Supporting Process Reuse in PROMENADE

Josep M. Ribó[1], Xavier Franch[2]

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

Abstract. Process reuse (the ability to construct new processes by assembling already built ones) and process harvesting (the ability to build generic processes that may be further reused, from existing ones) are two crucial issues in process technology. Both activities involve defining a set of mechanisms, like *abstraction, adaptation, composition, etc.* which are appropriate to achieve their goals. In this report, we define a general framework to process reuse and harvesting that proposes a complete set of mechanisms to deal with both activities. This general framework is particularized to the context of a process modelling language to model software processes, called PROMENADE. A definition of the identified reuse and harvesting mechanisms is proposed in the context of PROMENADE. Finally, two process reuse case studies which composes various reuse mechanisms are presented.

1. Introduction

Modularity and reuse capabilities in the context of process technology refer to the definition of a process model by means of reusing other process models already defined. Process reuse is a crucial feature in process technology since if we are able to rely on already built process models as pieces that will be assembled in the construction of a new one, instead of constructing it from the scratch, this will result in a modular and probably more efficient model construction.

There are several issues that are connected with reuse: the ability to identify and generalize useful parts of already constructed process models in order to be reused later (*harvesting*); the adaptation of these models to be effectively reused in the construction of a new one; the ability to compose adequately different models in the construction of a new one; the projection of those interesting aspects of a process model; the ability to select a built model from a repository to be reused in the construction of a new model; model configuration managements, etc.

These and other issues concerning modularity-reuse capabilities within a PML (both in the fields of workflow management and software process modelling, SPM) have been addressed in the literature [Chr94, Kal96, AC96, Jacc96, Per96, Car97, Kru97, PTV97, JC00, RRN01]. Although some process reuse frameworks have been proposed [JC00, RRN01] and there are various languages that cover this issue [INCO96, Bog95, Car97, Jacc96, AC96, ABC96, EHT97], we are not aware of any PML that offers a wide range of reuse mechanisms along with a standard notation to represent them. This absence has a negative impact in the usability of current PMLs for modelling real cases.

In this respect, the objectives of our approach are the following:

- To provide a powerful and expressive reuse framework focused on the language mechanisms that are necessary in the context of process technology to achieve reuse. This leads to the identification of a set of reuse mechanisms. Due to the objectives of our work, we only focus on modelling issues. Therefore, we do not consider other aspects (like model retrieval and configuration management) that are also relevant to reuse.

- To propose a particular definition of the identified mechanisms in the context of a specific PML. This definition should include the features of *expressiveness* (ability to model most of the situations) and *standardization* (the reuse mechanisms will be mapped into UML) and should support the identification and application of *process patterns* (i.e., common solutions to common problems that may be adapted and reused). The defined reuse operators have been incorporated into the PROMENADE metamodel (which is an extension of the UML metamodel. See [RF00, RF02]).

In this report we restrict to the reuse features of PROMENADE. A description of the language can be found in [RF00, RF01].

Section 2 presents an expressive framework for process reuse, which is centered in the necessary reuse mechanisms. Section 3 presents the definition of *process patterns* in PROMENADE as a way to achieve parameterization and instantiation of SPMs. Section 4 presents SPM morphisms which will be necessary to define the reuse operators of *generalization, specialization, renaming, projection, composition*, which are desctibed in sections from 5 to 9. Section 10 presents an example of process reuse and section 11 contains the conclusions.

2. A framework for software process modularity/reuse

We devote this section to the presentation of a framework for reuse which focuses specifically on the mechanisms that a PML should provide in order to help achieve reuse. This framework also presents the various reuse strategies that make use of these mechanisms. Afterwards, sections 5 to 9 show the specific definition of the reuse mechanisms in PROMENADE.

2.1 A mechanism-centered framework for process reuse

Traditionally, three levels of abstraction have been established in the process modelling literature for PMs (see, for instance, [DKW99]) yielding to three different types of models:

- *Process pattern*: An abstract model that offers a template solution to a common problem. This model may be parameterized and/or contain non-refined elements. It may be customized to meet the requirements of a particular process and/or combined with other process patterns or enactable models. A more detailed description of process patterns along with their modelling in PROMENADE can be found in section 3.
- *Enactable model*: A particular process model ready to be enacted. It may come either from the adaptation (in particular, instantiation) of a process pattern, or from the combination of other process models or have been created ad-hoc. The PML may incorporate delayed binding mechanisms so that it may contain an incomplete process descriptions.
- *Enacting model*: An instantiation of an enactable model. If the enactable model contains some incomplete process description, it will be completed at enactment time.

In the context of process reuse, we keep this notation just changing the term *process pattern* for *template model.* The term *process model* may refer to a model at any of the three levels. We remark that an enactable model may contain some not-implemented tasks to be refined during enactment, using some delayed binding mechanism, as presented below).

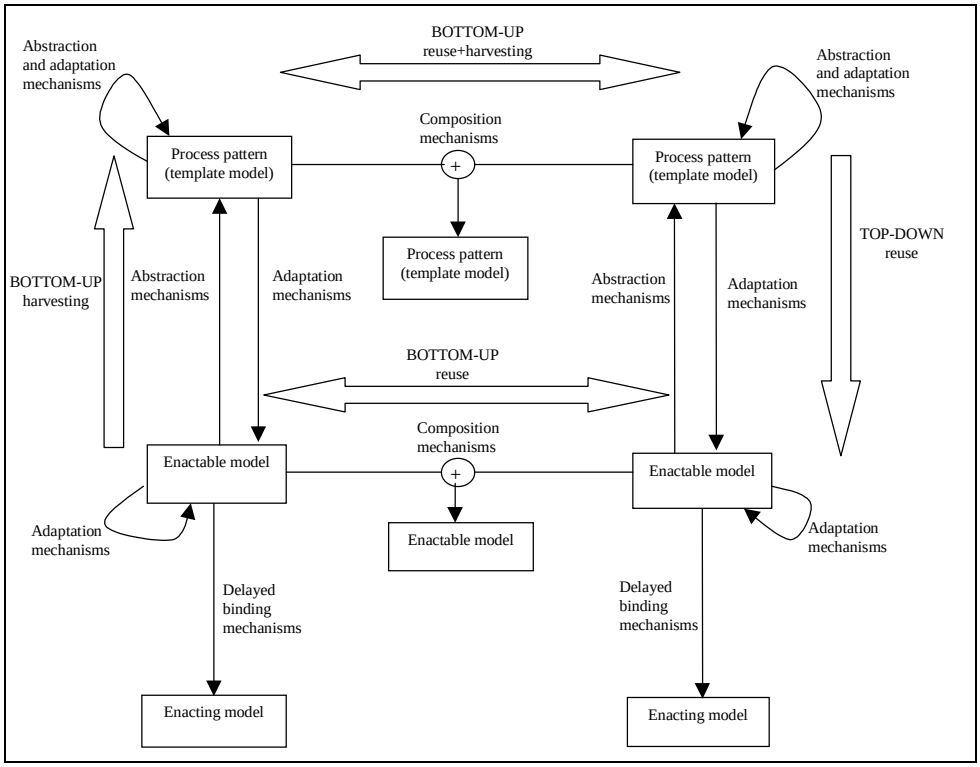


Fig. 1: Modularity/reuse framework

More precisely, the notion of *process reuse* refers to a pair of general and complementary activities which are involved in it:

- *Harvesting*: the process of transforming one or more PMs (enactable models or process patterns) into one process pattern so that it can be reused afterwards.
- *Reuse*: the process of transforming one or more PMs (enactable models or process patterns previously harvested or constructed ad-hoc) into one enactable model.

The activities of *harvesting* and *reuse* may be carried out using the following types of mechanisms:

- *Abstraction mechanisms*. To get rid of some specific or undesired details of a PM. Abstraction mechanisms come up in harvesting activities.
- *Adaptation mechanisms*. To make a PM (usually a process pattern) either more specific or suitable to be reused in a particular process.
- *Composition mechanisms*. To combine a set of PMs in order to create a new one. Composition mechanisms are used both in reuse and harvesting activities.
- *Delayed binding mechanisms*. To delay until enactment time the selection of a specific part of a SPM.

These mechanims may be grouped along two different, usual harvesting and reuse strategies:

- *Bottom-up*. Construction of PMs using either composition or generalization mechanisms. Using a bottom-up strategy we may obtain a more general model from another specific one (useful for harvesting) or a more complex model from its parts (useful for harvesting, if applied to process patterns and for reuse, if applied to enactable models).

- *Top-down*. Construction of PMs using adaptation mechanisms. Using a top-down strategy we may obtain a more specific model from another general one (useful for reuse). We do not use the top-down strategy for harvesting.

In both cases, the resulting PM is in the template or the enactable level. Both strategies may be combined in a reuse activity. In the same way, the two forms of the bottom-up strategy may be combined in a harvesting activity. A whole reuse process combining the activities of reuse and harvesting may itself be modelled as a sequence of reuse mechanisms application, following any valid path in the framework shown in figure 1. Notice that we have defined types of mechanisms to move along all the feasible directions of the diagram.

2.2 Harvesting and reuse mechanisms

In this section we detail and decompose the reuse types of mechanisms presented above.

Abstraction mechanisms. Two abstraction mechanisms are considered, which correspond to the two ways that are normally used in order to abstract details from a complex model: *generalization* (consistent removal of some elements from the SPM) and *parameterization* (encapsulation of the abstracted elements by means of parameters).

Adaptation mechanisms. Adaptation mechanisms involve some sort of model modification. The following modifications can be applied on a PM:

(1) Addition of new elements to the PM. This makes the PM more specific and leads to two different mechanisms: *specialization* (consistent addition of some elements to the SPM) and *instantiation* (tailoring a parameterized PM to a specific context by determining its parameters). Notice that both mechanisms are the opposite to the abstraction ones.

(2) Removal of (unnecessary) elements from the PM. We use the *projection* mechanism for this purpose. This mechanism restricts the model to the selected elements and their context (i.e., it obtains a *view* of it). Notice that a generalization may be considered a removal of elements with abstraction purposes while a projection is a removal of elements with adaptation purposes.

(3) Substitution of elements of the PM. This substitution may be performed at two different levels (lexical and semantic) leading to two mechanisms: *renaming mechanism* (lexical substitution) and *semantic substitution mechanism* (substitution of some model elements for other different ones which model similar concepts).

Composition mechanisms. We distinguish three ways to carry out a composition of a set of PMs:

(1) Shallow composition. This is a grouping of a set of PMs without adding any additional specific semantics (the behaviour of the resulting model is the sum of the behaviours of the components). Furthermore, the namespaces of the component PMs are kept separated in the resulting one (i.e., no merging of namespaces). This composition is performed by the *grouping* mechanism.

(2) Deep composition. This is a composition of a set of PMs by adding some additional behaviour to the resulting model (by defining several precedences between the main tasks of the components). The namespaces of the component PMs are merged into a single one. This is performed by the *combination* mechanism.

(3) Inclusion. In this kind of composition, the functionality of an entire PM is incorporated as a subtask into the behaviour of a composite task (which belongs to another PM), possibly with some additional behaviour. This is performed by the *inclusion* mechanism.

Delayed binding mechanisms. By delayed binding we mean the possibility to enact models that are not completely described (i.e., the implementation of some activities will be decided during enact-

ment. [Per96] uses the terms *primitivation* and *stratification* to refer to this). These models would be completed at enactment time using the *refinement* mechanism (a non refined task is substituted for a specific refinement of that task that has been included in the model at modelling time) and/or the *inclusion* one (i.e., a non-refined task *t* is substituted for an already constructed model which is included into the enacting one in the place of *t*).

2.3 Mechanism definition in PROMENADE

PROMENADE provides a specific definition of all the above-mentioned mechanisms. Specifically, an expressive definition of the parameterization/instantiation mechanisms has been provided in the context of *process patterns* (in fact, process patterns have been defined as parameterized SPMs). The rest of mechanisms have been defined by means of one or more PROMENADE operators. Thus, the term 'operator' here refers to the specific definition of a harvesting/reuse mechanism within the context of PROMENADE. These operators may be applied both to enactable models or to process patterns.

All the PROMENADE behavioural and reuse constructs have been mapped into UML (hence, all the diagrams used to describe the behaviour or reuse of a PROMENADE model are standard UML).

Wherever an extension to the UML metamodel has been necessary, we have used the UML built-in extension mechanisms.

We devote the next few sections to the definition of process patterns and the harvesting/reuse operators in PROMENADE. We rely on the notion of morphism between SPMs in order to provide the operators definition. Intuitively, a morphism between SPMs is a transformation from a SPM into another that keeps (part of) the structure of the SPM.

3. Process patterns in PROMENADE

Parameterization/instantiation reuse mechanisms are defined in PROMENADE by means of process patterns. These process patterns may be combined with other SPM/process patterns to construct a bigger SPM/process pattern. Process patterns are integrated into the PROMENADE metamodel. In the next few sections we detail the main features of process pattern definition in PROMENADE.

3.1 Process pattern definition

A process pattern may be defined as an activity abstraction which provides some solution template to a common problem in a given context (see, for instance, [UML01]). [Amb98] identifies three abstraction levels for process patterns (namely, *task process patterns*, *stage process patterns* and *phase process patterns*).

Usually, process patterns are described in a loose and somewhat ambiguous way. Normally, textual descriptions of process patterns are used, which clearly lack the rigour of model descriptions. Furthermore, these notations are not standard. Other approaches [Sto01, GM01, SW01] define the structural and behavioural aspects of a process pattern by means of UML diagrams. However, they do not integrate process patterns into the UML metamodel.

UML does define the notion of *framework* which is intended to be a pattern [RJB99]. A framework is defined as a stereotyped package. It abstracts a set of possibly parameterized collaborations (called *mechanisms*). A framework, itself, may be parameterized. Although UML frameworks are not restricted to a textual definition, they are still far from the rigorous definition which is convenient for the elements that come up in a modelling language. In particular, no definition of the elements that compose a process pattern is provided, the framework concept is not a first class metaelement defined in the UML metamodel but just a stereotype whose structure is not established

and no constraints about parameterization are given in the metamodel (e.g., which specific classes may instantiate a particular parameter, which particular constraints must be enforced on those instances and so on); the components of a framework (*name, intention, initial context, process, participants*, etc.) are not defined either.

We are not aware of any software PML which applies its modelling notation to the definition of process patterns.

The PROMENADE approach to process patterns can be characterized by the following key features (we have marked with a '*' the features that constitute PROMENADE contributions):

- A rigorous definition of process patterns integrated into the PROMENADE metamodel is given (see figure 2) (*).
  In this respect, notice that a process pattern is defined as a subclass of a SPM. In particular, a process pattern in PROMENADE is defined as a parameterized SPM.
- Expressive and flexible parameterization. (*)
  Some parameters may encapsulate some structural or behavioural aspects of a process pattern. Any descendant of *Task, Document, Tool, Role, Agent, SPM* or *Class* may be a process pattern parameter. Therefore, process patterns allow the definition of generic SPMs which may be reused in different situations by means of a specific instantiation of its parameters (adaptation). PROMENADE allows the definition of specific constraints on process patterns parameters (in OCL) and provides a specific definition of the correctness of a parameter binding. Two metaclasses (*ParameterConstraint* and *ProcessPatternBinding*) have been incorporated to the PROMENADE metamodel in order to deal with both issues.
  An example of such constraints on the parameters may be the following: "the document class that instantiates the parameter *P* of the pattern must have as subdocuments the document classes *A* and *B"*.
- Since a process pattern is, in particular, a SPM (see figure 2), the modularity/reuse operators that are described in the following few sections may be applied to process patterns resulting in a PML endowed with powerful reuse capabilities. (*)
- Specific process patterns features (*name, author, intent, initial context, result context, applicability*, etc.) have been defined for PROMENADE process patterns in its metamodel (this is not the case of other modelling approaches like UML).
- The model for a process pattern is rigorously defined using the features of the PROMENADE PML (precedence relationships, communications and ECA-rules may be used to describe the behaviour of a process pattern.) (*).
  Notice that this more rigorous definition of a pattern has nothing to do with the detail of such definition. In particular, a loosely defined process pattern may be perfectly well described by means of loose precedences between tasks (*weak, start*, etc.). Furthermore, these tasks may be process patterns parameters which need not to be precisely stated.

As it is shown in figure 2, process patterns are defined in the PROMENADE metamodel as a SPM subclass. The metaelements in this figure are presented below.

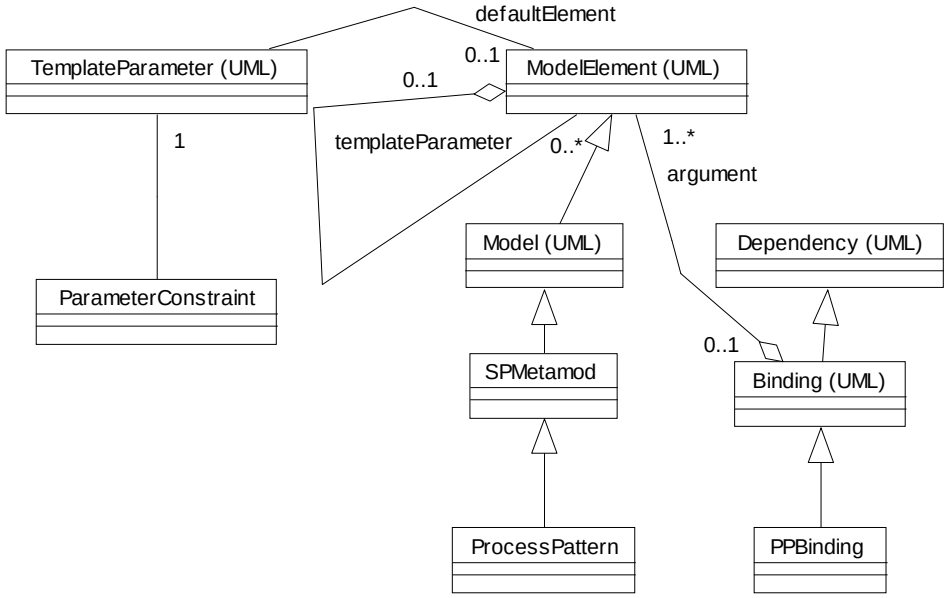


Fig. 2: Metamodel classes for Process pattern definition

3.2 ProcessPattern

Concept

In PROMENADE, a process pattern is defined as a template SPM. That is, a SPM that may have some parameters. Some constraints may be applied to the parameter instantiation in order to enforce some properties which must hold for the parameters. By definition, the instantiation of a process pattern in PROMENADE results in a SPM.

Process patterns in PROMENADE are characterized by means of several features. These features do not differ significatively from those presented in several sources (see [Sto01, GM01, Amb98], for instance). Some important differences arise, however, in the modelling of some of them.

Features

- Name (inherited from ModelElement)
- Author
- Also-known-as
- Keywords
- Intent
- Pattern parameters.
- Initial context.
- Result context
- Applicability (constraints). General constraints and parameter constraints
- Participants. Inherited from *SPMetamod*. The structural aspects of a SPM.
- Process. Inherited from *SPMetamod*. The behavioural aspects of a SPM. It includes both proactive behaviour modelling (precedences) and reactive behavioural modelling (ECA rules). See [RF01]. Notice that PROMENADE does not provide only a textual description of the process pattern behaviour but a rigorous one based on the usual behavioural mechanisms defined by this language.

- Pros & cons
- Example
- Related patterns (reference)

An example of process pattern definition can be found in section 10.

A process pattern is meant to be less specific than a SPM. This is typically achieved by means of parameters that encapsulate either structural or behavioural aspects. In fact, any descendant of *Task, Document, Tool, Role, Agent, SPM* or *Class* may be a process pattern parameter. The power of process patterns is mainly a consequence of this parameterization capability. The features to manage process patterns parameterization comes from the UML metamodel. Since a process pattern is a descendant of the UML metaclass *ModelElement*, it may have *template parameters*, which, in turn, will be instances of *ModelElement*. On the other hand, a *parameterized process pattern* may be instantiated by a specific SPM using a UML *Binding* (i.e. a sequence of model elements each one of which is associated to a process pattern parameter). Hereafter, we will call *actual parameters* the specific classes used to instantiate the template parameters of a process pattern.

However, there exist two issues related to process pattern parameterization which are not modelled in the UML metamodel and which we feel are very important:

- Definition of specific constraints on parameters.
- Definition of correctness of a parameter binding.

Two metaclasses have been added to the UML metamodel to cope with both aspects: *ParameterConstraint* and *ProcessPatternBinding*. This metaclasses are presented in the next few sections.

There are three issues that, along with parameterization, transform PROMENADE process patterns into powerful concepts:

- All the modularity/reuse mechanisms that we have discussed in this chapter are applicable to PROMENADE process patterns. In particular, this means that it is possible:
  - To combine process patterns (with each other or with SPMs) to create bigger process patterns.
  - To create hierarchies of process patterns. In this way, a process pattern could refine a more abstract one.
  - To create a view (a projection) of a process pattern.
  - To incorporate a process pattern into another or into a SPM.

  The definitions given above for these operators apply. However, it is important to keep in mind that any operator applied on a process pattern will lead to another process pattern.
- Since an instantiated PROMENADE process pattern is, by definition, a SPM, it may come up at any place where a SPM is required.
- A PROMENADE process pattern is an abstraction for a SPM. Therefore, we may apply process patterns (in the same way as SPM) to very different granularities and hence, use them to model the three process patterns levels described in [Amb98] (namely, *task process patterns*, *stage process patetrns* and *phase process patterns*).
- Unlike the textual definitions of other approaches, the model for a process pattern is rigorously defined using the features of the PROMENADE PML.

3.3 ParameterConstraint

Concept

A process pattern may define some constraints on the actual parameters that may be used to instantiate a particular template parameter. This is necessary because the process pattern may assume that the actual parameters have some specific features which it needs in order to perform its functionality. A typical example may occur (but it is not restricted) to parameters of the metaclass *MetaTask*. Let us consider the following example: A process pattern *Specification* that has a parameter called

SpecifyDocument, which accounts for a task, may require that the actual task to be bound to that parameter have, in its turn, a task parameter of type *DocumentSpecification.* Obviously, the motivation of this requirement is that the process pattern *Specification* uses that task parameter. This example is extended in section 10.

Each template parameter may be associated to (at most) one instance of the metaclass *ParameterConstraint*, which defines a list of constraints on that parameter.

Features

- tempParam: TemplateParameter
- constraints: set(Predicate)

3.4 Process Pattern Binding (PPBinding)

Concept

Binding is a UML metaclass that establishes a relationship between a template model element (supplier) and another model element (client) that instantiates it. This binding is established by means of a sequence of actual parameters that are bound to the formal parameters of the template class.

PPBinding is a subclass of *Binding* incorporated to the PROMENADE metamodel that regulates process pattern instantiation. In particular,

- The client of a *PPBinding* must be of type *SPM.*
- The supplier of a *PPBinding* must be of type *ProcessPattern*.
- The sequence of arguments of a *PPBinding* instance (i.e. the actual parameters that instantiate the process pattern to which the binding is associated) must:
  (1) be (one-to-one) of the same metatypes as the template parameters of the supplier process pattern.
  (2) keep the constraints stated in the parameter constraints associated to each template parameter of the supplier process pattern.
  (3) keep the general constraints stated by the supplier process pattern.

3.5 Graphical representation

Figure 3 shows how process patterns may be represented in PROMENADE. This figure refers to the example shown in section 10. Notice that the process pattern contain some parameters and that we may attach to its representation constraints concerning those parameters by means of a note. For instance, constraint (5) states that the task class that will instantiate *FSpecifyComp* must have a parameter of type *FSpecDoc* (because this parameter is used within the definition of *SpecifyCompWithNFPattern.*

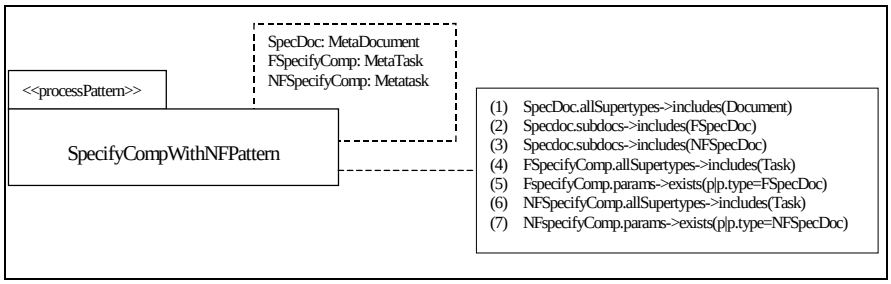


Fig. 3: Graphical representation of a Process pattern

4. SPMs morphisms

Operators are applied to one or more SPMs in order to obtain a resulting new one. They are defined in PROMENADE by means of morphisms between SPMs. Intuitively, a SPM morphism is a transformation established between two SPMs. A SPM morphism, in its turn, is stated in terms of a set of morphisms, each one of them defined from a cathegory of elements that compose the origin SPM (namely, *tasks, documents, roles,* etc.) to the corresponding cathegory in the destination SPM.

The image of a morphism may *forget* some elements of the origin SPM. In the same way, the destination SPM may possess some elements which do not come from the transformation stated by the morphism. If a bijective morphism can be established between the origin and the destination SPMs, we say that both models are isomorphms.

A morphism definition is made by means of OCL constraints. These constraints show how the various components of a specific SPMs on which the operator is applied are transformed into components of the resulting SPM by the application of the operator.

4.1 Formal definition

Let m_1 and m_2 be two SPMs. We say that $f:m_1 \rightarrow m_2$ is a SPM morphism (or a morphism between SPMs) if f is a collection of mappings:

$f=(f_{name}, f_{task}, f_{doc}, f_{role}, f_{class}, f_{comm}, f_{gen}, f_{assoc}, f_{dep}, f_{maintask})$

which are defined in the following way:

- $f_{name} : m_1.name \rightarrow m_2.name$
- $f_{task}: m_1.tasksCl \rightarrow m_2.tasksCl \cup \{\perp\}$
  - For all t in $m_1.tasksCl$, there exists a single t' in $m_2.tasksCl \cup \{\perp\}$ such that $f_{task}(t)=t'$
  - f_{task} is a task morphism.
- $f_{doc}: m_1.docsCl \rightarrow m_2.docsCl \cup \{\perp\}$
  - For all d in $m1.docsCl$, there exists a single d' in $m_2.docsCl \cup \{\perp\}$ such that $f_{doc}(d)=d'$
  - f_{doc} is a document morphism.
- $f_{role}: m_1.rolesCl \rightarrow m_2.rolesCl \cup \{\perp\}$
  - For all r in $m_1.rolesCl$, there exists a single r' in $m_2.rolesCl \cup \{\perp\}$ such that $f_{role}(r)=r'$
  - f_{role} is a role morphism.
- $f_{class}: m_1.otherCl \rightarrow m_2.otherCl \cup \{\perp\}$
  - For all c in $m_1.otherCl$, there exists a single c' in $m_2.otherCl \cup \{\perp\}$ such that $f_{class}(c)=c'$
  - f_{class} is a class morphism.
- $f_{comm}: m1.commsCl \rightarrow m2.commsCl \cup \{\perp\}$
  - For all c in $m_1.commsCl$, there exists a single c' in $m_2.commsCl \cup \{\perp\}$ such that $f_{comm}(c)=c'$
  - f_{comm} is a communication morphism.
- $f_{gen}: m_1.gens \rightarrow m_2.gens \cup \{\perp\}$
  - For all g in $m_1.gens$, there exists a single g' in $m_2.gens \cup \{\perp\}$ such that $f_{gen}(g)=g'$
  - f_{gen} is a generalization morphism.
- $f_{assoc}: m_1.assoc \rightarrow m_2.assoc \cup \{\perp\}$
  - For all a in $m_1.assocs$, there exists a single a' in $m_2.assocs \cup \{\perp\}$ such that $f_{assoc}(a)=a'$

- f_{assoc} is an association morphism.

- f_{dep}:$m_1.deps \rightarrow m_2.deps \cup \{\perp\}$
  - For all d in $m_1.deps$, there exists a single d' in $m_2.deps \cup \{\perp\}$ such that $f_{dep}(d)=d'$
  - f_{dep} is a dependency morphism.
- $f_{maintask}$:$m_1.maintask \rightarrow m_2.maintask$

Remarks:

- $\perp$ is the *undefined element*. It is used in order to define SPM morphisms that forget elements from the origin SPM.
- f_{doc}, f_{role}, f_{class}, f_{comm}, f_{gen}, f_{assoc} and f_{dep} are defined similarly to f.
- In the operator definition that will take place in the following sections, we will use the following notation:

  $F_{task}(m.tasks) \equiv Im(f_{task}) \equiv$
  ```
  m.tasks->iterate(t:MetaTask, res: set (MetaTask)
                                  | res->including(ftask(t))   )
  ```

  That is, $F_{task}(m.tasks)$ and $Im(f_{task})$ refer to the set of task classes in which $m.tasks$ are transformed by the morphism f_{task}.

  F_{doc}, F_{role}, F_{class}, F_{comm}, F_{gen}, F_{assoc} and F_{dep} are defined in an analogous way.

4.2 Isomorphism between SPMs

Let m_1 and m_2 be two SPMs. m_1 and m_2 are isomorphs if there exists a SPM morphism f such that:

1. $Im(f_{task})=m_2.tasksCl$
2. for all t,s in $m_1.tasksCl$: if $f_{task}(s)=f_{task}(t)$, then $s=t$
3. $m_1.tasksCl$ and $m_2.tasksCl$ are isomorphs
4. Conditions analogous to 1, 2 and 3 should hold for the morphisms f_{doc}, f_{role}, f_{class}, f_{comm}, f_{gen}, f_{assoc} and $f_{dep.}$
5. $f_{maintask}(m_1.maintaskCl)=f_{task}(m_1.maintaskCl)$

The isomorphisms between two sets of task classes, document classes, role classes, other classes, communications, generalizations, associations and dependencies are defined analogously.

It is clear from the definition that two SPMs which are isomorphs must have identical structure. They may differ only in the identifiers chosen to name their respective components. Therefore, an isomorphism between two SPMs may be given by a list of lexical substitutions such that, when they are applied to the origin SPM yield the image one.

$f:m_1 \rightarrow m_2$

$f=\{(id_1/id_2),\ (id_3/id_4), ..., (id_n/id_{n+1})\}$

which is read: m_2 is obtained by means of a parallel substitution of the identifiers of m_1: id_1, id_3, ..., id_n for id_2, id_4, ..., id_{n+1}, respectively.

In general, the opposite is not true. That is, a list of lexical substitutions such that, when they are applied to the origin SPM yield the image one, does not always constitute an isomorphism between two SPMs. In particular, it could happen that the SPM, m_2, that would result from the application of this substitution list to the origin one would not respect the unicity of names property (see section 6), which is required for a SPM. Hence, m_2 would not be a well-formed SPM.

The application of a list of lexical substitutions on a SPM m_1 will generate an isomorph one only in the case in which that application keeps the unicity of names property. We will go back to this idea later in the definition of the renaming operator (see section 6).

Two isomorphm SPMs are also called *equivalent SPMs.*

4.3 The identity SPMs morphism

There exists one morphism between SPMs that is very useful for the definition of the PROMENADE reuse operators: *the identity morphism.*

The identity morphism (*id*) transforms each one of the elements of the origin SPM into an identical one in the image SPM. This morphism may be defined in the following way:

$id=(id_{name}, id_{task}, id_{doc}, id_{role}, id_{class}, id_{comm}, id_{gen}, id_{assoc}, id_{dep}, id_{maintask})$

such that:

- $id_{name}\ (m_1.name)=m_1.name$
- $id_{task}: m_1.tasksCl \rightarrow m_2.tasksCl$ is the identity task morphism (For all t in $m_1.tasksCl$, $id_{task}\ (t)=t'$, where t' would be identical to t and would belong to m_2).
- The remainder of the morphisms are defined in an analogous way.

Two remarks are worth mentioning:

1. The images of the component morphisms (e.g., id_{task}) do not include $\perp$ (i.e. no element will be forgotten by the identity morphism).
2. An identity morphism may not be an isomorphism. Given an identity morphism, $id:A \rightarrow B$. If $Im(A)=B$, then *id* is an isomorphism. Furthermore, A and B will be identical. However, if $Im(A) \subset B$ and $Im(A) \neq B$ (i.e., *B* contains other elements apart from *Im(A)*), *id* will not be an isomorphism.

5. Generalization/specialization operators

A model is a generalization of another if it describes a larger number of cases [JC00]. In the opposite direction, we say that a model is a specialization of another. [JC00] confronts several approaches to generalization in the literature. We have adopted the approach by which a specialized SPM narrows (i.e. say more things) about the structure and/or the behaviour of a more general model. This is the most extended notion of model generalization.

The generalization/specialization mechanisms operators may serve to various purposes within the modularity/reuse framework:

- Generalization of useful SPMs in order to transform them into model templates (*process patterns*). The objective is to abstract and parameterize particular details of a well-known and used SPM in order to get a process pattern that will be reused in the future [DC99].
- Adaptation of model templates in order to be reused into a specific context. This is the opposite task to the previous one. It may involve both model specialization and parameter instantiation.
- Adaptation of SPMs previous to the application of some modularity/reuse operators (i.e. combination or projection operator).

The *SPMetamod* metaclass has been defined in the PROMENADE metamodel as a *GeneralizableElement* (see [RF00]). Therefore, it is possible to generalize/specialize SPMs. The semantics of package generalization in UML is the following:

the public and protected elements owned or imported by a package are also available to its children and can be used in the same way as any element owned or imported by the children itself [UML01].

Furthermore, packages may overload elements defined in their ancestors and define new ones.

Notice that UML takes an object-oriented approach to generalization (based on inheritance). Therefore, the PROMENADE approach is consistent with it.

PROMENADE defines several operators in order to generalize/specialize a SPM. They are enumerated in the following sections.

5.1 Operators to generalize a SPM

There are various ways in which a model can be generalized in a consistent way with the PROMENADE approach:

- Remove some classes from the model. This may include tasks, documents, roles, tools... The structure and/or behaviour of a SPM may be made more general by removing specific document or task classes. The operators that may be used for this purpose are *removeClass, removeDocumentCl, removeTaskCl, removeRole, removeTool.*
- Remove some class associations from the model. This may include both associations and aggregations (which are, in fact, a specific type of associations). This kind of removal makes the structure of a SPM more general. The operator used for this purpose is *removeAssoc.*
- Remove dependencies from the model. Again, the structure of a SPM may be made more general by removing some of its dependencies. We use the *removeDep* operator for this purpose.
- Remove some task parameters from some model task. Another way to generalize a SPM consists in keeping a specific task class but removing some of its parameters. In this way, the behaviour of that task class would be made more general. The *removeParam* operator may be used for this purpose.

Figure 4 shows the extension of the UML metamodel defined by PROMENADE to deal with the generalization operator.

5.2 Operators to specialize a SPM

The following operators (which act in the opposite way as the ones presented in the previous section) will be useful for making a SPM more specific. This are the following:

- Add some classes to the model (the operators *addClass, addTaskCl, addDocumentCl, addRoleCl*).
- Add some class associations and/or aggergations to the model (the operator *addAssoc*).
- Add some dependencies to the model (the operator *addDep*).
- Add some task parameters to some model task classes (the operator *addParam*).

In the following section we will define the behaviour of one generalization operator: *taskRemoval.*

5.3 Generalization operators: Task removal.

The objective of this operator is to obtain a new consistent SPM by removing a set of task classes from an existing one.

Formal definition

- *removeTasksCl(m,st)*

This operator removes from the SPM *m* the set of task classes *st,* and the transitive closure of their subtask classes and subclasses.

It is defined as follows:

```
removeTasksCl(m: SPMetamod, st: set(MetaTask)): SPMetamod

removeTasksCl(m,{})=m
removeTasksCl(m,st->prepend(t))=
     removeTasksCl(removeOneTaskCl(m,t),
                  [st->union(t.subtasksCl)]->union(t.allSubclasses))
```

- *removeOneTaskCl(m,t)*

It removes every occurrence of the task class *t* from the SPM *m.* This includes the removal of those precedences, associations, dependencies and communications which involve *t.*

We define the *removeOneTaskCl*(*m*,*t*) operator as a SPM morphism in such a way that the image of the SPM *m* with respect to this morphism will be a new SPM from which any reference to *t* has been removed. In order to reach this purpose, *removeOneTaskCl* induces a transformation for each one of m's components according to the following definition:

```
removeOneTaskCl (m1: SPMetamod, t: MetaTask): SPMetamod

result.tasksCl=m1.TasksCl->iterate(s:MetaTask,
                                   res: set(MetaTask)
                             |res->including (transfoTask(s,t) )
                             )

result.maintaskCl=transfoTask(m1.maintaskCl, t)

result.rolesCl=m1.rolesCl->iterate(r:MetaRole,
                                   res: set(MetaRole)
                             |res->including (transfoRole(r,t) )
                             )

result.docsCl=m1.docsCl->iterate(d:MetaDocument,
                                   res: set(MetaDocument)
                             |res->including (transfoDoc(d,t) )
                             )

result.otherCl=m1.otherCl->iterate(c:Class,
                                   res: set(Class)
                             |res->including (transfoClass(c,t) )
                             )

result.assocs=m1.assocs->iterate(a:Association,
                                 res: set(Association)
                             |res->including(transfoAssoc(a,t))
                           )

result.gens=m1.gens->iterate(g:Generalization,
                             res: set(Generalization)
                           |res->including(transfoGen(g,t))
                           )

result.deps=m1.deps->iterate(d:Dependency,
                             res: set(Dependency)
                         |res->including(transfoDep(d,t))
                           )

result.commsCl=m1.commsCl->iterate(c:Communication,
                                   res: set(Communication)
                               |res->including(transfoComm(c,t))
                           )
result.name=m1.name
```

The transformation of a SPM relies on its elements' transformations (e.g., the task classes of the transformed SPM is the union of the transformations of each original task class). In the following sections we define how the transformation of each kind of element is obtained.

- *transfoTask(s,t)*

transfoTask(s,t) is the task class into which a task class *s* which belongs to m_1 is transformed by the application of the *removeOneTaskCl (*m_1, *t)* operator. This task class is obtained by removing from the task class *s* any reference to *t*.

```
transfoTask (s: MetaTask, t: MetaTask): MetaTask

if (s=t) then NULL
else

result.subtasks=s.subtasks->iterate(u:MetaTask,
                                    res: set(MetaTask)
                                |res->including(transfoTask(u,t))
                          )

result.supertask= transfoTask(s.supertask)

result.precs=s.precs->iterate(u:MetaTask,
                              res: set(Precedence)
                         |res->including(transfoPrec(p,t))
                          )

result.in-comms=s.in-comms->iterate(c:Communication,
                                    res: set(Communication)
                               |res->including(transfoComm(c,t))
                          )

result.out-comms=s.out-comms->iterate(c:Communication,
                                      res: set(Communication)
                                 |res->including(transfoComm(c,t))
                          )

result.participants=s.participants->iterate(r:MetaRole,
                                            res: set(MetaRole)
                                    |res->including(transfoRole(r,t))
                          )

result.responsible=transfoRole(s.responsible)

result.name=s.name

result.features=s.features->iterate(f:Feature,
                                    res: set(Feature)
                                    |res->including(transfoFeature(f,t))
                          )

result.params=s.params->iterate(p:Parameter,
                                res: set(Parameter)
                           |res->including(transfoParam(p,t))
                          )

result.tools=s.tools->iterate(to:MetaTool,
                              res: set(MetaTool)
                        |res->including(transfoTool(to,t))
                          )
```

- *transfoRole(r,t)*

transfoRole(r,t) is the role class into which *r* (that is, a role class which belongs to m_1) is transformed by the application of the re*moveOneTaskCl (*m_1, *t)* operator.

This role class is obtained by removing from the role class *r* any reference to *t* (specificaly, removing *t* as task for which the image of *r* may be responsible or in which it may participate).

```
transfoRole (r: MetaRole, t: MetaTask): MetaRole

result.tasksClR=r.tasksClR->iterate(s:MetaTask,
                                    res: set(MetaTask)
                          |res->including(transfoTask(s,t))
                         )

result.tasksClP=r.tasksClP->iterate(s:MetaTask,
                                    res: set(MetaTask)
                          |res->including(transfoTask(s,t))
                         )

result.docsClR=r.docClR->iterate(d:MetaDocument,
                                 res: set(MetaDocument)
                          |res->including(transfoDoc(d,t))
                         )
 result.name= r.name
```

- *transfoDoc(d,t)*

transfoDoc(d,t) is the document class into which d is transformed by the application of the *removeOneTaskCl (m_1, t)* operator.

This document class is obtained by removing from the document class *d* any reference to *t*.

```
transfoDoc (d: MetaDocument, t: MetaTask): MetaDocument

result.subdocsCl=d.subdocsCl->iterate(d2:MetaDocument,
                                      res: set(MetaDocument)
                                |res->including(transfoDoc(d2,t))
                         )

result.superdocCl=transfoDoc(d2.superdocCl,t)

result.responsibleCl=r.tasksClR->iterate(s:MetaTask,
                                    res: set(MetaTask)
                          |res->including(transfoTask(s,t))
                         )
result.name= d.name
```

- *transfoClass (c,t)*

transfoClass(c,t) is the class into which *c* (a class from m_1, usually applied to classes that does not belong to the cathegories of *document*, *task*, *communication* or *role*) is transformed by the application of the *removeOneTaskCl (m_1, t)* operator.

This class is obtained by removing from the class *c* any reference to *t*.

```
transfoClass (c: Class, t: MetaTask): Class

if(c=t) then result=NULL
else if (c.oclType=MetaTask) result=transfoTask(c,t)
else if (c.oclType=MetaDocument) result=transfoDoc(c,t)
else if (c.oclType=MetaRole) result=transfoRole(c,t)
else if (c.oclType=MetaTool) result=transfoTool(c,t)
else if (c.oclType=MetaAgent) result=transfoAgent(c,t)
else result= result.name=c.name
                  result.features->iterate(f:Feature, res: set(Feature)
                                    |res->including (transfoFeature(f))
                                        )
```

- *transfoAssoc(a,t)*

transfoAssoc(a,t) is the association class into which *a* is transformed by the application of the *removeOneTaskCl (m1, t)* operator.

This association class is obtained by removing from the association class *a* any reference to *t* and by removing *a* itself, if it links *t* with some other class.

```
transfoAssoc (a: Association, t: MetaTask): Association

result=if (a.connection->exists(ae|ae.type=t)) then result=NULL
       else
           a.connection->iterate(ae:AssociationEnd
                                 res: set(AssociationEnd)
                                      |res->including(transfoAssocEnd(ae,t) )
                                )
```

- *transfoDep(d,t)*

transfoDep(d,t) is the dependency class into which the dependency *d* from the SPM m_1 is transformed by the application of the *removeOneTaskCl* (m_1, *t)* operator. This dependency class is obtained by removing from *d* any reference to *t* and by removing *d* if it links *t* with some other class.

```
transfoDep (d: Dependency, t: MetaTask): Dependency

result=if (d.client->includes(t) or d.supplier->includes(t)) then result=NULL
      else result.supplier=iterate->(c:Class, res: set(Class)
                                |res->including(transfoClass(d.supplier,t))
                                     )
           result.client= iterate->(c:Class, res: set(Class)
                                    |res->including(transfoClass(c,t))
                                        )
```

- *transfoPrec(p,t)*

transfoPrec(p,t) is the precedence class into which a precedence *p* that belongs to the SPM m_1 is transformed by the application of the *removeOneTaskCl (m1, t)* operator. This precedence class is obtained by removing from *d* any reference to *t* and by removing *p* itself if it links *t* with some other class.

```
transfoPrec (p: Precedence, t: MetaTask): Precedence

result=if (p.client->includes(t) or p.supplier->includes(t) or p.taskCl=t)
                           then result=NULL
                    else
                           result.taskCl=transfoTask(p.taskCl)
                           result.client=p.client->iterate(c:MetaTask,
                                                    res: set(MetaTask)
                                        |res->including(transfoTask(c,t))
                                     )
                           result.supplier=p.supplier->iterate(c:MetaTask,
                                                    res: set(MetaTask)
                                        |res->including(transfoTask(c,t))
                                     )
                           result.parbind=p.parbind->iterate(pb:ParBinding,
                                                    res: set(ParBinding)
                                       |res->including(transfoParBind(pb,t))
                                     )
```

- *transfoComm(c,t)*

transfoComm(c,t) is the communication class into which *c* is transformed by the application of the *removeOneTaskCl* (m_1 , *t)* operator. This communication class is obtained by removing from *c* any reference to *t* and by removing *c* itself if it links *t* with some other class.

```
transfoComm (c: Communication, t: MetaTask): Communication
```

Post:

```
   result=if (c.receivers->includes(t) or c.senders->includes(t))
          then result=NULL
          else
```

```
result.receivers=c.receivers->iterate(ce:CommunicationEnd
                                      res: set(CommunicationEnd)
                                   |res->including(transfoClass(ce))
                              )
result.senders=c.senders->iterate(ce:CommunicationEnd
                                     res: set(CommunicationEnd)
                                  |res->including(transfoClass(ce))
                              )
```

5.4 Graphical representation of generalization/specialization operators

As we have said, a PROMENADE SPM has been defined as a UML package, hence, according to the UML metamodel, it is a generalizable element. As a consequence, we may use the standard UML notation to represent a generalization/specialization relationship between two SPMs.

In the PROMENADE metamodel, a special kind of the UML *Generalization* has been defined (namely, *SPMGenRel;* see figure 4) in order to depict the relationship established between two SPMs, *A* and *B*, such that *B* is obtained from the application of a generalization/specialization relationship on *A*. Figure 5 shows an example of the graphical representation. In this case, *A* has been obtained from the application of a generalization operator (*taskRemoval*) to *B*.

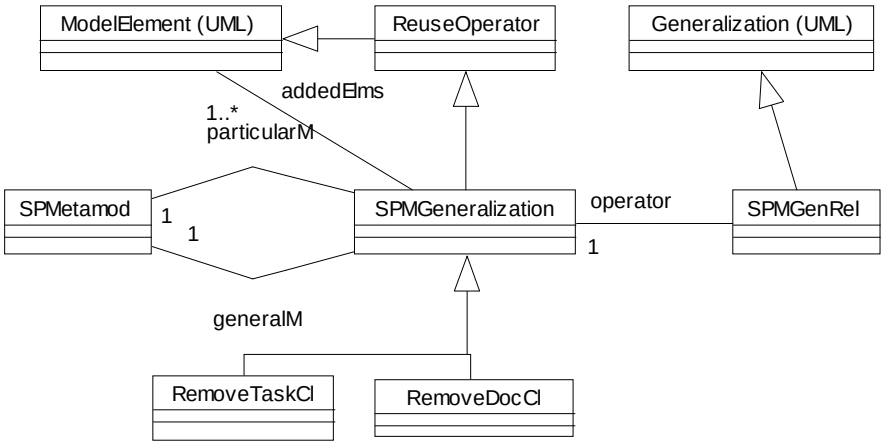


Fig. 4: Extension of the UML metamodel to deal with the Generalization operator (fragment)

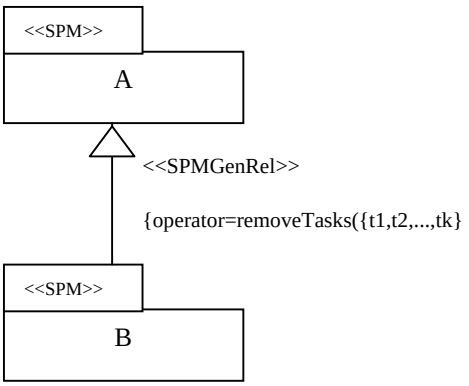


Fig. 5: Representation of the generalization/specialization operators in PROMENADE

6. Renaming operator

A renaming of a model consists of a set of lexical substitutions that are applied to that model. As a result of a renaming, the identifiers of some classes, features, associations, etc of a model may have changed. Renaming may bring about name conflicts (e.g., two different elements may have identical name after a renaming). Therefore, not all renamings should be allowed. These aspects are discussed in the following sections. The *renaming operator* is also defined.

6.1 Unicity of names property

PROMENADE keeps a name policy which ensures that element names in a SPM do not clash. This policy is rooted in the UML name policy, which is based on the notion of *namespace* and on some constraints defined in the UML metamodel for the various UML metaelements. PROMENADE adds some new constraints to those defined in UML. In the following we synthesize the constraints concerning names that are to be kept in a PROMENADE SPM.

1. Constraints at SPM level
   This group of constraints establish requirements about the naming of the elements owned by a SPM (i.e., task classes, document classes, role classes, communication classes, associations and other classes). The origin of the constraint is included.
   (1) All instances of the elements *owned by* a SPM *A* (i.e., *MetaRole, MetaDocument, MetaTask, Communication, Class*) must have different name. They must have different name from the elements defined in *A*'s supertypes as well.
   (UML metamodel definition: Constraint [1] of the *NameSpace* metaclass).
   (2) All the associations defined both in a SPM *A* and in its supertypes must have a unique combination of *<name, associated-classifiers>*.
   (UML metamodel definition: Constraint [2] of the *NameSpace* metaclass).
   (3) There may be no element imported by a SPM *A* with the same name as an element owned by *A* (including its supertypes).
   (UML metamodel definition: Constraints [2] and [4] of the *Package* metaclass).
   (4) The names of the imported elements should not clash.
   (UML metamodel definition: Constraints [3] and [5] of the *Package* metaclass).
2. Constraints at component level
   These constraints apply to different aspects concerning specific elements owned by a SPM. In particular, the constraints are defined on some aspects of classifiers (*MetaDocument, MetaTask, MetaRole, Communication, Class*) and associations.
   (1) The association-ends of a specific association must have unique names.
   (UML metamodel definition: Constraint [1] of the *Association* metaclass).
   (2) All associations defined on a specific classifier *A* (or *A*'s supertypes) must have different names for their opposite association-ends.
   (UML metamodel definition: Constraint [3] of the *Classifier* metaclass).
   (3) All attributes defined on a specific classifier *A* (or *A*'s supertypes) must have different names.
   (UML metamodel definition: Constraint [2] of the *Classifier* metaclass).
   (4) All ECA-rules defined on a specific metatask *T* (or *T*'s supertypes) must have different names.
   (PROMENADE metamodel definition: Constraint of the *MetaTask* metaclass).
   (5) Given a specific classifier *A* (including *A*'s supertypes), there should not be any conflict between the names of *A*'s attributes, opposite association-ends, ECA-rules and owned elements.
   (UML metamodel definition: Constraint [4], [5] of the *Classifier* metaclass

PROMENADE metamodel definition: Constraint of the *MetaTask* metaclass).

(6) No behavioural feature of the same kind may match the same signature in a classifier. (UML metamodel definition: Constraint [1] of the *Classifier* metaclass).

3. Constraints at parameter level
   These constraints apply either to task parameters or to behavioural features parameters.
   (1) All parameters of a behavioural feature must have different names. (UML metamodel definition: Constraint [1] of the *BehaviouralFeature* metaclass).
   (2) All parameters of a task must have different names. (PROMENADE metamodel definition: Constraint of the *MetaTask* metaclass).

We say that a SPM that keeps the referred constraints complies with the *unicity of names property*. Otherwise, it is an *inconsistent SPM* An application of the renaming operator should maintain consistency.

6.2 The Renaming operator

Renaming operator. Definition 1
A renaming of a model *m* consists in the application of a set of lexical substitutions on that model in such a way that the unicity of names property is kept.

$m_2=renaming(m_1,ss)$

ss is a list of substitutions that can be applied to m_1 keeping the unicity of names property. m_2 is the SPM resulting from applying, in parallel, the set of substitutions *ss* to m_1.

A substitution is a pair (s_1/s_2) such that s_1 is the textual element that is substituted for s_2 throughout a SPM.

We present below three conditions that are sufficient in order to ensure that the unicity of names property holds after the application of a set of substitutions.

Proposition 1. Let *m* be a PROMENADE model that satisfies the unicity of names property. Let *ss* be a substitution list to be applied to *m*.
The conjuntion of the following conditions is sufficient to ensure that the model *renaming(m,ss)* keeps the unicity of names property:

(1) For all pair of (s_i/s_k), (s_i/s_t) in *ss*, $s_i=s_r \Rightarrow s_k=s_t$
(2) For all pair of (s_i/s_k), (s_i/s_t) in *ss*, $s_k=s_t \Rightarrow s_i=s_r$
(3) For all (s_i/s_k) in *ss*, s_k is not either in *m* or in SPMs imported by *m* or in a *m* supertype.

- Condition (1) states that an identifier of the model *m* cannot be substituted for more than one identifier. Clearly this condition is a necessary one. Otherwise some indeterminism would be introduced in the application of a set of substitutions to a model.
- Condition (2) establishes that two diferent identifiers of *m* (say s_i, s_r) should be substituted for two different ones (i.e. $s_k \neq s_t$). This condition ensures that the new identifiers introduced in the SPM will not clash. This condition is not strictly necessary in order to ensure unicity of names. Consider, for instance, the situation in which two different attribute identifiers defined in different classes are substituted by the same string without breaking the unicity of names property.
- Condition (3) states that any identifier used to substitute another one should not belong to the SPM (or to an imported one). It constitutes a safe measure for many cases in which the substi-

tution of a model identifier for some other which is already in the model may break the unicity of names. Again, this is not a necessary condition. In situations similar to the presented in condition (2), condition (3) is unnecessarily strict.

Although conditions (2) and (3) certainly ensure that the unicity of names property holds after the application of the renaming operator to a SPM that complies with that property, they are, by far, too restrictive. In particular, they forbid a clash between any pair of identifiers contained in any SPM element. However, the metamodel constraints enumerated in section 6.1 only forbid certain clashes within a specific level (namely, *SPM, Classifier* or *Parameter* level). Therefore, we may redefine the renaming operator by restricting conflicts to the clashes occurring within a specific level and allowing *interlevel* clashes.

This may be enforced by incorporating to the renaming operator a different substitution list for each specific level and enforcing conditions (1), (2) and (3) for each one of the lists.

Renaming operator. Definition 2

We provide an alternative definition for the renaming operator in the following way:

$m'=renaming(m, substsLev_1, substsLev_2, substsLev_3)$

where,

- $substsLev_1$ is a list of name substitutions at SPM level. In particular, it may contain substitutions of:
  - SPM component names (namely, task classes, document classes, communication classes, role clases, other classes).
  - Association names.
- $substsLev_2$ is a list of pairs <*classifier, name-substitution-list*> such that the substitution list refers to *classifier* elements. In particular, this list may contain substitutions of:
  - Names of attributes or behavioural elements of the classifier.
  - Names of elements *owned-by* the classifier.
  - Names of the opposite-end of the associations in which the classifier participates.
  - Names of the associated ECA-rules (if the classifier is a task class).
- $substsLev_3$ is a list that may contain either triples <*classifier, behavioural-element, name-substitution-list*> or pairs <*task-class, name-substitution-list*>. In the first case, the substitution list refers to parameters of a behavioural element associated to a specific classifier, while in the second case it refers to task parameters.

The idea is to enforce the conditions (1), (2) and (3) to each substitution list as a less restrictive way to ensure that the SPM which results from the application of the renaming operator satisfies the unicity of names property. Unfortunatelly, the three conditions do not ensure the constraint that association-ends must have a unique name within each association (constraint 2.(1)). We need also a 4th condition to enforce that all association ends must have a unique name within each association.

Proposition 2. Let *m* be a PROMENADE model that satisfies the unicity of names property. Let *m'* be the SPM that results from the application of the renaming operator:

$m'=renaming(m, substsLev_1, substsLev_2, substsLev_3)$

with,

- $substsLev_1=<m,ss_1>$
- $substsLev_2=\{<cl_1,ss_2>, <cl_2,ss_3>, ..., <cl_k,ss_{k+1}>\}$

- $substsLev_3$={<t_1,ss_{k+2}>, ...,<t_m,ss_m>} (t_i may refer to either a task or a pair <*classifier, behavioural-element*>).

The conjuntion of the following conditions is sufficient to ensure that the model *m'* keeps the unicity of names property:

(1) For all substitution list ss_j *(j=1...m)*, and for all pair of (s_i/s_k), (s_r/s_t) in ss_j, $s_i=s_r \Rightarrow s_k=s_t$.

(2) For all substitution list ss_j *(j=1..m)*, and for all pair of (s_i/s_k), (s_r/s_t) in ss_j, $s_k=s_t \Rightarrow s_i=s_r$.

(3) For all pair of (s_i/s_k) in ss_j *(j=1..m)*, s_k not either in *m* or in a SPM imported by *m*, or in a *m* supertype.

For all pair <cl_r,ss_j>, and for all pair of (s_i/s_k) in ss_j, s_k not either in cl_r or in a cl_r supertype.

(4) The names of the association-ends in any association are unique.

Remarks:

- A classifier name substitution may affect other parts of the SPM (e.g. If a document class name changes, the type of some task class parameters will also change).
- Conditions (1) to (4) still impose some restrictions which are not strictly necessary (e.g., although it is not forbiden by the metamodel, any pair of associations with different names may not be substituted by another pair with the same name. Something similar is true for behavioural features).

Proposition 3. Let *m* be a SPM and let *m'* be another SPM resulting from the application of a renaming operator:

m'=renaming(m,ss), such that ss satisfies the conditions stated in proposition 1 **or**
m'=renaming(m, $substsLev_1$, $substsLev_2$, $substsLev_3$), such that $substsLev_1$, $substsLev_2$, $substsLev_3$ satisfy the conditions stated in proposition 2.

Then, *m* and *m'* are isomorphs.

As a consequence of proposition 3, a *renaming* done on the basis of either proposition 1 or 2 keeps model equivalence (the model that results from a renaming operation is equivalent to the one before the renaming).

6.3 Usage of the renaming operator

The renaming operator may be used to make compatible a pair of incompatible models (i.e. which are incompatibles for lexical reasons) or to simplify a model with redundancies. In particular, it may be used to:

- Separate two different (i.e. non equivalent) classes (of two different models) with the same name (before a combination process. See section 8).
- Give the same name to a pair of equivalent classes belonging to the same model or even different models (useful in a combination process. See section 8).

6.4 Graphical representation

The relationship established between a pair of models *A*, *B* such that, *A* is a renaming of *B* can be represented by means of a special kind of *Dependency*: the *Renaming* dependency (see the

PROMENADE extension of the UML metamodel concerning renaming operator shown in figure 6). Figure 7 shows an example of this representation.

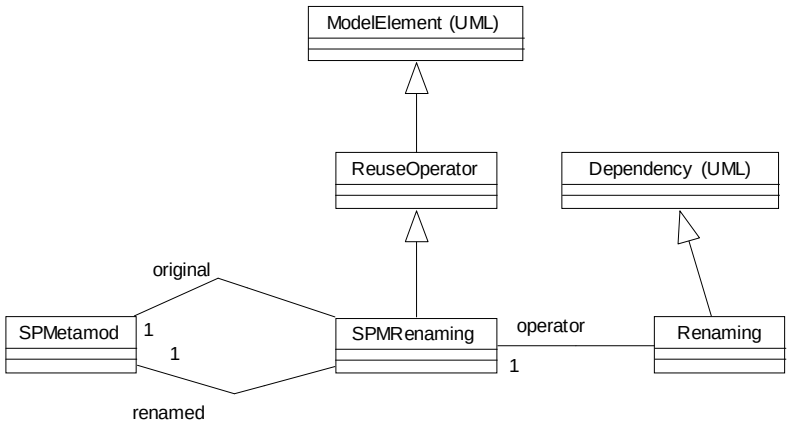


Fig. 6: Extension of the UML metamodel to deal with the *Renaming* operator

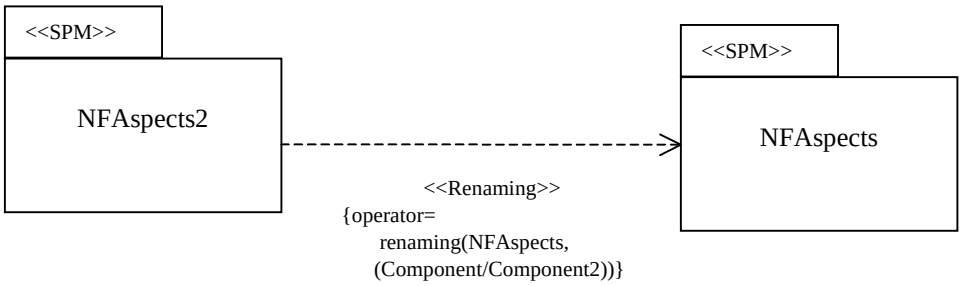


Fig. 7: Representation of the renaming operator

7. Projection operator

The projection of a SPM with respect to a set of classes $\{c_1, c_2, ..., c_n\}$ is constituted by $\{c_1, c_2, ..., c_n\}$ together with the minimum set of additional classes needed by $\{c_1, c_2, ..., c_n\}$ in order to make sense (we call these classes, the *context* of $\{c_1, c_2, ..., c_n\}$).

7.1 Classifier context

Intuitively, the context of a classifier *c* is composed of all the classes that are related with *c* by means of some links (e.g., associations, generalizations, dependencies, etc.).

In particular, we consider the following definitions:

- *Class context:* The context of a class *c* is composed of:
  - the set of classes related to *c* by means of associations (including aggregations)
  - the set of classes on which *c* depends (dependency relationship),
  - *c* superclass
- *Task context:* The context of a task class *c* is composed of:
  - the class context of *c*,

  - *c*'s subtasks,
  - *c*'s precedences,
  - the document classes that act as parameters for *c*,
  - the role class which is responsible for *c* and the role classes that participate in *c*.
- *Document context:* The context of a document class *c* is composed of:
  - the class context of *c*,
  - the role class that is responsible for *c*,
  - the task classes that have *c* as parameter and
  - the subdocument classes of *c*.
- *Role context:* The context of a role class *c* is composed of:
  - the class context of *c*,
  - the set of task classes for which *c* is responsible and in which *c* participates,
  - the set of document classes for which c is responsible.
- *Communication context:* The context of a communication class *c* is composed of:
  - the class context of *c*,
  - the set of task/roles classes that may send/receive the communication.

7.2 Projection of a model with respect to a set of classes

The projection operator allows us to generate model views. A SPM view can be defined in the following way [AC96]:

A view is a projection of a process model according to a well-defined characteristic.

This *well-defined characteristic* may be defined in terms of a set of classes belonging to the SPM (e.g. all the tasks classes for which a given role is responsible, all the tasks that use a documetn class as parameter...).

The projection of a SPM with respect to a set of classes *s* is defined intuitively as the transitive closure of the context of the classes in *s*. That is, the classes that belong to *s* together with all the classes on which they depend directly or indirectly.

More rigorously,

$v=projection(m,\{c_1,...,c_n\})$

v is the model obtained from *m* (*v* is a view of *m*) in the following way:

(1) The PROMENADE reference model belongs to *v*

(2) $c_1,...,c_n$ belong to *v*

(3) If *c* belongs to *v*, *context(c)* belongs to *v*

(4) No other class belongs to *v*.

A maintask for *v* may be chosen between any task class which has no supertask in *v* or, alternatively, a new task class may be defined, which has as subtasks all *v*'s tasks which have no supertask in *v*.

7.3 Graphical representation

The relationship established between a pair of models *A*, *B* such that, *A* is a projection of *B* can be represented by means of a special kind of *Dependency*: the *Projection* dependency (see figure 8). Figure 9 shows an example of this representation.

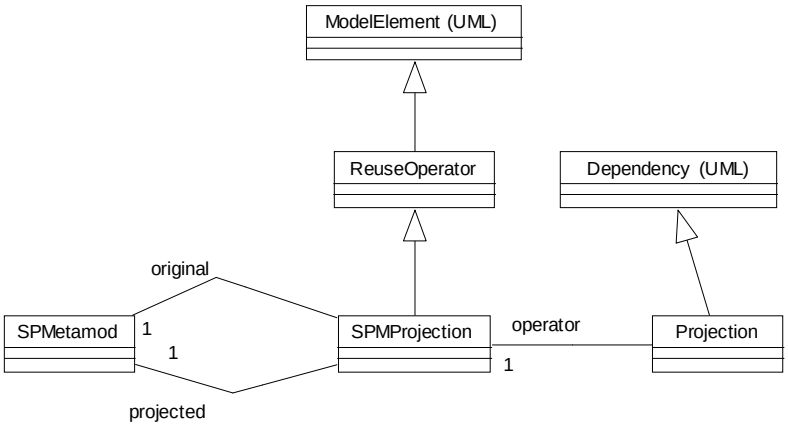


Fig. 8: Extension of the UML metamodel to deal with the *Projection* operator

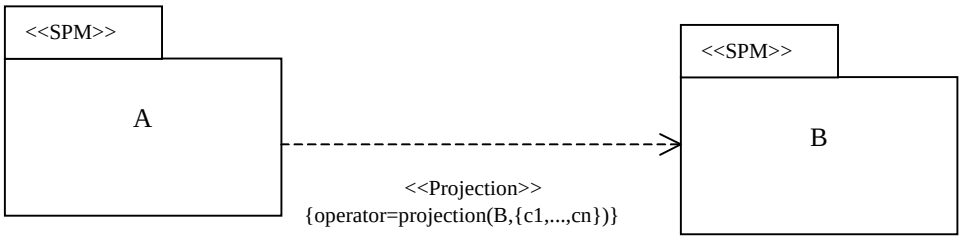


Fig. 9: Representation of the projection operator

8. Composition operators

These operators allow the construction of a new SPM by the composition of some already constructed ones. These existing SPMs may be seen as partial models 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 and with process patterns that will be reused (possibly combined with other process paterns or SPMs) after adaptation. Needless to say that these features will enhance model reuse.

PROMENADE defines three different composition operators: *model grouping, model combination* and *model inclusion.*

- **Model grouping:** This operator gathers together a set of SPMs without adding any additional specific semantics. Furthermore, the namespaces of the component SPMs are kept separated in the resulting SPM (i.e. no merging of namespaces).
- **Model combination:** This operator combines some SPMs by adding several precedence relationships among their main tasks. The namespaces of the component SPMs are merged into a single one. Therefore, component models must be compatible.
- **Model inclusion:** This operator is used whenever the functionality of an entire SPM is to be incorporated into the behaviour of a composite task of another SPM. That is: it allows us to mix models with subtasks in order to describe the behaviour of a composite task. Again, the included SPM must be compatible with the one in which it is included.

In the next few sections we will define these operators more formally.

8.1 Composition operators: SPM grouping operator

SPM grouping operator. Definition
The grouping of two SPMs *A* and *B* generates another SPM *C* that contains precisely *A* and *B* (that is, *A* and *B* are the only elements *owned-by C*).

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

m=grouping(identifier,M)

m is the SPM that contains precisely $m_1,...,m_n$.
The specific meaning of *containment* is given by the *ownedElements* relationship of the UML metamodel, which states which model elements (in this case, SPMs) are owned by a specific namespace[1] (again, a SPM).

Therefore, the previous definition can be rephrased in the following way: *m* is the SPM whose owned elements are exactly $m_1,...,m_n$.

An immediate consequence of this definition is that the namespaces of $m_1,...,m_n$ are not merged into that of *m*. Instead, they are kept separated (independent) within the composite model *m*. Hence, they are preserved without any modification (i.e. the object *x* that belongs to m_1 will be accessed in *m* as $m_1::x$).

Since the namespaces are kept separated and preserved, there is no need for the component models $m_1,...,m_n$ to be compatible.

Notice, finally, that the grouping operator does not introduce any additional semantics to the composite model. This will be different for the combination operator.

SPM grouping operator. Graphical representation
The semantics of model grouping corresponds to the UML notion of package containment. Therefore, a grouping of different SPMs can be represented by packages that are included in another package. Figure 10 represents the SPM that is constructed out of the grouping of SPMs *A* and *B*.

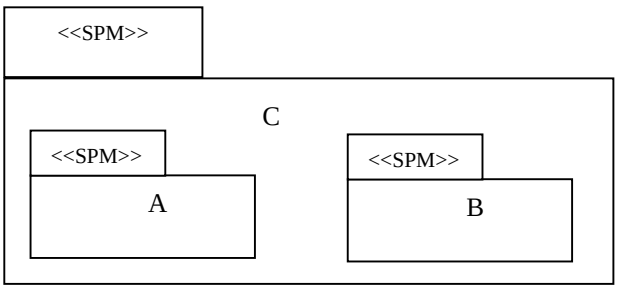


Fig. 10: Graphical representation of the grouping operator *grouping* (*"C"*, {*A,B*})

[1] Recall that in the PROMENADE metamodel *SPMetamod* belongs to the generalization hierarchy rooted at *Namespace*. Recall also that *Namespace* comes from the UML metamodel.

8.2 Composition operators: SPM combination operator

As we have seen, in an intuitive way, a SPM combination consists in gathering together all the elements of both models in a common namespace and provide a specific functionallity to the resulting model. The resulting namespace created by this operator comes from the merging of the component ones. Clearly, this process may lead to conflicts with the names unicity property (i.e. inconsistencies), redundancies and residues.

1. Consistency problems: Two elements that belong to different models have the same identifier but they are not identical (i.e. their definitions differ).
2. Redundancy problems: Two elements in the same model that represent the same concept but have different identifiers.
3. Residue problems: A class that takes part in a composite model since it belonged to one of its component models but which is not required in the composite one.

Coping with inconsistencies
Inconsistencies are clearly the most important problem that we face in the application of the combination operator. If two elements of the same cathegory that belong to two SPMs that are to be combined have the same identifier and different definition will generate an inconsistency in the combined SPM.

Two models without inconsistencies are said to be *compatible.* That is:

Definition. Two SPMs m_1, m_2 are compatible if every pair of elements e_1, e_2 of the same cathegory s.t. e_1 belongs to m_1 and e_2 belongs to m_2 with the same name have identical definition.

Two models can always be made compatible by means of a renaming process (i.e. an application of the renaming operator. See section 6). We may perform a renaming with two different purposes. Let us consider a pair of elements e1 and e2 coming from different models and that have the same name.

(1) If e_1 and e_2 are equivalent (i.e., *isomorphs*), they may be transformed into identical. This choice will be used for those elements e1, e2 that actually refer to the same concept.

(2) If e_1 and e_2 are not equivalent, we may change their names in order to get a pair of elements named differently. This choice will be used for those elements e1, e2 that refer to different concepts.

The combination operator may be applied modulo a renaming in order to get the component models compatibility. For this reason, a list of lexical substitutions is given as a parameter of the combination operator. These substitutions are carried out previously to the process of combination itself (i.e. previosuly to the merging of namespaces).

However, this lexical substitution list could be empty or incomplete (i.e. some inconsistencies could remain in component models). If the combination operator detects some inconsistencies, the combination cannot be completed.

The process of getting rid of the inconsistencies achieved by means of a renaming is called *compatibilization* and is performed by the combination operator before the combination itself.

Coping with redundancies
The compatibilization process enforces that the set of models that are to be combined are compatible but does not ensure the absence of other *combination* problems in the composite model (the model resulting from the combination). These problems are *redundancies* and *residuals.* We deal with both issues when the combination has been performed and a combined resulting model has

been obtained. We remove redundant and residual elements in that model by means of a *simplification* process.

As we have said, an element of a combined model is redundant if it models the same concept as another element already in the model (we call them *corresponding elements* or we say that one is the *counterpart* of the other). Both elements may be equivalent or not (they may be just similar). In any case, since they belong to the same model, they will have distinct names. The first part of the simplification process consists in removing all redundant elements.

Consider a pair of corresponding elements. Two different cases should be distinguished:

(1) Both corresponding elements are equivalent. This case may be addressed by a lexical substitution. Therefore, the list of renamings that are necessary to make both elements identical can be included into the lexical substitution list that we have presented in the previous section.

(2) Both corresponding elements are not equivalent. In this case, we will carry out a simplification process which consists in the suppresion of one of them. This suppression is not trivial since there may exist some other model elements which are not redundant but that refer to the redundant elements. Those references must be substituted for references to counterpart of the redundant one (i.e. the one that will remain in the model). Therefore, prior to the elimination of the redundant element, a substitution process must be carried out.

The combination operator has a parameter that provides a *semantic substitution list*. This is a list of pairs. Each pair contains a redundant element and its counterpart. The combination operator substitutes all references to the redundant elements throughout the model for their counterparts and finally, eliminates the redundant elements.

Coping with residuals

Residuals are elements that come up in the combined model (since they belonged to some component model) and that are not wanted anymore. The second part of the simplification process consists in removing those elements and any reference to them in the combined model.

As a notational tip, we call *intended elements* to all the elements of the combined model that are neither redundant nor residual.

SPM combination operator. Definition

A SPM m is obtained from the combination of a set of models $\{m_1,...,m_n\}$ (from which the inconsistencies have been removed) 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, aggregation and dependency relationships.
- The dynamic part of m is built by combining the maintasks of each model $m_1,...,m_n$ with the precedences $\{a_1,...,a_m\}$.
- In addition, some renamings and simplifications are usually required.

The notion of model combination is presented more formally below.

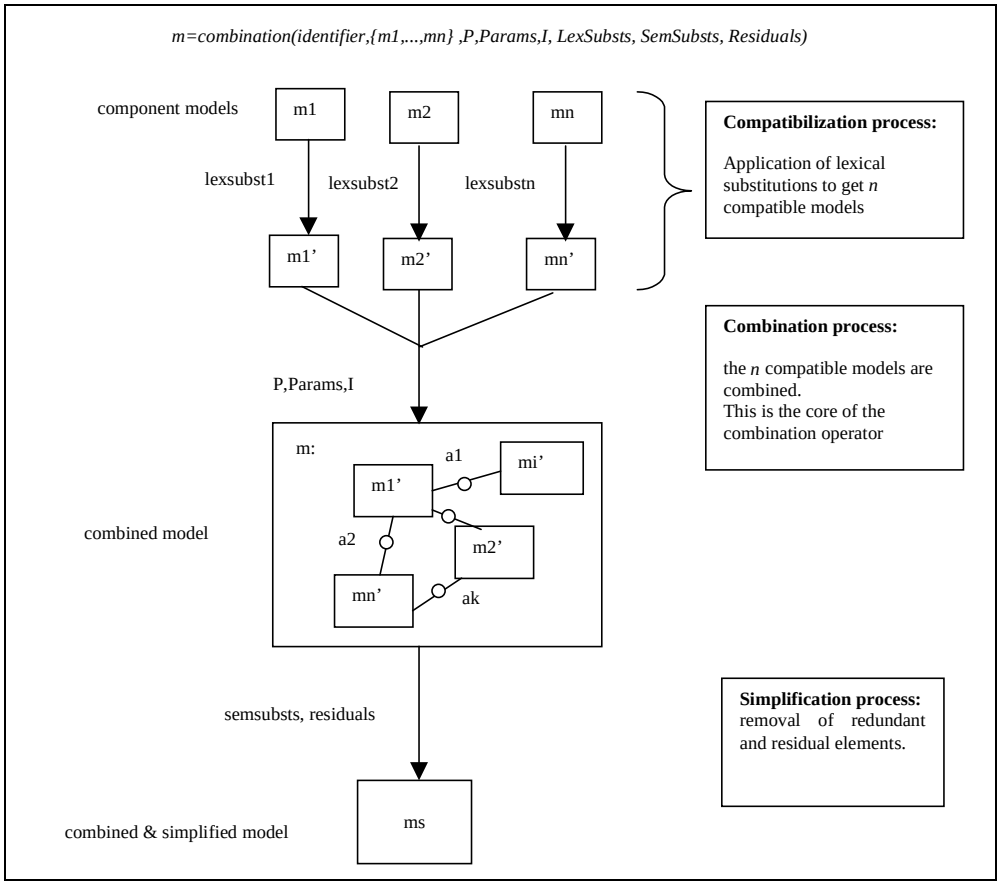


Fig. 11: The whole combination operator

Definition (Combination 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 combination *m* of $m_1,...,m_n$ in the following way:

m=combination(identifier,M ,P,Params,I, LexSubsts, SemSubsts, Residuals)

where,

- *m* is the SPM that results from the combination of $m_1,...,m_n$
- *identifier* is the name of the resulting model.
- $M=\{m_1,...,m_n\}$ is the set of *directly combinable* component models.
- *P* is a set of precedences defined among $m_1.maintask,...,m_n.maintask$
- *Params* is the set of parameters that define the combination model interface.
- *I* is the set of additional invariants for the composite SPM.
- *LexSubsts* is a list of n lists of lexical substitutions that are to be applied to each component model *before* the model combination takes place.
- *SemSubsts* is a list of pairs. Each pair is compounded of a redundant element and its counter-part. This list is necessary to remove redundancies.
- *Residuals* is a list of elements that come from some of the component models and that are not necessary in the resulting one. This list of elements is intended to identify residual classes and get rid of them.

The resulting combined and simplified model *m* is obtained from the application of the following procedure (this procedure is depicted in figure 11):

1. *Apply the substitutions of LexSubsts to the component models.*
   The i[th] list of *LexSubsts* is applied to m_i. We do this by means of n applications of the renaming operator.
   If after the application of the list of lexical substitutions the component models are not compatible yet, the combination operator cannot be applied.

2. *Perform the combination process and get a new model m defined as follows:*

- `m.name=identifier`
- ```
  m.gens=M->iterate (mod: SPMetamod,
                     res: set (Generalization)
            | res->including (id_mod,gen(mod.gens) )
           )
  ```
- ```
  m.assocs=M->iterate (mod: SPMetamod,
                       res: set (Association)
              | res->including (id_mod,assoc(mod.assocs) )
           )
  ```
- ```
  m.docsCl=M->iterate (mod: SPMetamod,
                       res: set (MetaDocument)
               | res->including (id_mod,doc(mod.docsCl) )
           )
  ```
- ```
  m.rolesCl=M->iterate (mod: SPMetamod,
                        res: set (MetaRole)
              | res->including (id_mod,role(mod.rolesCl) )
           )
  ```
- ```
  m.otherCl=M->iterate (mod: SPMetamod,
                        res: set (Class)
               | res->including (id_mod,class(mod.otherCl) )
           )
  ```
- ```
  m.tasksCl=[M->iterate (mod: SPMetamod,
                         res: set (MetaTask)
                 | res->including (id_mod,task(mod.tasksCl) )
              )
          ]->union(set(m.maintaskCl))
  ```
- ```
  m.invs=[M->iterate (mod: SPMetamod,
                      res: set (Predicate)
             | res->including (mod.invs) )
             )
     ]->union(I)
  ```

- *`m.maintaskCl= t`* with:
  - `t.name=identifier`
  - `t.params=PARAMS`
  - `t.precs=P`
  - `t.atomic=false`
  - `t.supertaskCl=NULL`
  - ```
    t.subtasksCl= M->iterate (mod: SPMetamod,
                              res: set (MetaTask)
                         | res->including (id_mod,task(mod.maintaskCl) )
                    )
    ```
  - `t.successCondition=Predicate(subtasks.forAll(t| t.status=completeSucc))`

Where,
id_{m1}: $m_1 \rightarrow m$
id_{m2}: $m_2 \rightarrow m$
...
id_{mn}: $m_n \rightarrow m$
are identity SPM morphisms.

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 (i.e. precedence relationships) the precedence relationships stated in the combination operator.
- The task classes of *m* adds to the union of the task classes of each component model the *m* main task .

3. *Simplify the combined model m that has resulted from step 2.*
   This process consists in performing all the necessary changes in the model *m* to be able to get rid of the redundant and residual classes without any inconsistency.

   Simplification process:
   1. Substitute each association in *m* that links an intended class with a redundant one for another association between the same intended class and the counterpart of the redundant one.
      We will do this by applying the operators of generalization and specialization.
      removeAssociations(m,lis1)
      addAssociations(m,lis2)
      where *lis1* is the list of all associations between an intended class and a redundant one and *lis2* is the list of associations lis1 in which each redundant class has been substituted by its counterpart.

   2. Substitute each association in *m* that links two redundant classes for another association between their counterparts.
      We will do this by applying the operators of generalization and specialization.
      removeAssociations(m,lis1)
      addAssociations(m,lis2)
      where *lis1* is the list of all associations between two redundant classes and *lis2* is the list of associations *lis1* in which each redundant class has been substituted by its counterpart.

   3. Substitute any parameter of the intended tasks of *m* that refer to a redundant document class for a parameter refering to its counterpart.

   4. In each intended task of *m*, substitute any reference to a subtask or supertask classes that belongs to the list of redundant task classes for its counterpart.

5. Identify the precedences of intended composite tasks of *m* that contain any reference to a task class *t* that belongs to the list of redundant task classes and substitute *t* for its counterpart.

6. Substitute the subdocument and superdocument classes of intended document classes of *m* which belong to the list of redundant classes for their counterparts.

7. Substitute the participants and responsibles for intended task classes that refer to redundant roles for their counterparts.

8. Substitute the responsibles for intended document classes that refer to redundant roles for their counterparts.

9. If the maintask class refers to one redundant task class, substitute it for its counterpart.

   At this point there is no reference to a redundant element in any intended element. That is, all associations, parameters, roles, supertasks, subtasks, subdocuments... from an intended element that referred to a redundant one, now refers to their counterpart. Therefore, we can:

10. Remove all the redundant classes.
    Notice that the removal of redundant classes will not do any harm since in previous steps we have substituted all the references to redundant classes for their counterparts.

    removeClasses(MakeComponentNF, Redundants)
    where *Redundants* is the list of redundant classes (obtained from *SemSubsts* getting the head of each of its pairs).

11. Remove all the residual classes.
    Notice that, by definition, the removal of the residual classes will result in the removal of any reference to them along the SPM.
    removeClasses(MakeComponentNF, Residuals)

Following this procedure we may construct a new SPM 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 the strategy of this operator induces a merging of the component namespaces.

Sometimes we are interested only in grouping some models without specifying the behaviour of the composite model, but merging their namespaces. We can also use the combination operator for this purpose. In this case, this operator does not specify either any behaviour or parameters for the main task (i.e. an empty set of precedences and parameters).

Structurally, the relationship between a model *m* that has been obtained from a combination of models $m_1,...,m_n$ and its component models $m_1,...,$mn is established by the association *model-combination* in the PROMENADE metamodel. This association induces, by definition, a dependency stereotyped <<*combination*>> from the composite SPM to the component ones.

SPM combination operator. Graphical representation

There are two complementary ways to represent graphically the combination of a set of SPMs. The first representation focuses on showing the component and the composite SPMs that are involved in the combination (see figure 13, which represents *M3* as the combination of *M1*and *M2*). This representation has the drawback that it does not show the behaviour of the composite SPM. The second one is more specific and it shows the maintask of the composite SPM as a set of precedences between its component SPMs (as we have said, these dependencies take place actually, between the maintasks of the component SPMs). Since we have used a two-tiered approach to the construction of the PROMENADE metamodel (explicit extension of the UML metamodel and creation of a UML-profile; see [RF01, RF02]), there exist two ways to depict the second representation: we can use either the representation that PROMENADE introduces for precedences (see figure 14) or the UML standard notation for dependencies (which will be conveniently stereotyped to denote a specific kind of precedence) accompanied with the appropriate tagged values (see figure 15). Notice that in this representation, arrows are depicted from the client to the supplier to be consistent with the menaning of dependencies in UML. In all cases, parameters are included in attached notes (as we have said, those parameters correspond to parameters of the maintask of the respective SPMs). Figures 14 and 15 show that the main task of *M3* may be described as a strong precedence from the main task of *M1* to the main task of *M2*. Figure 12, below, depicts the metamodel extension to deal with *grouping* and *combination* operators.

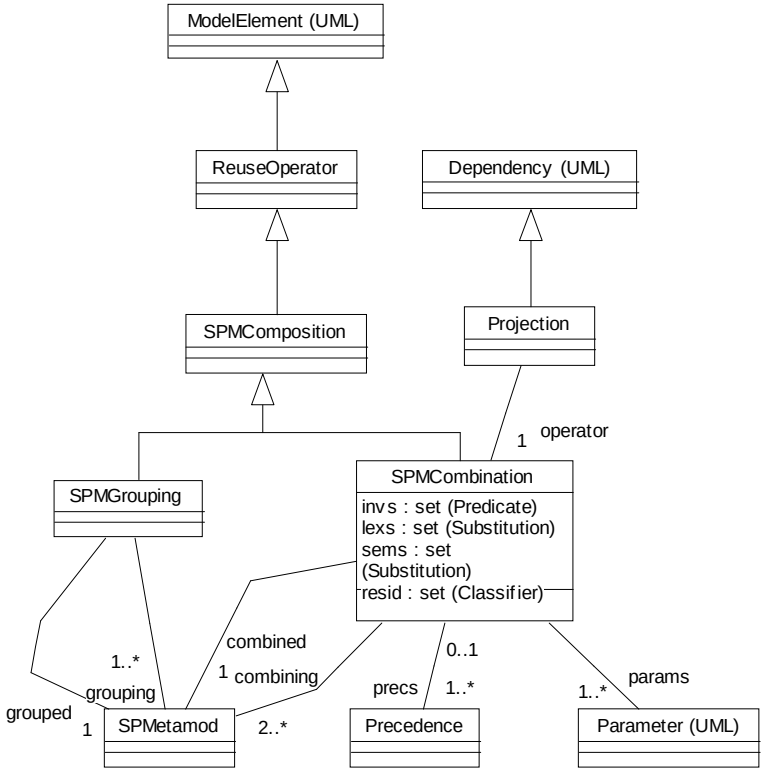


Fig. 12: Extension of the UML metamodel to deal with two *Composition* operators

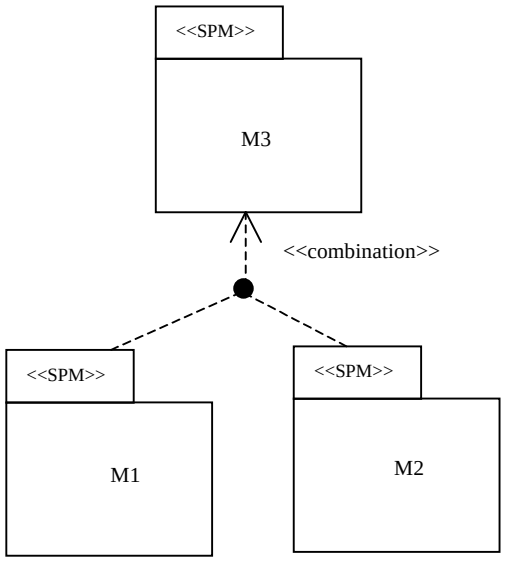


Fig. 13: *M3* obtained as a combination of *M1* and *M2*

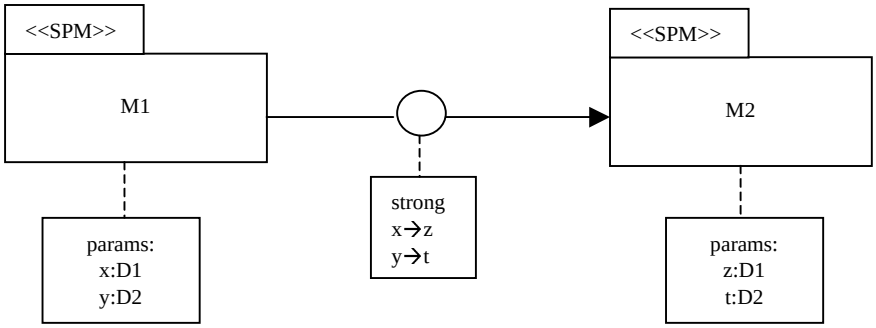


Fig. 14: Representation of the behaviour of a combined SPM using the PROMENADE notation

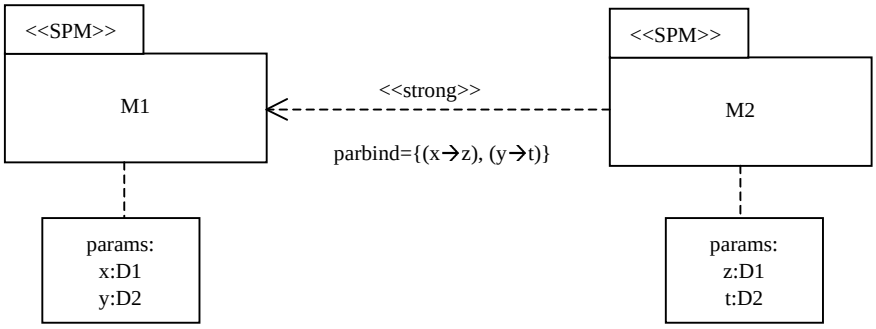


Fig. 15: Standard UML representation of the behaviour of a combined SPM

Model combination vs. model grouping

Whenever a composition between a group of SPMs is required we may apply either the model combination or the model grouping operators. In this section we try to state which is the best choice according to the particular situation.

There are two main differences between both operators:

(1) The combination operator allows the definition of some additional behaviour to the composite model (i.e. some precedences between the main tasks of the component models)

(2) Both operators create a new namespace for the composite model. However, while the grouping operator actually keeps the namespaces of the component models separate and independent (i.e. it is possible to refer, within the composite model to each component namespace. E.g. m1::A), the combination operator merges them into a single and integrated namespace. The conflicts that may arise when this operator is applied are avoided by requiring that the combined models must be directly combinable. We may say that the grouping operator leads to a loose composition while the combination operator produces a tighter and more integrated one.

In particular, notice that if we want to merge the equivalent (or similar) elements of the component models into a single element in the composite model, we will choose the combination operator. The grouping one would keep those similar or equivalent concepts separated and independent.

8.3 Inclusion operator

Sometimes, during the design of the behaviour of a composite task *t* that belongs to a model *m*, we notice that an already constructed model m_2 provides the functionality that is required for a specific *t*'s subtask.

The inclusion operator is used in these cases, that is: whenever the functionality of an entire model m_2 is to be incorporated as a subtask into the behaviour of a composite task *t* (which belongs to another model *m*), possibly with some additional precedences.

Therefore, we define:

m'=inclusion(m,t,m2,beh)

where m and m2 are PROMENADE SPMs, t is a task class of m and beh is a set of precedences linking m2's main task with some t's subtasks.

with the following meaning:

- The structure of m' is composed of the structural elements of m and those of m2. The main task of m2 is incorporated as a subtask of t in the new model m'.
- The behaviour of m' is that of m with the additional precedences beh which have been incorporated to t.

That is, the inclusion operator allows us to mix models with subtasks in order to describe the behaviour of a composite task.

This operator may be used, not only in the modelling phase, but also in the enactment one in connection with the delayed binding capability. When at enactment time, an activity t whose behaviour has not been established at modelling time has to be refined, an entire model (coming from

a library of reusable models) may be chosen to provide the behaviour for t. In this case, the inclusion operator is used.

Formal definition

Let *m* and m_2 be two compatible SPMs. Let *t* be a task class in *m*. Let *beh* be a set of precedence relationships established between subclasses of *t* and, possibly, the main task class of m_2.

We say that *m'* is the model obtained from the inclusion of m_2 in the task *t* of *m* with the precedence relationships *beh*, which we note:

$m'=inclusion(m,t,m_2,beh)$

In order to define *m'* we consider two model morphisms:
$f:m \rightarrow m'$ and
$f2:m_2 \rightarrow m'$
defined in the following way:

$f=(id_{name}, f_{task}, id_{doc}, id_{role}, id_{class}, id_{gen}, id_{comm}, id_{assoc}, id_{dep}, f_{mainta})$
$f2=(id2_{name}, f2_{task}, id2_{doc}, id2_{role}, id2_{class}, id2_{comm}, id2_{gen}, id2_{assoc}, id2_{dep}, f2_{mainta})$
with,

```
f_task(s,t)=
     if (s≠t) then result=id_task(s)
     else result.name=s.name
            result.subtasksCl=ID_task(s.subtasksCl)->union(f2_task (m2.maintaskCl))
            result.precs=transfoPrecs(t.precs)->union(transfoPrecs(beh))
            result.params=ID_param(s.params)
            result.supertaskCl=f_task(s.supertaskCl, t)
            result.atomic=false

f_maintask(m.maintask)=f_task(m.maintask, t)

f2_task(s)=
     if (s≠m2.maintaskCl) then result=id2_task(s)
     else result.name=s.name
            result.subtasksCl=ID2_task(s.subtasksCl)
            result.precs=ID2_precs(s.precs)
            result.params=ID2_param(s.params)
            result.supertaskCl=f_task(t,t)
            result.atomic=s.atomic
f2_maintask(m_2.maintask)=f2_task(m_2.maintask)

transfoPrecs(sp)=sp->iterate(p:Precedence, res:set(Precedence)
                          |res->including(transfoP(p))
                          )

transfoP(p)=
  if (p.supplier=m2.maintaskCl)  result.supplier=set{f2_task (m2.maintaskCl)}
                                 result.client=F_task(p.client)
  else if (p.client=m2.maintaskCl) result.client=set{f2_task (m2.maintaskCl)}
                                   result.supplier=F_task(p.supplier)
  else      result.supplier=F_task(p.supplier)
            result.client=F_task(p.client)
```

Once these model morphisms have been defined, we can easily define *m'*:

```
m'.name=m.name
m'.gens=ID_gen(m.gens)->union(ID2_gen(m2.gens))
m'.assocs=ID_assoc(m.assocs)->union(ID2_assoc(m2.assocs))
m'.deps=ID_dep(m.deps)->union(ID2_dep(m2.deps))
```

```
m'.docs=ID_doc(m.docs)->union(ID2_doc(m2.docs))
m'.rolesCl=ID_role(m.rolesCl)->union(ID2_role(m2.rolesCl))
m'.othersCl=ID_class(m.othersCl)->union(ID2_gen(m2.othersCl))
m'.tasksCl=F_task(m.tasksCl,t)->union(F2_task(m2.tasksCl))
m'maintask=f_task(m.maintaskCl,t)
```

According to the above definition, the new model *m'* will contain the generalizations, associations, dependencies, communications, document classes, task classes, role classes and other classes of *m* and m_2. Its main task will be that of *m*. Most of the task classes in both *m* and m_2 will remain the same in *m'* (notice the *identity* morphisms applied); however, task class *t* will incorporate a new subtask (namely, the main task class of m_2) along with the precedences established in *beh*. In its turn, the main task class of m_2 will keep the same structure and exhibit the same behaviour. It will incorporate, however, *t* as its supertask class.

Notice that the inclusion operator may be used as well to define a specific behaviour for a not refined task class.

m'=inclusion(m, t, m_2, {})

where *t* is a not refined task class of *m*. According to the inclusion operator the behaviour of *t* in *m'* will be that of m_2. Notice that no precedence set is necessary in this case.

Graphical representation

A SPM that results from the inclusion of a SPM m2 into a composite task t, with additional behaviour beh is represented graphically by means of precedences between m2 ant t's subtasks. These precedences will be the ones stated in beh. Alternatively, a standard graphical representation may be taken by means of stereotyped dependencies (<<inclusion>>) between m2 and the t's subtasks (see figure 16).

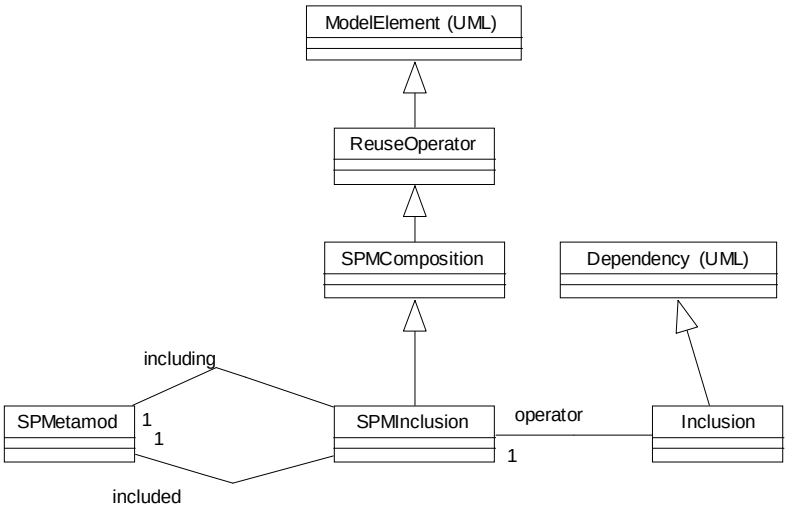


Fig. 16: Extension of the UML metamodel to deal with the *Inclusion* operator

9. Delayed binding

The delayed binding ability in PROMENADE may take two different forms:

(1) A substitution at enactment time a task/document class A fo the SPM for some A's refinement (i.e. another task/document class from the SPM that specializes A). A specific selection may be enforced by other previous selections. This is discussed in [RF01].

(2) A SPM may be parameterized (i.e. it may be a process pattern). As a result, some of its generic elements may be instantiated at enactment time. In this case, the actual parameters used for the instantiation will come from a library of SPMs (or process patterns) for reuse. The *inclusion* operator is likely to be used.

10. An example of process reuse

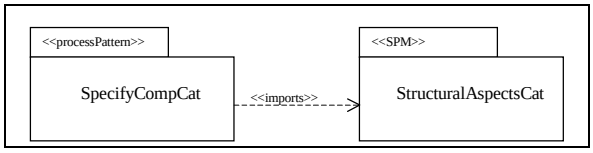
Catalysis [DC99] is a collection of software development processes which are described in a modular way by the application of several patterns. These patterns share a common structure (i.e., the static elements used by them, like *Spec*, *Collaboration*, *Design*, are supplied by Catalysis). They are described informally in natural language and may be used in the description of one or more Catalysis processes.

One of the Catalysis patterns which is used more recurrently in different processes is the one intended to specify a component (pattern 15.7 in [DC99]; let us call it *SpecifyCompCat*). This pattern relies on the notions of *Collaboration* and *SpecType* in order to construct the specification for a component referred to a specific role. The strategy followed by this pattern is the following: (1) definition of the static model, which provides the vocabulary to specify the operations; (2) specification of the operations; (3) assignment of responsibilities.

Figure 17 shows a definition of this pattern in PROMENADE. It contains no parameters since it is not intended to be used universally but only within the Catalysis processes (i.e., the structural elements, including *Collaboration* and *SpecType* are defined in the Catalysis context and imported by all the patterns). Notice that PROMENADE allows a rigurous definition of the pattern. In addition to the usual textual features included in most pattern languages (*author, keywords, intent, etc.* [Amb98]; see fig. 17), other aspects described with a more precise (and standard) formalism are included.

In particular, the pattern structural elements are presented by means of a UML class diagram (this diagram is encapsulated in *StructuralAspectsCat* and it is shared by all the Catalysis processes); its behaviour is described by means of a set of precedence relationships between the component activities (which may be decomposed in other subactivities). These relationships establish some temporal precedences between activities; include some parameter binding to link the documents involved in them; and are depicted by means of stereotyped UML dependencies. See [RF00, RF01] for a complete description. The details of the semantics are not necessary in the rest of the paper. The behavioural description of this pattern for the sake of brevity. However, an example of process behaviour description is shown in figure 18.

Catalysis does not support the statement of software quality attributes (i.e., non-functional requirements [ISO99], short NFR) in a component specification. As other approaches do [CNM99, CL99], it can be considered useful to incorporate such NFRs in that specification. We have done this in [RF01]. Figure 18 shows the behaviour description for the SPM *SpecifyCompWithNF*.



Name: SpecifyCompCat
Author: JMR & XFG
Keywords: ***software component, specification, Catalysis.***
Intent: To incorporate into a specification document (SpecType) the functional specification of a software component according to a specific role.
Pattern parameters: No parameters defined.
Pattern structure: Imported from *StructralAspectsCat*
Initial context: A component has to be designed and implemented. The participants should be available and ready to start the specification task.
Input parameters:

- A collaboration document corresponding to a specific role (*col: Collaboration*)
- The specification document (partially constructed) (*spec: SpecType*).

Result context: The specification document of the component has been built.
Output parameters:

- A specification document that contains the component specification corresponding to a specific role. (*spec:SpecType*).

Applicability: No additional constraints have been defined.
Participants: One or more members of the specifier team
Structure: The structure shared by all Catalysis processes (imported from *StructuralAspectsCat*). Not shown
Process: Not shown.
Related patterns: ComponentSpecification

Fig. 17: Definition of the *SpecifyCompCat* pattern

SpecifyCompWithNF is a specific SPM which states that a component specification should consist of a functional and a non-functional specifications and also a validation of the specification. This SPM proposes the strategy of performing the functional specification first. It also applies a concrete strategy for specifying the functional and the non functional requirements. The details of this strategy are encapsulated in the subtasks of *FSpecifyComp* and *NFSpecifyComp*.

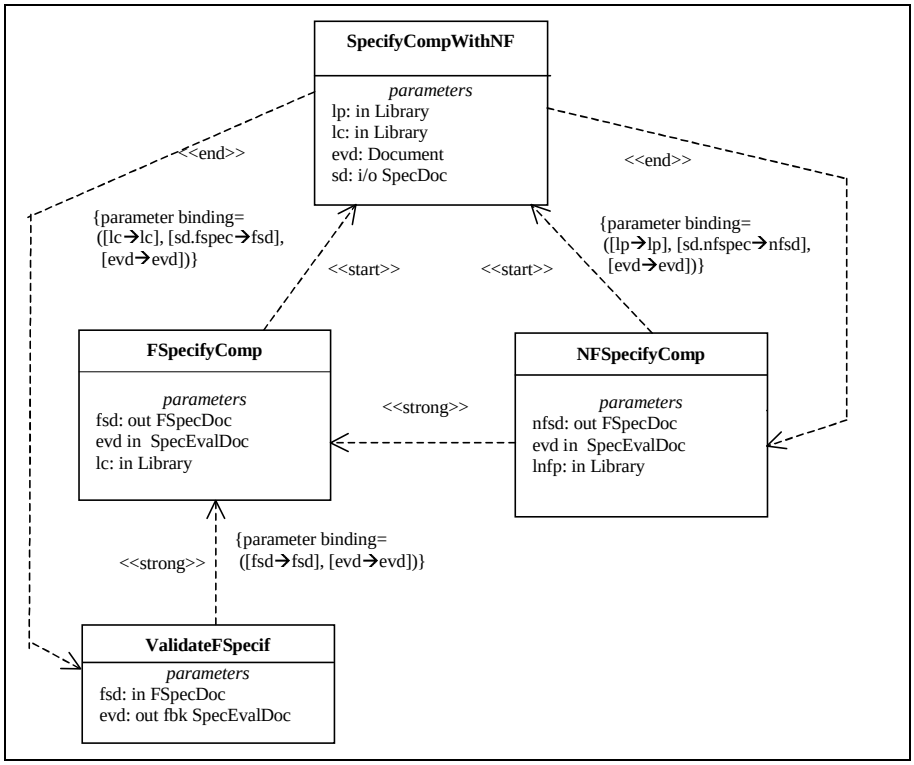


Fig. 18: Behaviour description of the model *SpecifyCompWithNF*.

In the current situation, it is interesting to incorporate NFRs into the *SpecifyCompCat* Catalysis process pattern by reusing *SpecifyCompWithNF*. Not all the steps that we will follow are strictly necessary but, in this way, we will illustrate how various reuse operators may be composed along the reuse framework (see figure 1). The proposed operator composition process is shown in fig. 19. Notice that the diagram depicted in this figure is standard UML. The relationships between the various models are represented by means of UML dependencies. The sense of the arrows has been selected according to the UML semantics. The UML metamodel has been extended with several stereotypes and tagged values using the standard UML extension mechanisms (see [UML01]). The steps followed in this reuse process are detailed next.

Step 1. Abstraction of *SpecifyCompWithNF→SpecifyCompWithNFPattern*

First of all, we will build the process pattern *SpecifyCompWithNFPattern* which abstracts the particular specification strategy so that different functional and non-functional methodologies can be used in each particular situation. This abstraction process is carried out by means of a *parameterization*. The result in shown in fig. 20. By the parameterization of the documents and tasks that are necessary to perform this pattern, many different specific strategies to carry out a specification may be used to instantiate it (e.g., by diagrams, by templates, purely textual, formal –model-oriented, equational, etc..). To preserve consistency, some requirements are established on these parameters (e.g., the document type that instantiates the parameter specification document *SpecDoc* must have as subdocuments, a functional specification and a non-functional specification documents). These requirements have been expressed in OCL. Notice again that the model for the process pattern is not described just textually but in a rigorous manner using the PROMENADE PML constructs (see fig. 20), which have been mapped into UML. In particular, notice that PROMENADE allows the rigorous definition of parameters by means of their types and of constraints for the parameter instantiation. Notice also that other patterns which propose a different component specification strategy or which do not perform a non-functional specification may coexist with the proposed one.

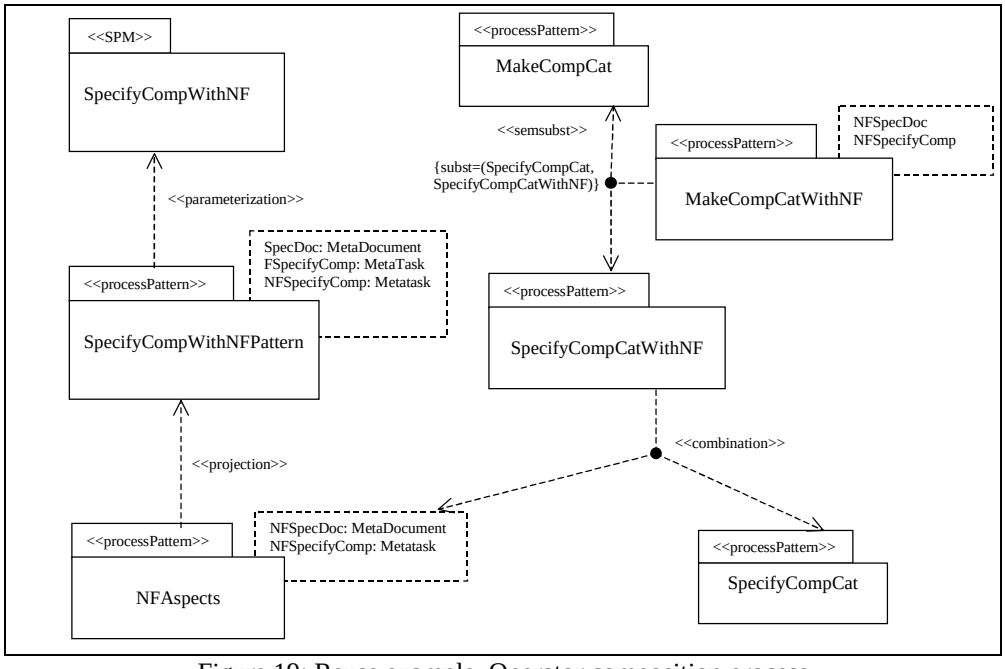
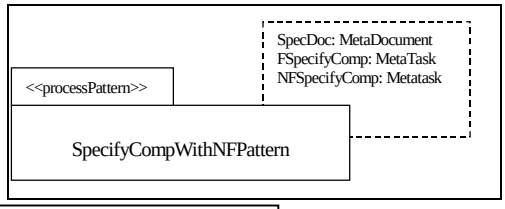


Figure 19: Reuse example. Operator composition process



Name: SpecifyCompWithNFPattern
Author: JMR & XFG
Keywords: ***software component, specification, software attributes, component, non funcionality.***
Intent: To provide a document (SpecDoc) which contains the functional and non-funcitonal specification of a software component; or to review a previous specification.
Pattern parameters:

- The specification document type (*SpecDoc*).
- A task to perform the functional specification (*FspecifyComp*)
- A task to perform the non functional specification (*NFspecifyComp*)

Initial context: A component has to be designed and implemented. The participants should be available and ready to start the specification activity.
Input parameters:

- A library of components (*lc: Library*)
- A library of non-functional properties (*lp: Library*) (optional)
- An evaluation document (if it is a specification revision) (*evd: Document*) (optional)
- The specification document to be reviewed (if it is a specification revision) (*sd: SpecDoc*).

Result context: The specification document of the component has been built.
Output parameters:

- The specification document completed (*sd: SpecDoc*).

Applicability: Some constraints have been defined on the pattern parameters:
(1) SpecDoc.allSupertypes->includes(Document)
(2) Specdoc.subdocs->includes(FSpecDoc)
(3) Specdoc.subdocs->includes(NFSpecDoc)
(4) FSpecifyComp.allSupertypes->includes(Task)
(5) FspecifyCom.params->exists(p|p.type=FSpecDoc)
(6) NFSpecifyComp.allSupertypes->includes(Task)
(7) NFspecifyCom.params->exists(p|p.type=NFSpecDoc)
Participants: One or more members of the specifier team
Structure: (not shown)
Process: (not shown)

Fig. 20: The *SpecifyCompWithNFPattern* process pattern

Step 2. Adaptation of *SpecifyCompWithNFPattern→NFAspects*

We do not want to incorporate the whole functionality of *SpecifyCompWithNFPattern* into *SpecifyCompCat*; only the non-functional aspects. For this reason we will adapt this pattern. The idea is to perform a projection of *SpecifyCompWithNFPattern* onto its non-functional aspects. Therefore, we apply the *projection* operator, once the elements on which the projection is carried out have been identified.

The projection is carried out on the elements that provide the non-functional component specification, namely: *NFSpecDoc*, as document class; *NFSpecifyComp*, as task class; and *Library*, as other cathegory of classes. Therefore, the following operator usage must be performed: *NFAspects* = projection(*SpecifyCompWithNFPattern*, {*NFSpecDoc, NFSpecifyComp, Library*}).

The resulting model (*NFAspects*) will keep (by definition of *projection* in PROMENADE) the context of the classes on which the projection has been performed. As a result, other classes like *FSpecDoc* and *SpecEvalDoc* will belong to *NFAspects* (since they are parameters of the task class *NFSpecifyComp*, hence, part of its context). Something similar will happen with *ValidateFSpec* (since this task class have *FSpecDoc* as parameter).

Step 3. Combination of *SpecifyCompCat* with *NFAspects→SpecifyCompCatWithNF*
This is the central step of the reuse process we are following. The specification of the NF aspects will be combined with the Catalysis pattern that models the specification of a software component. The new pattern (*SpecifyCompCatWithNF*) will establish some additional behaviour: the functional specification should be made before the non-functional one.

Some considerations must be made prior to the application of the combination operator:

(1) Study of compatibility. Since there are no name conflicts between models, it is possible to put them together. If this were not the case, some preliminar rearrangements should be made.

(2) Study of redundancies. We look for semantical equivalences in the involved models. We find that two types of documents with different names are equivalent (from the new model point of view), because they both refer to the functional specification of a component. We build thus a set of pairs (in this case, just one) that will be passed as a parameter of the combination operator. Therefore, we have *semsubsts*={(*FSpecDoc, SpecType*)}. Instances of (our) *FSpecDoc* class will be substituted by equivalent ones from the (Catalysis) *SpecType*

(3) Identification of residual classes. We look for elements that become unnecessary in the new model, in order to remove them. We identify as residual classes all those coming from *NFAspects* that are not related to functional capabilities and that are not redundant classes either. In this way we have *residuals*={*SpecEvalDoc, ValidateFSpec, Library*}.

(4) Statement of precedence relationships between the main tasks of the component models. Both models become coordinated from the behaviourial point of view. We choose to apply the non-functional specification right after the functional one. Therefore, we provide the following precedence list: *precs*={(*strong*, [*SpecifyCompCat*], [*NFAspects*])}. This list contains just one precedence relationship of type *strong* which ensures that the non-functional specification of a component will start only after the successful end of the functional one.

(5) We will choose the parameters of the composite model to be the same as those of the *SpecifyCompCat* with the addition of the non-functional specification document. *params*={(*col, in, Collaboration*), (*sp, out, SpecType*), (*nfsp, out, NFSpecDoc*)}

Suming up, the operator is applied as: *combination(SpecifyCompCatWithNF, {SpecifyCompCat, NFAspects}, precs, params, λ, semsubsts, residuals).*

Step 4. Composition of *SpecifyCompCatWithNF* within *MakeCompCat*

The new pattern *SpecifyCompCatWithNF* may be used within other Catalysis processes. If these processes have not been constructed yet, *SpecifyCompCatWithNF* will be incorporated in the moment of their construction by means of the *inclusion* operator. Otherwise, the *semantic substitution* operator may be used. For instance, the main task of *MakeCompCat* (an already constructed model to build a component using the Catalysis methodology) includes the model *SpecifyCompCat* as a part of its funcitonality. This model may be substituted for the pattern *SpecifyCompCatWithNF* by the application of the semantic substitution operator.

11. Conclusions

We have presented a general and expressive process reuse framework focused on the mechanisms that are necessary in order to achieve harvesting and reuse of processes. In this respect, we have defined a wide range of reuse mechanisms and we have adapted these mechanisms to a specific PML called PROMENADE. This PML particularizes virtually all the mechanisms defined in the framework and provides a standard UML representation for them. Finally, we have outlined a reuse example consisting in the incorporation of non-functional requirements into the specification of a software component (a specification obtained using the Catalysis methodology). The example has been presented in PROMENADE. Some aspects of our approach have been issued in it: the joint application of reuse operators following a path along the framework depicted in fig. 1; the expressiveness of those operators and the standard UML representation of the relationships between the models involved in the reuse process.

[JC00, RRN01, Per96] present other reuse frameworks. Essentially, these frameworks identify a reuse life-cycle together with the requirements for a PML in order to supply reuse capabilities

(therefore they are not restricted to mechanisms). [JC00] and [RRN01], the most complete ones, identify *generalization*, *inheritance*, *composition*, *projection* and *parameterization* to be the required reuse and harvesting mechanisms that a PML should provide. However, they just present them as general mechanisms. They do not provide particular definitions of them in the context of a PML. Notice that we introduce a systematic enumeration of mechanisms and we identify some new ones (i.e., *inclusion*, *renaming*, *semantic substitution*, different types of *composition*, etc.). We also propose an expressive parameterization mechanism (including the definition of constraints concerning parameters) and we propose specific definition of each mechanism in the context of PROMENADE. Furthermore, we provide a more general definition of *harvesting* and *reuse*: we do not restrict the harvesting (reuse) notion to the generation of a more abstract (specific) model from a more specific (abstract) one; a composition of several models at the same abstraction level may also be a harvesting/reuse activity.

In SPM, our field of study, modularity-reuse abilities provided by PMLs are scarce. E^3 [Jacc96], OPSIS [AC96], PYNODE [ABC96] and [EHT97] provide limited reuse capabilities (mostly based on views and generalization/inheritance in the case of E^3). They do not offer expressive ways to combine already constructed models (e.g., building the behaviour of the composite model as a customizable combination of the behaviours of the components). They do not offer process pattern support either.

In the related area of workflow management, many approaches use just the *cut-and-paste* strategy to deal with the topic of reuse [Kru97]. However, there exist some PMLs, like In-Concert [INCO96], OBLIGATIONS [Bog95], APM [Car97, Kru97], MOBILE [HHJ99] that address it in a more sofisticated, although limited way, even in the case of APM, which is the most powerful one. It provides a top-down approach to reuse-based model construction. A model is a pattern with some indefined activities. These activities may be substituted by adaptable template fragments. However, it does not support either a systematic pattern definition or a bottom-up reuse strategy (e.g., model composition). Overall, these languages offer a poor approach to the modelling of process patterns [JC00].

In summary, although some approaches that support reuse do exist in both the fields of SPM and workflow management, they do not provide all the mechanisms required in the frameworks by [JC00, RRN01]. On the other hand, some mechanisms like *composition* are only supplied by few PMLs and using not very expressive approaches (e.g., superposition). Other constructs, like parameterization and support for process patterns are scarce and poorly achieved [JC00]. We are not aware of any PML, both in the fields of SPM and workflow management, which defines a reuse framework endowed with all the mechanisms we have outlined in section 2. Furthermore, to our knowledge, no PML uses a standard notation to express reuse abilities.

REFERENCES

[ABC96] Avrilionis, D.; Belkhatir, N; Cunin, P-Y. Improving Software Process Modelling and Enacting Techniques. In C. Montagnero (Ed.) Proc. of the 5th. European Workshop on Software Process Technology (LNCS-1149). Nancy, France. October, 1996.

[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.

[Amb98] Ambler, S.W.: Process Patterns: Building Large-Scale Systems Using Object Technology. New York: SIGS Books/Cambridge University Press.

[Bog95] Bogia, D.P.: Supporting Flexible, Extensible Task Descriptions In and Among Tasks. Ph. D. thesis from Dept. iof Compuer Science, University of Illinois at Urbain Champaign, 1995.

[Car97] Carlsen, S. "Conceptual Modelling and Composition of Flexible Workflow Models", PhD-thesis, NTNU - Norwegian University of Science and Technology, Trondheim, Norway, 1997.

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

[CL99] Cysneiros, L.M.; Leite, J. "Integrating Non-Functional Requirements into Data Modeling". Procs. 4th ISRE, June 99, Limerick (Ireland).

[CNM99] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, ISBN 0-7923-8666-3. October 1999.

[DC99] D'Souza D.F.; Cameron, A: Objects, Components and Frameworks with UML. The Catalysis Approach. Addison Wesley, 1999.

[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.

[GM01] Gnatz, M.; Marschall, F.; et alt. Towards a Living Software Development Process Based on Process Patterns. In LNCS 2077, p. 182 ff. Springer Verlag, 2001.

[HHJ99] Heinl, P.; Horn, S.; Jablonski, S. et alt.: A Comprehensive Approach to Flexibility in Workflow Management Systems. In proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC'99), San Francisco, USA, 1999.

[INCO96] InConcert 3.0 product information. http://www.xsoft.com/XSoft/products/ict/ic30.html

[ISO99] ISO/IEC Standards 9126 (Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their use, 1991) and 14598 (Information Technology – Software Product Evaluation: Part 1, General Overview; Part 4, Process for Acquirers), 1999.

[Jacc96] Jaccheri, M.L. "Reusing Software Process Models in E3", IEEE International Software Process Workshop 10, Dijon France, June, 1996.

[JC00] Jørgensen H.D.; Carlsen, S, Writings in Process Knowledge Management: Management of Knowledge Captured by Process Models, SINTEF Telecom and Informatics, Oslo STF40 A00011, ISBN 82-14-01928-1, 2000-01-27.

[Kal96] Kalinichenko, L. Leonid A. Kalinichenko: Type Associations Identified to Support Information Resource Reuse in Megaprogramming. In Proceedings of the Third International Workshop on Advances in Databases and Information Systems, ADBIS 1996, Moscow, Russia, September 10-13, 1996.

[Kru97] Kruke, V: Reuse in Workflow Modeling Diploma Thesis. Department of Computer Systems. Norwegian University of Science and Technology. 1997

[Per96] Perry, D.E.: Practical Issues in Process Reuse. In proceedings of the International Software Process Workshop, 10 (ISPW'10). June, 1996

[PTV97] Puutsjärvi, J.; Tirry, H.; Veijalainen, J. Reusability and Modularity in Transactional Workflows Information Systems. Vol. 22 N. 2/3 pp. 101-120, 1997.

[RF00] Ribó J.M; Franch X.: PROMENADE, a PML intended to enhance standardization, expressiveness and modularity in SPM. Research Report LSI-00-34-R, Dept. LSI, Politechnical University of Catalonia (2000).

[RF01] Ribó J.M; Franch X.: Building Expressive and Flexible Process Models using an UML-based approach. Proceedings of the 8th. European Workshop in Software Process Technology. Witten (Germany). Lecture Notes in Computer Science (LNCS), Vol. 2077, pp. 152-172. Springer-Verlag (2001).

[RF02b] Ribó J.M; Franch X.: A two-tiered Approach for Extending the UML metamodel. To be submitted to the 21st. International Conference on Conceptual Modeling (ER-2002).

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

[RRN01] Reis, R; Reis, C; Nunes, D.: Automated Support for Software Process Reuse: Requiremetns and Early Experiences" in Proceedings of the 7th International Workshop on Groupware (CRIGW-01) Darmstadt (Germany) September-2001.

[Sto01] Störrle, H.: Describing Process Patterns with UML. In LNCS 2077, p. 173 ff. Springer Verlag, 2001.

[SW01] Sa, J.; Warboys, B. et alt.: Modeling a Support Framework for Dynamic Organizations as a Process Pattern Using UML. In LNCS 2077, p. 203 ff. Springer Verlag, 2001.

[UML01] Unified Modelling Language (UML) 1.4 specification. OMG document formal/ (formal/2001-09-67). September, 2001.