our opinion do not seem to be satisfactory enough (see section 8 for a more detailed discussion). Some studies suggest that, although UML is powerful enough to deal with the static part of a software process, the notations offered to express its behaviour (in particular activity diagrams) suffer from several limitations [AL98, FR99a, FR99b, McL98].

Our proposal is to extend UML in such a way that it becomes a suitable software process modelling language. Unlike other approaches, we propose extending UML by: (1) adding new "real" metaclasses (specific to SPM) to its metamodel and (2) representing precedence relationships by means of UML dependencies. From our point of view, both issues represent an improvement to current research results.

Modularity

If *software processes are software too* [Ost87],it makes sense to apply the notion of reuse to SPM. In fact, one of the most challenging issues in SPM is the ability to incrementally (modularly) construct a model by combining existing ones. This leads to a bottom-up approach which so far has not been completely attained by existing systems. To our knowledge there are not many approaches that aim at constructing models modularly by defining some methodology to reuse and compose (integrate) existing submodels. With the exception of [Chr94], which gives some guidelines to define operators to combine models; all the approaches that follow this direction we are aware of rely on the notion of *view.*

Views suggest a top-down approach to SPM; i.e. they seem a good way to decompose a model into simpler ones in order, for example, to restrict the part of the process available to a given role or to show a more comprehensible model by focusing on one of its aspects. However, since views must be derived from an existing model, we believe that they are not the most natural way to perform model composition from submodels, which is clearly a bottom-up approach. Sometimes a view composition operator is provided (see, for example, [AL98]). But usually, this is limited to a superposition of existing views. More powerful and high-level operators to combine submodels in specific and high-level ways would be valuable.

PROMENADE considers several levels of reuse, namely, (1) reuse of primary model components, (2) derivation of model components and (3) composition-combination of partial models. Clearly, the last is the most challenging and, in our opinion, the less achieved one. PROMENADE also takes into account the correctness problems that arise when a model is reused.

Flexibility

Most initial approaches to SPM (specially the activity-oriented ones) tended to be too prescriptive and did not deal with the inherent evolutionary nature of both software and software processes. Two general solutions to this problem have been provided by different systems, namely (1) to allow *changes* of the enacting process *on-the-fly* and (2) to provide means to refine parts of the model during enactment.

PROMENADE chooses the second alternative and defines hierarchies of task refinements (i.e. different ways to realize an activity). This is different from other approaches that consider only one level of activity realization [HJK96]. At enactment time one of those refinements is to be selected (or a new refinement can be created). Sometimes this is called a flexible enactment of a model.

Table 1: Some existing SPML approaches regarding expressiveness, standarization, flexibility

Feature	Existing SPMLs	PROMENADE
Expressivity	Pure reactive control-flow (Adele [FKN94]), basic behavioural features (Merlin [FKN94]), Petri-nets (SPADE [BFG94]), imperative programs (APPL/A [SHO95]), transitions (APEL [DEA98]), goal-orientation (Peace+ [ALO96])...	Different kinds of basic precedence relationships between tasks, ability to define derived precedences, dynamic precedences.
Flexibility	Several realizations for a single task [JH99], changes *on-the-fly* ([CLM95], [KD96]), goal-oriented approaches ([ALO96]).	A hierarchy of task refinements. One of such refinements may be selected at enactment time.
Modularity	Reuse of single model components [JPL98]. Model decomposition in *views* [AC96], [EHT97].	Reuse of any model component, high-level operators to combine exisitng partial models. Correctness concerns taken into account.
Standarization	Ad-hoc languages (most of the approaches), partial (stereotype-based) UML extensions [JSW99].	UML extension with an enhancement of UML-activity diagrams.

3 Static part of a model in PROMENADE

The part of the PROMENADE metamodel intended to deal with the static (structural) part of a SPM is built as an extension of the UML metamodel [Ratb], adding a metaelement for each one of the core elements that take part in any PROMENADE model (e.g., it adds a *MetaTask* metaelement for tasks, a *MetaDocument* metaelement for documents, a *SPMetamod* metaelement for the model itself, etc.). The model elements (e.g., the class *SpecifyComponent* for the task of specifying a component) are seen as instances of the metaelements (e.g., *MetaTask*) whose (meta)features (metaattributes and metaoperations) have been given a value. Therefore the process of building a model in PROMENADE will consist mainly in creating instances of the metamodel classes and give values to its metafeatures.

The PROMENADE approach to process modelling defines a universal or *reference model* (an instance of the PROMENADE metamodel) that constitutes the basis on which any other process model will be built (as done for instance in [DEA98]). Hence, this *reference model* is the common and extensible kernel shared by all processes modelled in PROMENADE. It is responsible for defining the core elements that will be a part of any model described in PROMENADE (e.g., the classes *Task, Document* and *Model* are defined by the reference model). The features associated to these elements are the ones needed to characterize any instance of them. Notice the difference between the metaelement features (which are used to characterize model classes) and the element features (which are used to characterize model class instances). Sometimes we may refer to the former group as *class features* and to the latter one as *class instance features.* Figure 1 shows the association relationships existing between the metaclasses that take part in the metamodel description. On the other hand, table 2 shows the main classes of the reference model. These elements are shown in more detail in [FR99b].

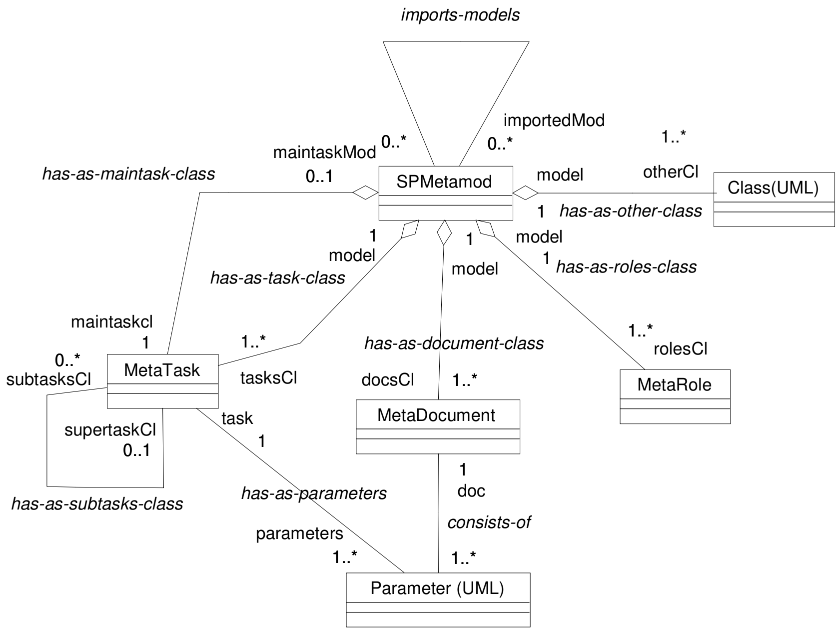

Figure 1. Associations of the PROMENADE metamodel

Table 2. Predefined classes in the PROMENADE reference model

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

4 Task structure

Tasks (or activities) are structural elements that have a major importance in the modelling of the dynamic part of a SP. They are represented in PROMENADE by means of the class *Task* (which is an instance of the *MetaTask* metaclass). This is the class of the PROMENADE reference model which acts as an abstraction of any important action (atomic or composite) that is to be performed during the process of software development. In this section we will outline some task attributes and operations which are important in order to describe the behaviour of a process (see [FR99b] for more details). We will also present tasks life cycle by means of a statechart diagram.

The main properties of tasks are the following:

- *atomic* or *composite* tasks. In the first case, there is an attribute (*definition*) 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 (*precedences*) contains the task behaviour in terms of precedence relationships between its component tasks (subtasks).
- Task *parameters.* The documents that are involved in a task are declared as attributes, which can be seen as *parameters* of the task.
- Task *successCondition*. This is a predicate (written in OCL) that states the conditions that should be fulfilled at the end of a task to consider that it has finished successfully.
- Task *precondition*, which states a constraint that must hold when a task is chosen to be enacted (i.e. when all the precedence requirements are fulfilled).
- *Automatic* tasks. A task which is defined to both start and end as soon as it is allowed to. In contrast, non-automatic tasks start and end upon user's request (provided that they are ready to be enacted).
- *Model maintask*. This is a composite task that encapsulates the model behaviour.

In addition, all tasks have some operations defined:

- *init(),* which is executed when the task instance is created,
- *start(),* executed at the starting point of a task instance,
- *end()*, executed when the task ends its execution, and
- *reinit(),* when a task must be reexecuted.

Further operations and attributes may be defined for a task class.

Task instances evolve through a set of states following some transitions as it is shown in figure 2. The states are summarized in table 3.

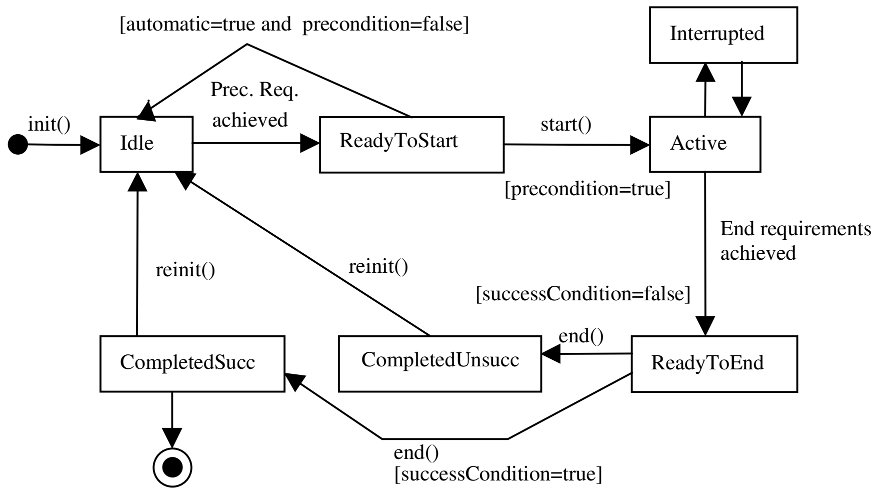

Figure 2: Statechart for a task

Table 3. Task states

State	Description	Arrival on...
Idle	The task has been created, but it is not ready to (re)execute (its precedence requirements are not fulfilled)	Execution of the methods *init()* or *reinit()* on the task
ReadyToStart	The precedence requirements to start the task are fulfilled, but the task has not been enacted yet.	The fulfillment of the prece-dence requirements to start the task
Active	The task has been enacted and it is currently in execution (i.e., it consumes some resources)	Execution of the method *start()* on the task and the task precon-dition holds; also resuming an interrupted task.
Interrupted	The task has been enacted, it has not finished yet but it is currently not in execution	External stimulus to the task (resource not available, logout..)
ReadyToEnd	The task has completed its mission (the precedence requirements to end it hold).	The fulfilment of the precedence requirements to end that task.
CompletedSucc	The task has ended successfully	Execution of the method *end()* and successful condition true
Completed Unsucc	The task has ended in failure	Execution of the method *end()* and successful condition false

5 Task decomposition

Tasks may be decomposed along two dimensions that turn out to be orthogonal: task decomposition *by aggregation* and *by refinement*. This section focuses on these two aspects, which are part of the static model.

5.1 Task decomposition by aggregation

Tasks in PROMENADE may be of two kinds according to the complexity of their behaviour: *atomic* tasks and *composite* ones. Atomic tasks cannot be decomposed in other tasks. Their enactment is performed by means of a call to an external tool or in a manual way. On the other hand, composite tasks may be further decomposed into *subtask classes*, which may be in turn atomic or composite. Therefore, a composite task enactment may involve the enactment of several *subtasks* in some suitable order (determined by a set of precedence relationships between tasks. See section 6).

Some of these aggregation relationships concerning task classes are shown in the example of figure 3. This figure shows the task classes that are involved in the upper levels of the construction of the component library, which should be completed by tasks concerning lower levels. Specifically, some subtasks of *BuildComponent* and *SpecifyComponent* are shown. On the other hand, table 4 shows the parameters (documents) used and generated by some of these tasks along with a brief description. Notice in this table that in addition to the usual parameter modes (input, output and input/output), a feedback mode is introduced to represent the flow of information when a task ends in failure.

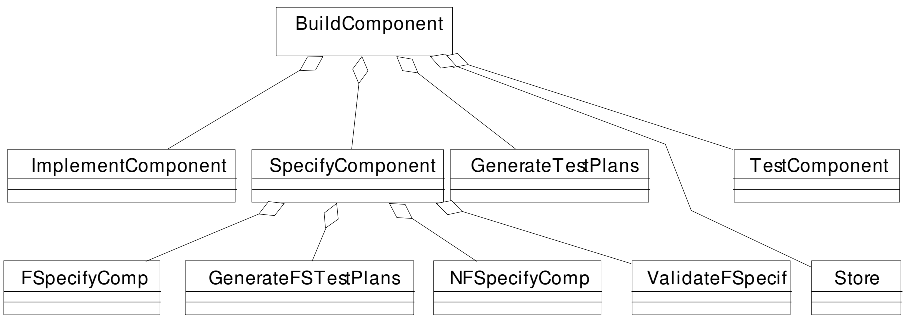

Figure 3: Aggregation relationships between tasks

Table 4. Task classes

Task	Parameters	Description
BuildComponent	b: **i/o** Library, c: **out** Component	Construction of a component
SpecifyComponent	c: **out** Component, sd: **in** SpecDoc	Specification of a component (both functional and non-functional)
ImplementComponent	c: **i/o** Component, id: **out** ImplDoc, evd: **in fbk** ImplEvalDoc, b: **in** Library	Implementation of a component.
GenerateTestPlans	sd: **in** SpecDoc, stp: **out** seq(ImplTestPlan)	Generation of plans for testing a component.
TestComponent	id: **in** ImplDoc, evd: **out fbk** ImplEvalDoc, stp: **in** seq(ImplTestPlan)	Test of a component according to its test plans.
Store	c: **in** Component, b: **i/o** Library	Storing a developed component into a library.
FSpecifyComp	c: **i/o** Component fsd: **out** FSpec evd: **in fbk** SpecEvalDoc b: **in** Library	Functional specification of a component (different task refinements may be considered.
ModelOrFSpecify	c: **i/o** Component fsd: **out** ModelOrFSpec evd: **in fbk** SpecEvalDoc b: **in** Library	One of the task refinements of the functional specification of a component. It describes a model-oriented functional specification of a software component.

5.2 Task decomposition by refinement

The behaviour of a composite task is encapsulated in PROMENADE by means of *task refinements*. Intuitively, a task refinement is a concrete way to perform a task. A bit more formally, a task refinement of a composite task class *T* is a task class that expresses one specific way in which *T* may be decomposed into subtasks and the precedence relationships that should be kept among them at enactment time. Since, in general, it is possible to think of several ways to perform a task, it makes sense to define several task refinements for a specific composite task.

Task refinements are modelled in PROMENADE by means of generalization relationships between a task class and the set of task classes that refine it. In this way, the subclasses of a task class *T* represent its possible refinements. Notice that task refinements and aggregations are two complementary mechanisms which will appear tightly intertwined in class diagrams: a task can be refined in many ways, and for each task refinement a particular decomposition into subtasks may exist, and those subtasks can be in turn refined in different ways, and so on.

Some examples of how a task class may be refined are given in figure 4. We consider two ways to carry out the functional specification of a component (*FSpecify-Comp*): *model-oriented functional specification* (by which a formal model is associated to a component and its operations are specified according to that model) and *informal functional specification* (which describes in natural language the component and its operations). Each one of these functional specification methods is considered to be a new task refinement in PROMENADE and it is represented by means of a generalization relationship.

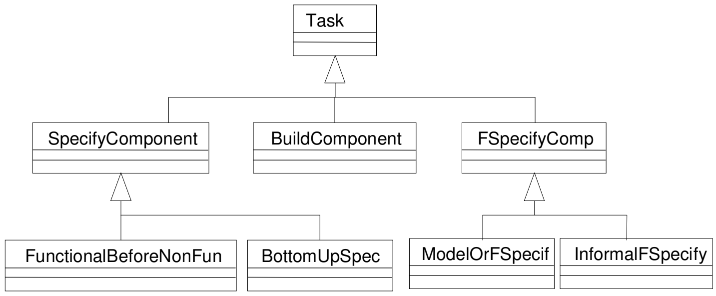

Figure 4: Generalization relationships between tasks (task refinements)

Usually, the selection of a particular task refinement enforces the use of some specific documents. This enforcement is established in the parameter definition of the refinement. For example, the selection of the *ModelOrFSpecify* task refinement (for the task *FSpecifyComp*) must result in the automatic selection of a *ModelOrFSpec* as functional specification document and a *ModelOrSpecTestPlan* as specification test plan and this is reflected in the *ModelOrFSpecify* parameters (see table 4). In general, a task refinement for task *A* may use as parameters some subclasses of the parameter classes of *A*.

The behaviour of a task refinement is described by means of a specific control flow which is depicted in a PROMENADE precedence diagram (see section 6). Other task refinements may come up in the activity diagram that describes a specific task refinement for a given task class *T* (including other *T*'s refinements).

Task refinements help in achieving *expressiveness* (since various different behaviours may be given to a single activity; hierarchies of task refinements are also allowed), *modularity* (task refinements are encapsulated into tasks; therefore it is easy to add /remove task refinements to/from a model), and *model flexibility* (since we can decide at enactment time, instead of modelling time, which specific task refinement is to be enacted; this allows the enactment of incomplete models).

6 Control-flow

PROMENADE provides both proactive and reactive control-flow mechanisms in order to model the behavioural (dynamic) part of a software process. Reactive control (i.e. the enactment of some actions as a response to events) is achieved by means of some operations associated to tasks that are triggered on the occurrence of a certain event (e.g. the change of the state of some object). The description of the reactive control-flow mechanism in PROMENADE is currently in progress.

Proactive control (i.e. the enactment of some actions according to a pre-established plan) in PROMENADE is managed by means of three different kinds of elements: (1) precedence relationships, (2) automatic tasks and (3) task preconditions.

Automatic tasks and task preconditions have been introduced in section 4. We will devote the remainder of the section to study the concept of precedence relationships.

6.1 Basic precedence relationships

Definition. A *basic precedence relationship* is a tuple:

<type,$\{s_1, ..., s_n\}$, $\{t_1, ..., t_m\}$, combi, parBinding>

where:

- *type* is the precedence relationship type (see below).
- $\{s_1, ..., s_n\}$, $\{t_1, ..., t_m\}$ are sets of tasks (the *source* and *target* of the precedence).
- *combi* is the combination form, used in precedences with more than one source task, which may be AND or OR.
- *parBinding* is a correspondence from the parameters of the source tasks to the ones of the target tasks. Parameters are documents, and this correspondence is the way to link the outputs and inputs of the tasks.

Basic precedence relationships allow the establishment of relationships between tasks, hence the formulation of models using concrete development plans. Their graphical representation appears in fig. 5 (p's and q's are the parameters).

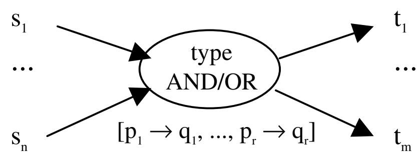

Figure 5: Representation of precedences

Although the most natural semantics of a precedence seems to be: target tasks cannot proceed until all (or one, depending on the combination form) of the source tasks have finished, real process modeling require other kinds of precedence relationships. For instance, it does not seem very realistic to model something of the sort *the implementation of a component will start once its specification has finished*. Actually, the trend is to overlap to a certain extent both activities. Hence, it seems better to model the previous example as: *the implementation of a component should begin some time after the starting point of its specification and should finish after the end of its specification.* Notice that this modelling provides not only a higher degree of freedom when enacting the model, but also a more expressive, realistic, flexible and less strict model. The predefined basic precedence relationships of PROMENADE are summarised in table 5. To be strict, *weak* precedences are not a basic type of precedences since they can be obtained from the combination of *start* and *end* synchronizing precedences. We have included them as basic precedence types because they come up very often in usual SPMs.

PROMENADE incorporates a formalism to define precedence relationships which is outlined in section 7.

Table 5: Different types of basic precedence relationships

Type	Notation	Meaning
start	$s \rightarrow_{start} t$	*t* may start only if *s* has started previously
end	$s \rightarrow_{end} t$	*t* may finish only if *s* has finished previously
strong	$s \rightarrow_{strong} t$	*t* may start only if *s* has finished successfully previously
weak	$s \rightarrow_{weak} t$	*t* may start only if *s* has started previously and *t* may finish only if *s* has finished previously
feedback	$s \rightarrow_{fbk} t$	*t* may be reexecuted after the unsuccessful end of *s*

6.2 Derived precedence relationships

The set of basic precedences we have presented has proved its usefulness in the case studies we have gone through. However, other precedence relationships can raise when modeling processes (hence, making resulting models more comprehensible and the language itself more user-friendly). Many of these new precedence relationships can be defined in terms of the basic ones.

Specifically, a derived precedence is a particular case of precedence that can be defined as a combination of basic precedence relationships. Derived precedences incorporate two features to the precedence definition:

Definition. A *derived precedence relationship* is a tuple:

$$\langle \text{type}, \{s_1, ..., s_n\}, \{t_1, ..., t_m\}, \text{combi}, \text{parBinding}, \text{definition}, \text{cond} \rangle$$

where:

- $\{s_1, ..., s_n\}$, $\{t_1, ..., t_m\}$, *type*, *combi* and *parBinding* as before
- *definition* defines the derived precedence in terms of the basic ones.
- *cond* is a predicate that is used in some kinds of derived precedences as a condition associated to the precedence.

Some examples of derived precedences are shown in table 6. Furthermore, figure 6 shows graphically how those derived precedence relationships can be defined in terms of the basic ones. Section 7 presents a high-level formalism to do this in a formal way.

Table 6: Some derived precedence relationships

Type	Notation	Meaning
Successful end-end	<$\{s_1,s_2,\ldots,s_n\}$, $\{t_1,t,\ldots,t_m\}$, "se-e">	Each task of $\{s_1, s_2,\ldots, s_n\}$ must be successfully finished in order to finish any of $\{t_1, t_2, \ldots, t_m\}$
Grouping	<$\{t_1,t,\ldots,t_m\}$, "grp">	Tasks $\{t_1, t_2,\ldots, t_m\}$ must be executed indivisibly.
Transition	<s, t, "trans", cond>	Task *t* is activated at the end of task *s* provided that the condition *cond* holds.
Deadline	<s, t, "ddl">	Task *t* may start only if task *s* has not started.
OR-branching	<s, $\{t_1,t,\ldots,t_m\}$, "OR-br", $\{c_1,c_2,\ldots c_m\}$>	One of $\{t_1, t_2, \ldots, t_m\}$ may be executed after the end of *s*, depending on which of the conditions $\{c_1, c_2, \ldots, c_m\}$ holds.

6.3 Dynamic precedence relationships

Precedence relationships are part of the model, hence they must be introduced at model definition time. Some of them, however, may only be determined precisely at enactment time, when the model is instantiated. This is the case of those precedences that must be considered individually for each single element in a set.

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

In order to cope with this limitation, PROMENADE allows the definition of *dynamic* precedences. A dynamic precedence stands for an unknown number of usual precedence relationships; the concrete number will be fixed during enactment, depending on the value of an expression defined in terms of one or more parameters of the tasks involved in the precedence.

Definition. A *dynamic basic precedence* relationship is a tuple:

<type, $\{s_1, \ldots, s_n\}$, $\{t_1, \ldots, t_m\}$, combi, parBinding, gener>

where:

- $\{s_1, \ldots, s_n\}$, $\{t_1, \ldots, t_m\}$, *type*, *combi* and *parBinding* as before
- *gener* is a universal quantification generalizing the application of the precedence.

A similar definition can be done for dynamic derived precedence relationships.

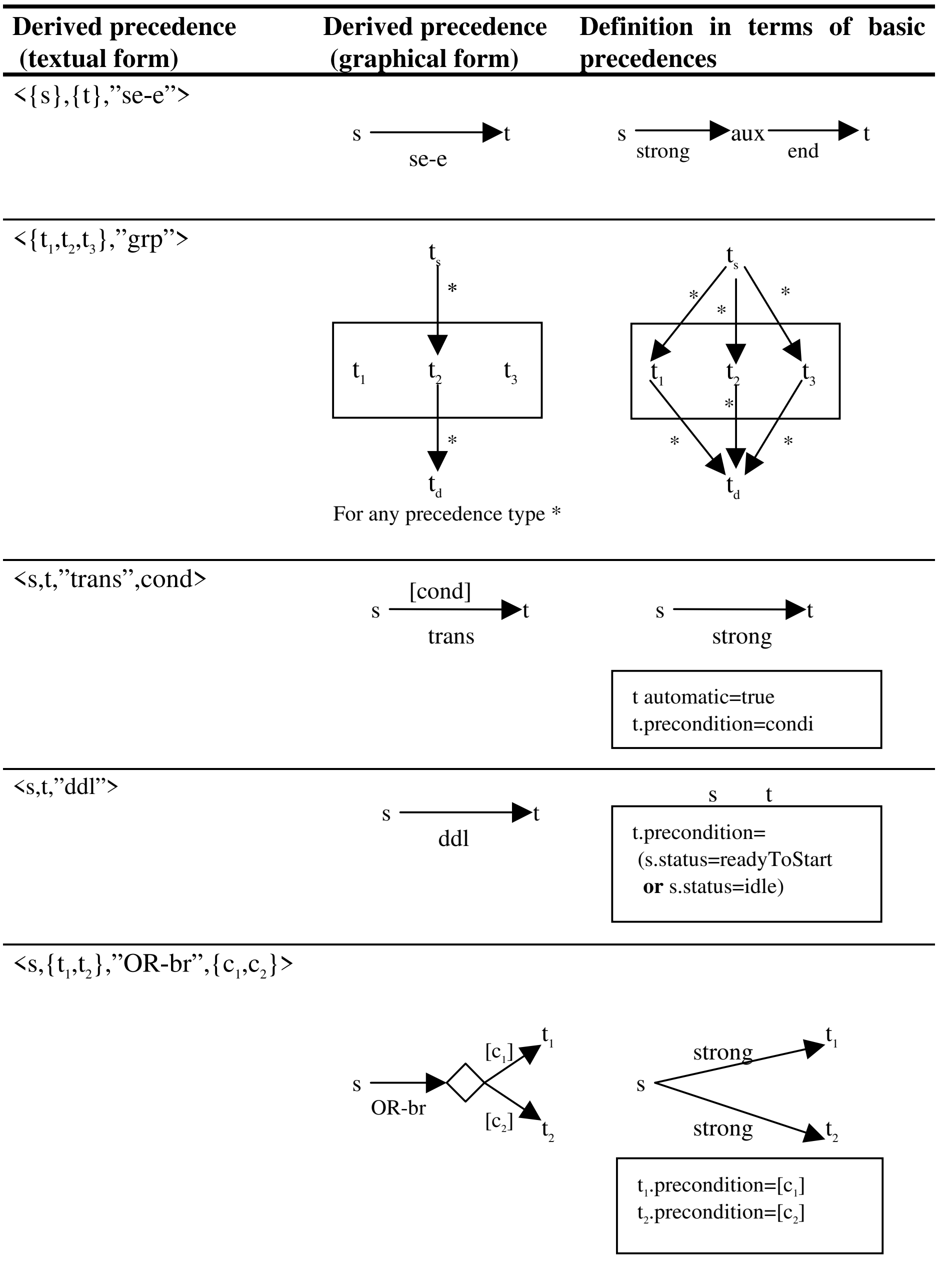

Figure 6: Graphical description of derived precedences in terms of basic ones

Fig. 7 shows the above example using the graphical representation. The *ImplementComponent* parameter *c* is the *component* being implemented, which has an attribute *implDoc* that, in turn, contains the *operations* multivalued attribute. The *TestOperation* parameter *operation* is the operation being tested. The *for all* quantification is the *gener* part of the precedence. Note that this expression complements the parameter binding association.

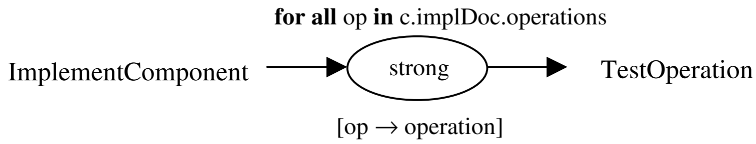

Figure 7. An example of dynamic precedence

Therefore, a dynamic precedence relationship is a pattern for precedence relationships which is defined at modelling time in terms of some document attributes whose value will be known only at enactment time. Dynamic precedence relationships are expanded at enactment time when such values have been calculated.

6.4 Precedence diagrams

Tasks in PROMENADE split into composite tasks and atomic ones. Composite tasks may be further decomposed in simpler tasks (either composite or atomic). The behaviour of a composite task is described in PROMENADE by means of a precedence diagram associated to the composite task. This diagram contains all the precedence relationships that affect the subtasks of the composite task. We will discuss the representation of precedence graphs using UML in section 8.

7 A formalism to define precedences

It has been established in the literature (see, for instance, [JH99], [JB96]) the necessity for a PML to provide means to define new control-flow constructs that fit in the user's modelling requirements. PROMENADE deals with this aspect by supplying a formalism to define precedence relationships. Its main features are the following:

- It is a high-level formalism. This makes unnecesary for the user to struggle with the underground definition of the control flow constructs.
- It allows both the definition of basic precedences in terms of the requirements needed for a task to start (end) and the definition of derived precedences in terms of some basic ones.
- It is based on the OCL language, which provides some degree of standarization.

In the remainder of this section we will outline the most important aspects of this language and we will illustrate them with some examples.

Examples of precedence definition

A precedence states the requirements that must be fulfilled in order to either start or finish the enactment of a task. Therefore, a precedence may be defined in terms of a *start condition* and an *end condition*. Sometimes, precedences are more easily defined in terms of other simpler ones. This is the case of some derived precedences. Therefore, the formalism we are presenting provides both ways to define a precedence. Some examples of precedence definition are shown in the following.

Figure 8 presents the definition of the *strong* precedence. The *is_noted_as* clause provides the elements that take part in the precedence definition. The *with* clause defines those elements using OCL expressions. The *is_defined_as* clause defines the precedence in terms of a start and end conditions (both referred to the target set of tasks: *T*) . The start condition states that all (or some, depending on *combi*) of the source tasks (those belonging to *S*) should have finished successfully in order to start all the tasks in *T.* In this case, no end condition is necessary. Figures 9 and 10 show the definitions of *start* and *feedback* precedences respectively.

```
precedence strong
    is_noted_as  <"strong",S,T,combi,pb>
    with
            S,T:set(Task)
            combi:enum(AND,OR,NULL)
            pb: set(Binding)
    is_defined_as
            start_condition:
                    (combi=#AND implies S->forAll(s|s.status=completeSucc) ) and
                    (combi=#OR implies S->exists(s|s.status=completeSucc) )
            end_condition:
                    NULL
end precedence
```

Figure 8: Definition of the strong precedence

```
precedence start
    is_noted_as  <"start",S,T,combi,pb>
    with
            S,T:set(Task)
            combi:enum(AND,OR,NULL)
            pb: set(Binding)
    is_defined_as
            start_condition:
                    (combi=#AND implies S->forAll(s|s.status=active or
                                                    s.status=completeSucc or
                                                    s.status=completeUnsucc)) and
                    (combi=#OR implies S->exists(s| s.status=active or
                                                    s.status=completeSucc or
                                                    s.status=completeUnsucc))
            end_condition: NULL
end precedence
```

Figure 9: Definition of the start precedence

```
precedence feedback
    is_noted_as  <"fbk",S,T,combi,pb>
    with
            S,T:set(Task)
            combi:enum{AND,OR,NULL}
            pb: set(Binding)
            combi=NULL
            S.size()=1 and T.size()=1
    is_defined_as
            start_condition:
                    (S->forAll(s|s.status=completeUnsucc))
            end_condition:
                    NULL
end precedence
```

Figure 10: Definition of the feedback precedence

Weak precedences may be defined both in terms of start and end conditions (see figure 11) and also as a derived precedence (see figure 12).

```
precedence weak
    is_noted_as  <"weak",S,T,combi,pb>
    with
            S,T:set(Task)
            combi:enum(AND,OR,NULL)
            pb: set(Binding)
    is_defined_as
            start_condition:
                    (combi=#AND implies S->forAll(s|s.status=active or
                                                    s.status=completeSucc or
                                                    s.status=completeUnsucc)) and
                    (combi=#OR implies S->exists(s| s.status=active or
                                                    s.status=completeSucc or
                                                    s.status=completeUnsucc))

            end_condition:

                    (combi=#AND implies S->forAll(s|s.status=completeSucc or
                                                    s.status=completeUnsucc)) and
                     (combi=#OR implies S->exists(s| s.status=completeSucc or
                                                    s.status=completeUnsucc))
end precedence
```

Figure 11: Definition of the weak precedence in terms of start and end conditions

```
precedence weak
    is_noted_as  <"weak",S,T,combi,pb>
    with:
      S,T:set(Task)
     combi:enum{AND,OR,NULL}
     pb: set(Binding)
  is_defined_as
            precedences:
                        <"start",S,T,combi,pb>
                        <"end",S,T,pb>
end precedence
```

Figure 12: Definition of the weak precedence as a derived precedence

Notice in the second definition that we are establishing that a *weak* precedence from the set *S* to the set *T* is equivalent to the yuxtaposition of a *start* precedence and an *end* precedences from *S* to *T*.
In order to define the *successful_end-end* precedence (see figure 13), it becomes necessary to add a new task (*aux*). In this case, *aux* is a very simple task, with no attributes. Sometimes a more complex definition for the newly created tasks may be given. In those situations the *with* part of the precedence definition may get significantly more complicated and, as a result, the whole definition, less comprehensible. For this reason, the addition of new complex tasks should be avoided whenever possible.

```
precedence se-e
    is_noted_as  <"se-e",S,T,combi,pb>
    with:
      S,T:set(Task)
     combi:enum{AND,OR,NULL}
     pb: set(Binding)
  is_defined_as
            precedences:
                        <"strong",S,aux,combi,pb2>
                        <"end",aux,T,pb3>
                        <"end",S,T,combi,pb>

  with:
            pb2,pb3:set(Binding)
            aux:Task
            aux.automatic=true
            aux.params=∅
            pb2=∅
            pb3=∅
end precedence
```

Figure 13: Definition of the successful end-end precedence

Using this notation, it is possible to define in a straightforward way the rest of the derived precedence relationships that we have presented in section 6 (*or-branching*, *deadline* and *transition*).

8 Modelling of Precedence Relationships in UML

We have proposed in section 6 the notion of precedence as a construct which can improve the modelling of complex and detailed processes. Unfortunately, UML does not provide directly this notion. UML transitions used in statecharts and activity diagrams are the most similar concepts offered by UML. In this section we discuss why UML transitions are not precedences and also how precedences can be modelled in UML.

8.1 Precedences are not transitions

UML activity diagrams are a special case of statechart diagrams in which processes are modelled by transitions that link activities (our tasks) for describing the control-flow during process execution. Hence, the state of a process at a given instant is represented by the activities that are being executed at that instant. The state of a process changes by the firing of a transition which enables its target activity. Therefore, activity diagrams focus on the change of the active activity of a process along the time.

This kind of modelling is not suitable when it is necessary to model complex and detailed processes, with a high degree of concurrence between their activities. In this case, more expressive constructs are required. There are at least three reasons for this:

- In the previous section we have defined a set of precedence relationships. Unfortunately, only two of them (*strong prececedences* and *concurrency*) can be modelled by UML transitions and activity diagrams. This is due to the fact that only one semantics can be associated to the issue that a transition is fired and a target state is reached. In the UML case, this semantics corresponds to the start of the activity represented in the target state.
- When different kinds of precedence relationships are used, the modeling with transitions becomes inadequate. On the occurrence of an event, several activities may be affected and may need to change their state (e.g., as a result of an activity starting its execution some other activities may become ready to start or ready to end or completed). As a result, the state of a process at a given time spreads over all the activities that take part in that process. Modelling this with transitions would involve, not only defining different types of transitions, but also changing its firing policy: on the one hand, different events could fire a transition, depending on the type of the transition; on the other hand, more than one transition would need to be fired at a given instant.

- Last, it is also important to notice that activity diagrams have received the criticism that they are not pure object-oriented diagrams, which can impair their use in an object-oriented modelling environment.

Therefore, it seems more suitable to model the process itself by a declarative enumeration of all the precedence relationships between activities. The state of a process will change according to the rules given by this set of precedence relationships.

Once activity diagrams have been discarded, the next suitable choice seems to be the use of dependencies.

According to the UML metamodel, a dependency states that *the implementation or functioning of one or more elements requires the presence of one or more elements* [Ratb]. Therefore a dependence is modelled as a relationship between a (set of) client and a (set of) supplier meaning that the client is dependent on the supplier. It seems quite natural to represent a precedence as an behavioural dependency from a (set of) client activities to a (set of) supplier activities meaning that the the enactment of the client activity depends on that of the supplier activity. As we have shown in section 6 there are many ways in which such a dependency may occur.

This decision is encouraged by the fact that UML defines many stereotypes to be applied to dependencies in order to define different kinds of dependencies that may arise between classifiers. Hence, it is natural to asssociate another stereotype to dependencies to model precedences. Furthermore, some guidelines for modelling in UML encourage the usage of different kinds of dependencies for those relationships that do not fit in the categories of generalizations or associations [RJB99].

8.2 Extensions of the UML metamodel

We can think of two ways of extending the UML metamodel: by making an explicit modification to that metamodel, and by using the UML extension mechanisms.

The first approach has the clear advantage of extending UML in a *clean* way. The required metaelements, with their respective relationships and constraints are defined and incorporated to the metamodel. But it has the drawback of its lack of standarization. On the other hand, the second approach provides a standard extension but it suffers from a lack of expressiveness.

We claim that the modifications that can be made to the UML metamodel via stereotypes, tagged values and constraints are insufficient for many cases (e.g. no new typed-attributes or operations can be incorporated to a stereotyped metaclass) and the resulting metamodel is not completely comprehensible. We strongly believe that UML should provide either more powerful mechanisms to extend its metamodel or define expressive built-in extensions for different domains (e.g., software processes and in general, process technology; real-time systems; etc.).

In the next two subsections we explore both approaches to extend the UML metamodel in order to introduce precedences.

8.3 Modeling Precedences with Explicit UML Metamodel Extensions

The first enlargement done in UML metamodel consists of adding metaclasses for tasks, documents, and the other elements involved in SPM. This has been done in our previous work [FR99b].

Now, we propose to add to the UML metamodel the metaclass *Precedence* and several subclasses which represent the different types of basic and derived precedences introduced in section 6. A sketch of the definition of these additional metaclasses is shown in table 7, following the structure of the UML Semantincs document [Ratb]. The meaning of the attributes come from the definitions given in section 6; some of them are multivaluated, as *source* and *target*. Some constraints have been added (e.g. , a precedence cannot be reflexive). Notice that the predefined associations for dependencies *client* and *supplier* could have been used instead of the new defined associations *source* and *target*. We have preferred to define these new associations in order to make its use more specific: precedences can only be applied to tasks whereas dependencies may be applied to any model element.

Table 7: Metaclasses for precedence relationships

Metaclass	Attributes and associations	Constraints (in OCL)
Precedence	*type: String* *combi: enum{NULL, AND, OR}* *source, target: set(MetaTask)* *parBinding: set(ParameterPrec, ParameterPrec)*	*combi < > NULL implies source.size > 1* *target.forAll(t| not (source->includes(t)))* *parBinding -> forAll(pb| source->includes(pb.param1.task) and target->includes(pb.param2.task)* *client.size = 0 and supplier.size = 0*
BasPrecedence		*type=weak or type=strong or type=start or type=end or type=feedback*
DerPrecedence	*definition: DPDefExpression* *cond: OCLConstraint*	*type< >weak and ... and type< >feedback*
DynPrecedence	*gener: DynPExpression*	*allInstances -> forAll(i| i.oclType < > DynamicPrecedence)*

The relationships between the metaclasses are shown in figure 14. Multiple inheritance has been used to enforce the orthogonality between basic and derived precedences on the one hand, and dynamic and not dynamic precedences on the other.

Several subclasses of *BasDynPrecedence and DerDynPrecedence* have been predefined. The dynamic behaviour of these precedences may be inhibited by using a void vaule for their *gener* attribute. Further derived precedences can be defined as stereotypes of *DerDynPrecedence* or *DerPrecedence*. Each one of them must provide a specific value for the attributes *type* and *definition*.

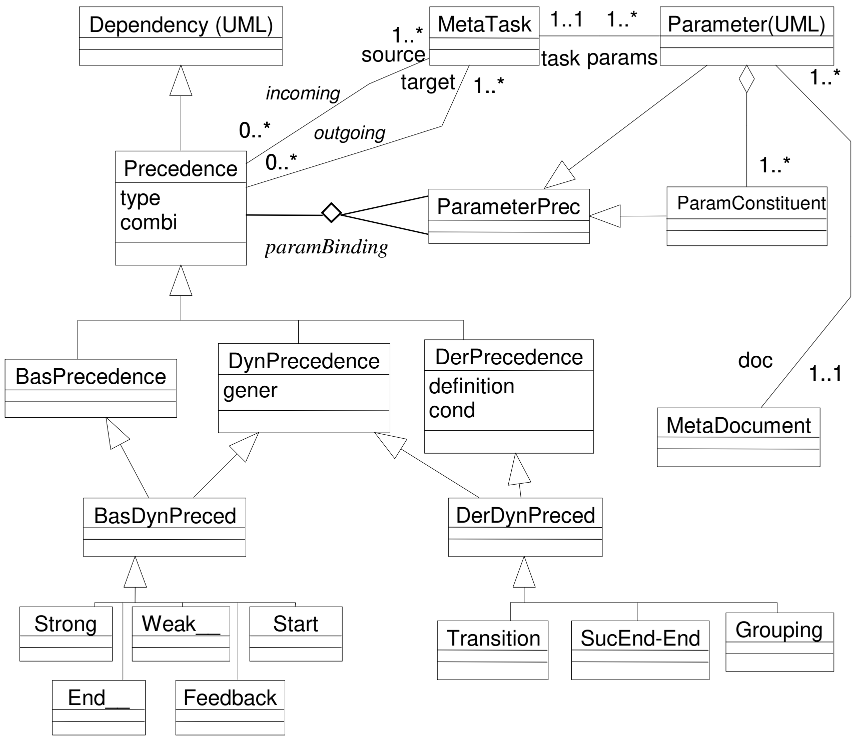

Figure 14: UML metamodel extension with precedence relationships

Parameter binding deserves a careful look. Bindings are usually established between a parameter of a source task and a parameter of a target task, but sometimes, when parameters state for structured documents, it becomes necessary to link a parameter of one task with a part of an aggregate parameter of another task (we will see an example later). For this reason, the parameter binding performed by precedences is modelled as a ternary association between *Precedence* and *ParameterPrec*. *ParameterPrec* is proposed as a superclass for metaclasses *Parameter* (from UML) and *ParamConstituent* (a parameter is an aggregation of its constituents).

Some sort of representation becomes necessary to depict precedences in class diagrams. Fig. 15 shows an example of the chosen representation of precedence relationships. We show explicitly the parameters of tasks. The information of precedence relationships appears not inside the circle, but in a association-class form, except for the combination mode (if any). In the general case, as shown here, parameter binding may include task qualification. The binding between T_2::q_2 and T_4::b implies that one of them is a constituent of the other.

Precedence diagrams

In order to represent the behaviour of a composite task, PROMENADE uses a special kind of class diagrams that contain only tasks (i.e., instances of *MetaTask*) and precedences. We call them *precedence diagrams*.

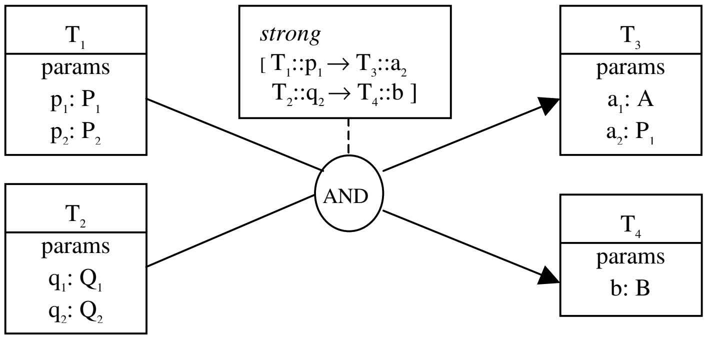

Figure 15: Representation of precedence relationships

Every composite task has an associated precedence diagram, which decomposes the task into its component tasks (which can be composite in turn) and establishes the proactive control-flow. An example of precedence diagram is shown in figure 16. It defines the decomposition of the composite task *BuildComponent*, which is in charge of building and delivering a software component. The composite task appears in the diagram itself, and it is linked with many of its subtasks by means of *start* precedence relationships. Some of the precedences appear just to ensure correct parameter propagation, in the sense that a task cannot start until its input parameters are available.

As it is part of a design process, the precedence diagram encloses some design decision that are in part subjective. In fact, many precedence diagrams may exist for a single composite task, to be selected in the appropriate contexts. In this design, component implementation and test plan generation may take part as soon as specification starts, although they are not allowed to finish until specification ends. On the other hand, component test may not take part until the implementation and the tests are finished, and once the component is successfully tested, it can be stored in the library. Note that unsuccessful testing yields again to the implementation task. The *end* precedence to the main task forces it to finish just once the store task has ended.

As a last remark, the diagram shows the flow of data from one task to another. Implicit parameter binding by name equality are not shown in the diagram. It is worth remarking the association from *BuildComponent* to *SpecifyComponent*: the specification document of the second task is bound to a constituent of the first one, the *spec-doc* attribute of the component. This situation arises in other precedences of the diagram.

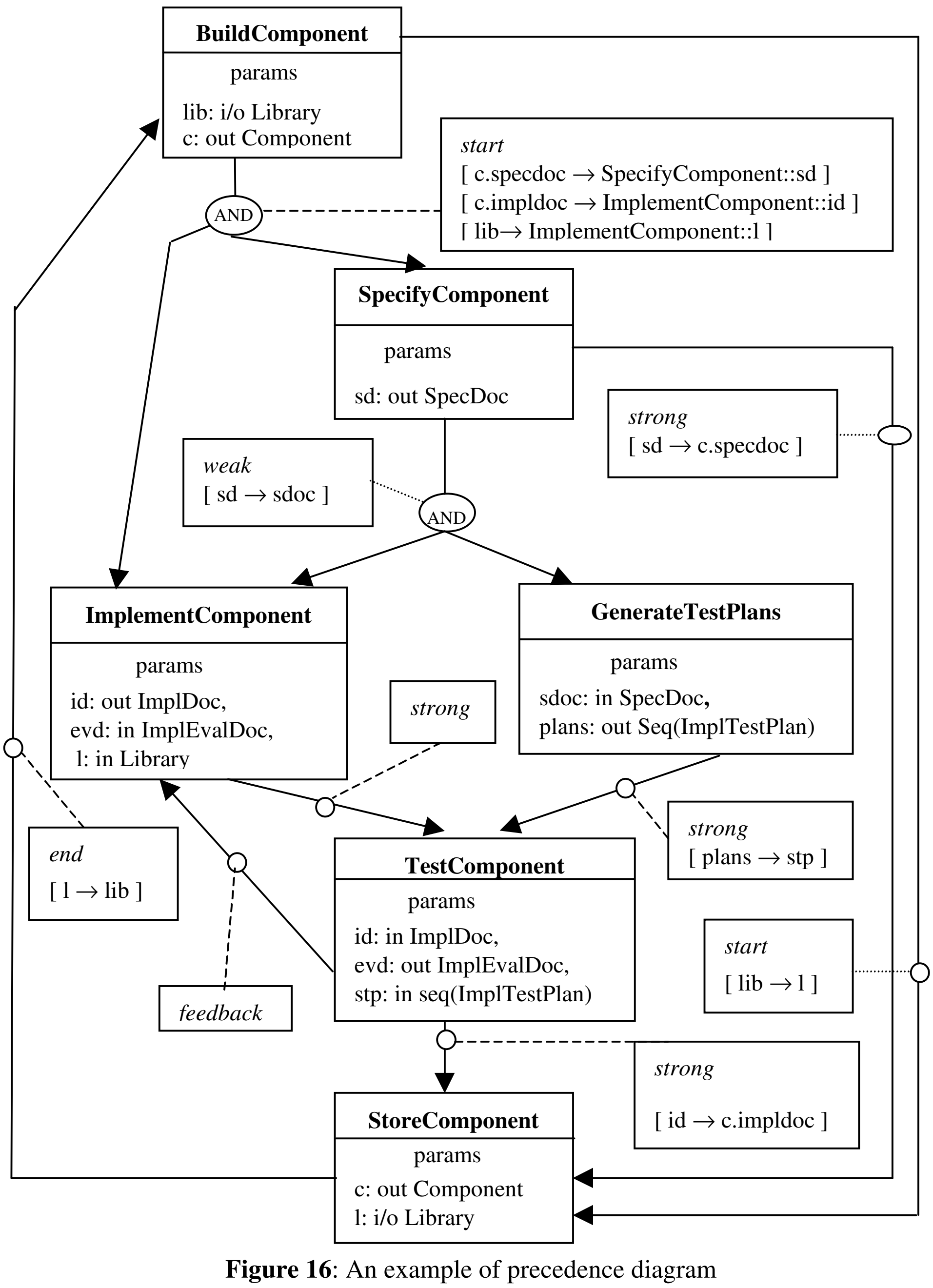

Figure 16: An example of precedence diagram

8.4 Modelling Precedences with the UML Extension Mechanisms

We strongly believe that the extension of the UML metamodel is the most suitable and powerful approach to incorporate precedences in the UML definition. In fact, following the call of the UML revision comitee, we do propose such extension appearing in the forthcoming revisions of the language. However, since future versions of UML might not incorporate such metamodel enlargement, it is important to have the ability to do so in a standard way.

In this section we provide an alternative definition of precedences using UML extension mechanisms. These mechanisms do not have the same expressiveness of a direct and explicit enlargement of the metamodel but they keep standarization.

Precedence relationships are added by applying a new stereotype to the *Dependency* metaclass: *<<Precedence>>*. Some tagged values and constraints have been added to this stereotype, as it is shown in table 8. We take advantage of the fact that stereotypes are generalizable elements in the UML metamodel (hence, tagged values and constraints are inherited) and we define a hierarchy of stereotypes rooted in *<<Precedence>>* which represents all the different kinds of precedences allowed in PROMENADE. All these stereotypes are applied to the metaclass *Dependency*.

Table 8. Definition of a hierarchy of stereotypes to model precedences

Stereotype	Definition	Tagged values
<<Precedence>>	Applied to *Dependency*. It models a precedence of type *type* from a (set of) source task (client) to a (set of) target class (supplier) with some parameter bindings and some combination betwen tasks.	*type*, *combi*, *parBinding*
<<BasPrecedence>>	Subtype of <<Precedence>>, models precedences of types *strong, weak, feedback, start, end.*	
<<DerPrecedence>>	Subtype of <<Precedence>>, models derived precedences.	*definition*, *cond*
<<DynPrecedence>>	Subtype of <<Precedence>>, models dynamic precedences.	*gener*
<<BasDynPrecedence>>	Subtype of <<BasPrecedence>> and <<DynPrecedence. It models precedences which are both basic and dynamic.	
<<DerDynPrecedence>>	Subtype of <<DerPrecedence>> and <<DynPrecedence. It models precedences which are both derived and dynamic.	

The source and target of the precedence will be represented respectively by the *client* and *supplier* association roles of the metaclass *Dependency* in the UML metamodel. This issue changes a little bit the original conception of the roles of *client* and *supplier* in UML dependencies (the *target* of a precedence can be considered more naturally as the *client* of a dependency instead of its *supplier* since its start/end depends on that of the *source*). We have made the presented choice since we feel more important to keep in the precedence representation the notion of *temporal precedence*. Representing the precedence the other way round would have been more consistent with the definition of UML dependencies but would have impaired the intuition of the precedence in the context of process modeling. Anyway, we believe that this is not a crucial point of our approach.

Notice also that multiple clients and suppliers are allowed by the UML metamodel (the multiplicity of both associations is *..*) which is fundamental in order to represent PROMENADE precedences.

Figure 17 shows an example of the representation of a simple precedence using the extensions we have outlined in this section. Notice that, in this case, the UML representation for dependencies has been used.

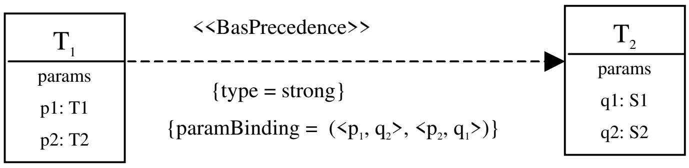

Figure 17. Precedence representation with the UML extension mechanisms

Representing multiple precedences

Although from the UML metamodel it becomes apparent that a dependency may have at the same time multiple clients and multiple suppliers, we have not found any guidelines on how to represent in UML these kinds of dependencies. Therefore, we believe that some form of representation for this situation should be incorporated into the standard UML. This feature becomes essential for PROMENADE since, as we have stated, precedences that may have multiple sources and targets are allowed.

9 Modularity features

We mean by modularity the ability to reuse and combine (compose) both SPM elements (e.g. a task or a document class) and already constructed partial SPMs in order to create a new one. This process of reusing and/or composing elements may involve some updates in order to assemble them smoothly.

We consider three operations that may be applied to SPMs (or its elements) related with modularity: *refinement, renaming* and *composition.*

9.1 Refinement

A SPM (or model element) *m*' is a refinement of another SPM (model element) *m* if *m'* may be obtained from *m* as a composition of correct operations applied to *m*.

We mean by *correct operations applied to m* operations of the class to which *m* belongs such that their precondition holds when they are to be applied.

Refinement allows the derivation of an existing SPM in order to be reused in the construction of a new one (usually within a model composition process).

Correctness of a SPM refinement

A refinement of a correct SPM leads to another correct SPM, since the operations applied on a SPM preserve the validity of its constraints, hence the SPM correctness.

Example: Adding a non-functional specification to a component

This example aims at refining a SPM *m* to construct a software component in such a way that it includes a task to state the non-functional requirements of the component which is being developed. This is a quite common situation which comes up when we need to refine a SPM stored in our SPM library in order to reuse it. In fact, the SPMs stored in the library are usually quite general and need to be refined in order to fulfill some specific requirements.

The steps that are to be carried out in order to do so are the following:

1. Add the task *StateNFBeh* and its subtasks (*importNFProp*, *StateNFBehProp*) to the SPM:

```
m1=m.setTasksCl((((m.getTasksCl())->
      including(stateNFBeh))->
      including(importNFProp))->
      including(stateNFBehProp))
```

2. Add the generalizations: *StateNFBeh→Task, ImportNFProp→Task* and *StateNFBehProp→Task.*

```
m2=m1.setGens((((m1.getGens())->
        including(<Task,stateNFBeh>))->
        including(<Task,ImportNFProp>))->
        including(<Task,StateNFBehProp>))
```

3. Add the aggregations between *StateNFBeh* and its subtasks (*importNFProp, StateNFBehProp*).

```
m3=m2.setAssocs(((m2.getAssocs())->
        including(<StateNFBeh,ImportNFProp>))->
        including(<StateNFBeh,StateNFBehProp>))
```

4. Add a new kind of documents: *NFSpecDoc.*

```
m4=m3.setDocsCl((m3.getDocsCl())->including(NFSpecDoc))
```

5. Add an aggregation between *Component* and *NFSpecDoc* (since the latter is a part of the former).

```
m5=m4.setAssocs((m4.getAssocs())->
```

```
including(<Component,NFSPecDoc>))
```

6. Modify the main task (*BuildComponent*) of the SPM.

```
m6=m5.setMainTaskCl(m5.getMaintaskCl().copysetPrecs(...))
```

m6 is the refined SPM.

9.2 Renaming

A SPM (or model element) *m*' is a renaming of another SPM (model element) *m* if *m'* may be obtained applying to *m* a list of name substitutions.

9.3 Model composition

This operation allows the construction of a new SPM by the composition of some already constructed SPMs. These existing models may be seen as partial models or as particular views of a complete SPM.

This ability is crucial in order to deal with complex models which can be thus built from small pieces. This may happen for models that cover different parts of software development as specification, design or implementation, which may be developed by different teams. Some composition tools will be needed then in order to put together all those partial models and build a model for the entire software development process. Something similar may happen with off-the-shelf models which are to be stuck in some other models that need to incorporate their functionality. Needless to say that these features will enhance model reuse.

The approach PROMENADE has taken in order to deal with model composition consists roughly in building a model *m* from the composition of a set of models $\{m_1,...,m_n\}$ and a set of precedences $\{a_1,...,a_m\}$ in the following way:

- The static part of *m* is the superposition of the generalisation hierarchies of $m_1,...,m_n$, together with the union of their association and aggregation relationships.
- The dynamic part of *M* is built by combining the maintasks of each model $m_1,...,m_n$ with the precedences $\{a_1,...,a_m\}$.

This informal notion is presented in more detail below.

Formalization of the model composition

Definition (Composition of a set of SPMs)
Let $M=\{m_1,...,m_n\}$ be a set of SPMs (instances of the *SPMetamod* metaclass). We define a model composition *m* of $m_1,...,m_n$ in the following way:

m=composition(identifier,M ,P,EXTP,ASS,PARAMS,I) where,

- m is the SPM that results from the composition of $m_1,...,m_n$
- *identifier* is the name of the resulting model.
- $M=\{m_1,...,m_n\}$ is the set of component models.
- P is a set of precedences defined among $m_1.maintask,...,m_n.maintask$
- *EXTP* is a set of extern precedences defined among $m_1.maintask,...,m_n.maintask$ or their subtasks.
- *ASS* is the set of associations stated between the models $m_1,...,m_n$
- *PARAMS* is the set of parameters that define the composition model interface.
- I is the set of additional invariants for the composite SPM.

The resulting composition model m is defined as follows[2]:

- *m.name=identifier*
- $m.submodels = \bigcup_{\forall m_i \in M} m_i$
- $m.supermodels = \varnothing$
- $m.gens = \bigcup_{\forall m_i \in M} m_i.gens$
- $m.assocs = (\bigcup_{\forall m_i \in M} m_i.assocs) \cup ASS$
- $m.docsCl = \bigcup_{\forall m_i \in M} m_i.docsCl$
- $m.rolesCl = \bigcup_{\forall m_i \in M} m_i.rolesCl$
- $m.otherCl = \bigcup_{\forall m_i \in M} m_i.otherCl$
- $m.maintaskCl = t$ with:
    - *t.name=identifier*
    - $t.params = PARAMS$
    - $t.behT.precs = P$
    - $t.behT.taskCl = t$
    - $t.atomic = false$
    - *t.models={m}*
    - $t.supertasksCl = \varnothing$
    - $t.subtasksCl = \bigcup_{\forall m_i \in M} m_i.maintaskCl$

---

[2] This definition is made in terms of the attributes of a SPM. The list of such attributes is given in appendix A.

   - $t.successCondition=subtasks.forAll(t \mid t.status=completeSucc)$

- $m.tasksCl = (\bigcup_{\forall m_i \in M} m_i.tasksCl) \cup m.maintaskCl$
- $m.invs = (\bigcup_{\forall m_i \in M} m_i.invs) \cup I$

That is:

- The generalization hierarchy, the document classes, the role classes, the *other* classes and the imported models of *m* are built by the union of the corresponding elements in each component model.
- The associations defined for *m* are the union of the associations defined for each component model to which some associations *between* component models have been added.
- The main task of *m* is constructed by using as parameters the parameters given to the composition operator, as subtask classes, the main task classes of the component models and as behaviour, the precedence relationships (between the main tasks of the component) stated in the composition operator.
- The task classes of *m* adds to the union of the task classes of each component model the *m* main task .

Following this definition we may construct a new model that has, as structure, the superposition of the structures of its component models and, as behaviour, the behaviours of its component models (these behaviours are encapsulated in their respective main task classes) organized and structured by means of a set of precedence relationships between their corresponding main task classes. Notice that this construction ensures that all PROMENADE SPMs share the reference model structure on which they are constructed. Notice also that, unlike other approaches, the PROMENADE compostion operator does not limit itself to perform a yuxtaposition of the component submodels. It combines those component submodels in the way described by the precedence relationships given to the composition operator.

Correctness of a model composition

The application of the composition operator in order to construct a composite SPM is defined in terms of metamodel operations. Since, by definition, such operations maintain the constraints of the metaclasses, it is clear that, as long as the preconditions of such operations are satisfied, the metamodel constraints will be preserved. Therefore the resulting SPM will be correct. In the following we enumerate which properties must be fulfilled by the participant models in order to ensure that the preconditions of the operations involved in the application of the composition operator *m=composition(identifier,M ,P,EXTP,ASS,PARAMS,I)* are kept.

1. *identifier* (the name of the composite SPM *m*) must be different from any of its submodels.

2. All the elements in *ASS* must link classes belonging to either *s.tasksCl* or *s.docsCl* or *s.rolesCl* or *s.otherCl* for some *s* in *M*.
3. All the occurrences of a task class either in *P or EXTP* must have a sequence of parameters consistent with the parameters of that task class (i.e. equal number and type).
4. There cannot be repeated elements in *PARAMS.*
5. Neither *P*, *EXTP* or the combination of both do contain any deadlock situation or lead to a task starvation.
6. All task classes that take part in *P* are the main task classes for some component model.
7. All task classes that take part in *EXTP* are task classes of some component model (or the main task of the composite model itself).
8. All the invariants in *I* must be consistent to each other and with the invariants of the component models.
9. All task, document, role and other classes that come up in two or more component models with the same name must be identical. That is: for any pair of model elements of the same type *c1, c2* that appear in different component models with the same name: *c1.renaming (∅)=c2*.

10 Conclusions and future work

Some limitations have been identified in current SPML, concerning *expressiveness*, *standarization*, *modularity* and *flexibility* of most of current approaches. We have presented a SPML called PROMENADE which aims at improving the above-mentioned features. In [RF00] a detailed comparison between different approaches concerning these features is presented.

PROMENADE provides a high degree of expressiveness in the description of the control-flow of a SPM by means of several types of *basic precedence relationships* which can be extended (using a high-level notation) to form *derived precedence relationships*. In addition PROMENADE defines the notion of *dynamic precedence relationship* to model precedences which are not known until enactment time.

PROMENADE keeps standarization by using the UML syntax to describe SPM. In this context, it provides an extension of the UML metamodel both to cover the structural parts of a SP and the dynamic parts of it (definition of *Precedence* as a particular UML dependency).

The languages provides flexibility by means of hierarchies of task refinements. A specific refinement may be selected or even created at enactment time.

Finally, PROMENADE defines powerful operators (namely *refinement, renaming* and *composition*) that allow the composition of partial SPMs in order to construct a new one. Some correctness concerns have been taken into account in the definition of these operators.

There are several aspects of the PROMENADE language that are incomplete and on which we are currently working. These are the following:

- The statement of the specific conditions that a SPM must satisfy in order to guarantee its correctness.
- Changes *on-the-fly* should be allowed in PROMENADE in order to provide a higher degree of flexibility.
- The formal semantics of the language are not yet established (this includes the formal semantics of the notation to define precedence relationships). We plan to describe these semantics using a reactive approach (i.e. ECA rules).
- A graphical software tool to define SPMs in PROMENADE is ongoing.

Appendix A: Definition of the metaclasses: SPMetamod, MetaTask and MetaDocument

A.1 SPMetamod

- **Supertype:** *Model* (A class of the UML metamodel)
- **Attributes and associations:**

  - *name: String* (SPM identifier. Inherited from *ModelElement*, defined in UML metamodel).
  - *invs: set(OCLConstraint)* (Collection of the SPM invariants to be kept at enactment time).
  - *gens: set(Generalization)* (Collection of the generalization relationships defined by the SPM. *Generalization* is defined in UML metamodel).
  - *assocs: set(Association)* (Collection of the association relationships defined by the SPM. *Association* is defined in UML metamodel).
  - *docsCl:set(MetaDocument)* (Collection of the document classes that compose the SPM being defined).
  - *tasksCl:set(MetaTask)* (Collection of the task classes that compose the SPM being defined).
  - *rolesCl:set(MetaRole)* (Collection of the role classes that compose the SPM being defined).
  - *otherCl:set(Class)* (Collection of the other classes that compose the SPM being defined).
  - *maintaskCl: MetaTask* (The main task of the SPM).
  - *submodels: set (SPMetamod)* (The collection of submodels that constitute this SPM).
  - *supermodels:set(SPMetamod)* (The collection of the models that use this SPM).
  - *externPrecs: set(Precedence)* (Set of precedences between two task classes that both have different supertask class and one of them is not the supertask class of the other).

A.2 MetaTask

- **Supertype:** *Class* (UML metamodel element)
- **Attributes and associations**

  - *name: String* (Task identifier. Inherited from *ModelElement*, defined in UML metamodel)
  - *features: set(Feature)* (Inherited from Class, in the UML metamodel. *Feature* is defined in the UML metamodel). It includes both attributes and operations. Three operations are predefined for each task class: *init(), start()* and *end().*
  - *params: sequence(Parameter)* (the set containing the task parameters).
  - *tbeh:TaskBeh* (the attribute containing all the precedence relationships concerning its subtasks).
  - *precondition:OCLConstraint* (a predicate that establishes a requirement that should be fulfilled in order to start the execution of this task).
  - *atomic: Boolean* (a boolean to inform whether the task is atomic or not).
  - *automatic: Boolean* (a boolean to inform whether the task is automatic or not).
  - *successCondition:OCLConstraint* (a predicate that establishes the conditions that must be fulfilled in order to finish successfully the execution of this task.
  - *models: set(SPMetamod)* (it refers to the set of models that use this task class).
  - *supertasksCl: set (MetaTask)* (the set of task classes that use this in order to describe their behaviour. The cardinality of this set may be bigger than one only in the case of the maintask of a SPM that is reused several times).
  - *subtasksCl: set(MetaTask)* (a set containing all the task classes that may take part in the description of the behaviour of t).

A.3 MetaDocument

- **Supertype:** *Class* (UML metamodel element)

- **Attributes and associations**

  - name: String (Document identifier. Inherited from *ModelElement*, defined in UML metamodel)
  - *features: set(Feature)* (Inherited from Class, in the UML metamodel. *Feature* is defined in the UML metamodel). It includes both attributes and operations.
  - *models: SPMetamod* (it refers to the model to which this document class belongs).

A.4 TaskBeh

The TaskBeh class represents the behaviour of a task in terms of precedence relationships between its subtasks.

- **Attributes and associations**

  - *taskCl:MetaTask* (The supertask class).
  - *precs: set*(*Precedence*) (Set of precedences between subtasks that describe the behaviour of the supertask).

References

[AC96] Avrilionis, D.; Cunin, P-Y.; Fernström, C. OPSIS: A View-Mechanism for Software Processes whcih Supports their Evolution and Reuse. in Proc. of the 18th. Intl. Conf. on Software ENgineering (ICSE-18). Berlin, Germany. March, 1996.

[ALO96] Alloui I.; Latrous S.; Oquendo, F. A Multi-Agent Approach for Modelling, Enacting and Evolving Distributed Cooperative Software Processes. In C. Montagnero (Ed.) Proc. of the 5th. European Workshop on Software Process Technology (LNCS-1149). Nancy, France. October, 1996.

[AL98] Allweyer, T; Loos, P: Process Orientation in UML through Integration of Event-Driven Process Chains. Proceedings of UML 98' Workshop, Ecole Superioeure des Sciences Appliquées pour l'Ingénieur-Mulhouse Université de Haute-Alsace (1998), 183-193.

[BFG94] Bandinelli, S.; Fuggeta, A.; Ghezzi, C.; Lavazza, L.: SPADE: An Environemnt for Software Process Analysis, Design and Enactment. In Finkelstein, A.; Kramer, J.; Nuseibeh, B. (eds.): Software Process Modelling and Technology. Advanced Software Development Series, Vol. 3. John Wiley & Sons Inc. (1994).

[Chr94] Chroust, G.: Partial Process Models. Software Systems in Engineering, PD-vol. 59 (1994)

[DEA98] Dami, S.; Estublier, J.; Amiour, M.: APEL: a Graphical Yet Executable Formalism for Process Modeling. E. di Nitto, A. Fuggetta (eds.), Kluwer Academic Publishers (1998).

[DKW99] Derniame, J.-C.; Kaba, B.A.; Wastell, D. (eds.): Software Process: Principles, Methodology and Technology. Lecture Notes in Computer Science, Vol. 1500. Springer-Verlag, Berlin Heidelberg New York (1999).

[EHT97] Engels, G.; Heckel, R.; Taentzer, G.; Ehrig, H. A View-Oriented Approach to System Modelling Based on Graph Transformation. In Proc. of the European Software Engineering Conference (ESEC'97). LNCS-1301. Springer-Verlag. 1997.

[FKN94] Finkelstein, A.; Kramer, J.; Nuseibeh, B. (eds.): Software Process Modelling and Technology. Advanced Software Development Series, Vol. 3. John Wiley & Sons Inc., New York Chichester Toronto Brisbane Singapore (1994).

[FR99a] Franch, X.; Ribó, J.M. Some Reflexions in the Modelling of Software Processes. In Proceedings of the International Process Technology Workshop (IPTW-99) (Villard de Lans, France). January 1999.

[FR99b] Franch, X.; Ribó, J.M. Using UML for Modelling the Static Part of a Software Process. In Proceedings of UML '99, Forth Collins CO (USA). Lecture Notes in Computer Science (LNCS), Vol. 1723, pp. 292-307. Springer-Verlag (1999).

[HJK96] Heimann, P.; Joeris, G.; Krapp, C. A.; Westfechtel, B. DYNAMITE: Dynamic Task Nets for Software Process Management. In Proc. of the 18th. Int. Conf. on Software Engineering. Berlin, Germany, 1996 pp. 331-341.

[JB96] Jablonski, S.; Bussler, C.: *Workflow Management. Modeling Concepts, Architecture and Implementation*. ISBN 1-85032-222-8 International Thomson Computer Press (1996).

[JPL98] Jaccheri, M.L.; Picco, G.P.; Lago, P.: Eliciting Software Process Models with the E3 Language. ACM Transactions on Software Engineering and Methodology 7(4) October, 1998.

[JSW99] Jäger, D.; Schleicher, A.; Westfechtel, B.: Object-Oriented Software Process Modeling. Proceedings of the 7th European Software Engineering Conference (ESEC), LNCS 1687 Toulouse (France), September 1999.

[JH99] Joeris, G.; Herzog, O.: Towards a Flexible and High-Level Modeling and Enacting of Processes. Proceedings of the 11th. Conference on Advanced Information System Engineering (CAiSE), LNCS 1626, pp. 88-102, 1999.

[KD96] Kaba, A.B.; Derniame, J.C. Modelling Processes for Change: Basic Mechanisms for Evolving Process Fragments. In Proc. of the European Workshop on Software Process Technology (EWSPT'96), 1996.

[McL98] McLeod, G: Extending UML for Entreprise and Business Process Modeling. Proceedings UML 98' Workshop, Ecole Superioeure des Sciences Appliquées pour l'Ingénieur-Mulhouse Université de Haute-Alsace (1998), 195-204.

[Ost87] Osterweil, L. Software Processes are Software Too. In Procs. of the Intl. Conf. on Software Engineering (ICSE-9), 1987.

[Rata] Rational Software Corporation: UML extension for Objectory Process for Software Engineering. http://www.rational.com/uml.

[Ratb] Rational Software Corporation *et al.*: UML Semantics. http://www.rational.com/uml

[RD98] Reichert M, Dadam P: ADEPT-flex Supporting Dynamic Changes of Workflows Without Losing Control. Journal Of Intelligent Information Systems, 10, 93-129 (1998). Kluwer Academic Publishers.

[RF00] Ribó, J.M.; Franch, X. Searching for Expressiveness, Modularity, Flexibility and Standarisation in Software Process Modeling. Submitted to the Brazilian Symposium on Software Engineering (SBES-00).

[RJB99] Rumbaugh, J.; Jacobson, I.; Booch, G.: The UML User Guide. Addison Wesley (1999).

[SHO95] Sutton S.M.; Heimbigner D.; Osterweil L.J.: APPL/A: A Language for Software Process Programming. ACM Transactions on Software Engineering and Methodology. Vol 4. N. 3, July 1995. 221-286.

[SO97] Sutton, S.M.; Osterweil, L.J.: The Design of a Next-Generation Process Language. Proceedings of ESEC/FSE '97, Lecture Notes in Computer Science, Vol. 1301, M. Jazayeri and H. Schaure (eds.). Springer-Verlag, Berlin Heidelberg New York (1997), 142-158.