Searching for Expressiveness, Modularity, Flexibility and Standardisation in Software Process Modelling

Josep M. Ribó, Xavier Franch

1 Universitat de Lleida
C. Jaume II 69, 25001 Lleida (Catalunya, Spain)
fax: +34.973.70.27.02
josepma@eup.udl.es

2 Universitat Politècnica de Catalunya (UPC),
c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
fax: +34 93 401.70.14
franch@lsi.upc.es

Abstract:
Although an important research effort has been carried out in the last decade in the field of software process modelling (SPM), some crucial issues still remain as challenges to the community. The expressive power of process control-flow descriptions in most current approaches is still not optimal; their capabilities to provide a flexible process model in which it is possible to perform "on-the-fly" modifications in a safe manner are often insufficient and, usually, their modularity features are limited. Furthermore, the research community has not succeeded in finding a standard process modelling language, which has clearly impaired the development of the area. In this article we propose some objectives to be accomplished concerning the above-mentioned aspects and we provide an overview of some different approaches to these issues that have been already developed. We compare them and we stress both their strong points and their limitations. As part of this analysis, we outline the PROMENADE approach to software process modelling, aimed at solving these limitations while keeping the strong points by means of the use of a complete set of control-flow and modularity constructs. To enhance standardisation, PROMENADE constructs are defined in terms of UML; UML constructs support also specialisation and flexibility of process models.

Keywords:
Software Process Modelling, Process Modelling Languages, UML

1. Introduction
A model for a software development process [DWK97] (i.e., a software process model) is a description of this process expressed in some process modelling language (PML). The process can be viewed as the execution in a suitable order of a set of tasks (e.g., requirements elicitation or module testing) intended to develop some documents (e.g., specification or test plan). These tasks are developed by some agents (e.g., people or hardware media) with the help of some tools (e.g., editors or debuggers) and using some resources (e.g., data bases or computer networks). Hence, the definition of a software process model must state all the elements just mentioned, and also the way in which this model must be executed (enacted). The systematic description of software processes not only helps in understanding software development, but also makes feasible the construction of systems for supporting automation of the process up to an acceptable level, centered on the PML.

An important research effort has been made to define well-suited PMLs (see [FKN94], [DWK97] for a survey). As a result, some important features have been attained: object-

* This work is partially supported by the spanish research program CICYT under contract TIC97-1158.
orientation has been introduced as a natural way to model the structural part of the process, while achieving a certain degree of reuse and modularity; several different paradigms have emerged leading to a wide variety of different approaches, which have shown their usefulness (from process programming languages to graphical notations, from prescriptive to reactive control paradigms, from document-oriented to activity-oriented systems...); the abstraction level of systems has increased, which has made them easier to use, and so on.

However, there are several aspects involved in the act of modelling a software process using PMLs that seem to need a more detailed study and which remain as challenges for the software engineering research community.

We feel that some of these aspects are the following: most of PMLs are difficult to use for describing fine-grained processes while keeping a high-level notation; although some of them are widespread in the community, none of the existing PMLs have emerged as a standard language for modelling software processes; most of process models seem to be too strict in order to deal with some deviations and decisions taken at enactment time. Finally, we believe that the modularity and reuse abilities provided by current PMLs are not powerful enough to deal with the modelling of complex software processes.

This paper studies these limitations in more detail and presents some of the most representative attempts to deal with them. Although we focus mainly on BPM, this paper also contains some relevant issues (addressed to these issues) achieved in the related field of workflow management. For each one of these aspects we present our own approach, PROMENADE (Process-Oriented Modelling and Enactment of software Developments).

PROMENADE is a PML for modelling software processes designed with the aim of improving the above-mentioned issues (standardisation, expressiveness, flexibility and modularity).

In order to model the structural part of a process, PROMENADE extends the UML metamodel with some process-specific metalevels. Concerning its behavioural part, PROMENADE: 1) allows the composition of partial models to construct new complex models in a modular way; 2) supports hierarchies of activity refinements which allows the selection of a particular way of performing an activity at enactment time (hence, improving the flexibility of the process enactment); and 3) defines a twofold control-flow with reactive control (based on triggers) and proactive control (based on precedence relationship between activities).

A more detailed description of PROMENADE can be found in [FR99a, FR99b, FR99c, RF00].

2. A general classification of PMLs

A comparative study of existing PMLs arranges them into three groups depending on which is the central element of their modelling. Following this idea, we can find document-oriented, goal-oriented and activity-oriented approaches (see table 1).

In document-oriented approaches, processes are usually modelled in terms of the states of the documents that take part in the process. Activities often play a secondary role as operations associated to documents. The enactment of an activity leads to the change in the state of some document(s) involved in that activity (e.g. the state of a document that is generated by the activity may change from not-yet-completed to completed). The roles involved in the process are assigned a workspace which shows the activities (associated to documents) that may be performed at a given instant.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Characteristics</th>
<th>Advantages</th>
<th>Drawbacks</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document-oriented</td>
<td>processes modelled in terms of the states of the involved documents. Activities are basic ops. associated to documents.</td>
<td>Full life-cycle approach. High level of completeness. Workspaces assigned to roles naturally.</td>
<td>Too basic activities. Process is not explicitly represented. Difficult to model complex processes.</td>
<td>MERLIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[PSW92, RS97]</td>
</tr>
<tr>
<td>Goal-oriented</td>
<td>The model describes what has to be done. An ordering of the activities to reach that goal is decided.</td>
<td>More declarative and abstract approach. No detailed description of the whole process available. Model evolution easier.</td>
<td>No detailed description of the whole process available. Model evolution more difficult.</td>
<td>EPOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[CJM95, Cor05]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Peace</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[AL06]</td>
</tr>
<tr>
<td>Activity-oriented</td>
<td>The model describes how the process is to be developed. Leads to precise process models. Allows a detailed description of the whole activity. Easier enactment.</td>
<td>Models are too prescriptive and too static.</td>
<td>Models evolution is more difficult.</td>
<td>APEL[DEA98]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JIL [W96]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EJ [J98]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SPADE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[BF94]</td>
</tr>
</tbody>
</table>

Table 1: A comparison between different modelling strategies.

Both activity-oriented and goal-oriented approaches provide a dynamic ordering of process activities [AO97] but, while in the case of the activity-oriented approach that ordering is established at enactment time (they model how the process will be enacted, the name of the process), goal-oriented approach model only the objectives that are to be fulfilled by the process (the what opposed to the how). The enactment engine is responsible for deciding an activity ordering aiming at achieving the proposed objective. [AO97] shows the advantages and drawbacks of both approaches.

In practice, most approaches for process modelling are activity-oriented which ensures something very important for human beings: to have an explicit description of the process being enacted. In addition model enactment becomes easier. These approaches have intended to overcome the drawbacks of the activity-oriented strategy providing means to deal with process evolution and describing processes in a more flexible way. Unfortunately, these solutions have not been completely satisfactory, as we will show in the remainder of the paper.

3. Expressiveness and comprehensiveness of process description

The need for an expressive and comprehensible modelling of process control-flow, together with the ability to define new control-flow dependencies, has been recognized in the last few years in the field of workflow management and some research has been carried out (see, for instance, [J996, RD98, J999]), but, in our opinion, it has been somewhat neglected in the related area of BPM.

In spite of the very nature of software processes (which involve a loose control and a high degree of collaboration) most of existing approaches lead to very prescriptive models with few basic control-flow constructs which do not conform to the way in which software is actually developed (some complaints about this are shown in [Kn98]). This leads to a great difficulty in the modelling of detailed and complex processes. In the following sections we outline some limitations of process-centered software engineering environments (PSEE) regarding expressiveness and comprehensiveness of process description.

3.1 Initial Proposals

The first few serious attempts to formalize and enact the process of software construction were developed in the early-nineties (see [FKN91]).
Among those initial process-centered environments, document- and goal-oriented approaches suffered from the problems outlined in the previous section. For instance, in the case of MERLIN, although it exhibits a high degree of cooperation and concurrence between participants, its activities are very simple and cannot be decomposed. Furthermore, the process is not explicitly represented. With respect to goal-oriented initial approaches, process description and comprehensibility of the process control-flow were not good enough.

SPADE [BF94], ADELE [FG94] and APPLA [SH05] are some examples of initial process-centered environments which followed an activity-oriented strategy. All of them use a formal, enacting and low-level underlying formalism: SPADE uses Petri-nets, ADELE is a reactive approach based on ECA-rules (event-condition-action rules) and APPLA defines a process-oriented extension of Ada (with relations, triggers and consistency management statements) to create its own process programming language.

A recurrent criticism that has been addressed to these preliminary approaches is that models built with these kinds of low-level formalism are not very intuitive and comprehensible for humans. Sometimes, this criticism has been accompanied by empirical evidence [AB97]. The underlying idea is that the modeling of a process by means of formal and usually textual languages (i.e., not graphical) is not the best way for humans to gain an understanding of the whole process.

3.2. Second generation of PMLs

Ten years after his seminal paper "Software Processes are Software too" [Ost87], L. Osterweil identified several problems in the existing approaches to SPM and proposed a list of goals to be reached by new process languages (ease of use, semantic richness, composability, clarity through visualization, multiple paradigms...) [OS97]. He called second generation process languages to the ones that met (most of) these requirements.

One way in which these requirements have been addressed has been the provision of a high-level and intuitive (usually graphical) language for modeling the process and a mapping from that language into a formally defined one that allows reasoning about the process and also enactability.

An example of transition from first to second generation is APEL [DEA98], a heir of ADELE, which also uses ECA-rules as underlying formalism. It provides a graphical language that allows a higher-level process definition. A model written in this language is translated into a lower-level one based on ECA-rules. Unfortunately, the APEL control-flow is very basic. It is limited to the usual end-start transitions which impairs the achievement of a high degree of expressiveness in modeling processes. For example, in the context of component-based software development, the following modeling situation does not seem very realistic: the implementation of a component C will start once its behavior has been defined. It seems better to overlap to a certain extent both activities: Implementation of a component C should begin some time after its behavior has been defined, and should finish after the end of this task. But this requires some control-flow constructs other than the end-start transition.

3.3 Workflow management approaches

Approaches to model processes that supply more powerful control-flow constructs do not come from the field of SPM but from that of workflow management. For instance, [JB96] recognizes the lack of expressiveness of traditional sequence, parallel and branching control-flow constructs and not only proposes more powerful control-flow constructs (which are given a formal semantics by mapping them into Petri-nets), but also they suggest that modeling formalisms should allow the definition of new control-flow constructs.

The kinds of control-flow constructs suggested by [JB96] are transition-oriented (that is, their objective is to describe what activity must be executed next rather than what requirements are necessary in order to execute an activity). We believe that transition-oriented control-flow constructs lead to more prescriptive and less expressive process models (see 3.5).

[HI99] takes a similar approach. They recognize the necessity of a more flexible and higher-level modeling, including means to define new control-flow dependencies. They supply a high-level graphical language to model processes, which is mapped into ECA-rules (which have a formal semantics). However, no set of basic, useful dependencies is given. On the other hand, the way of creating new dependencies is very low-level (they must be described in terms of ECA-rules and activity states) and it seems to suffer from some difficulties in order to generate several kinds of precedences (e.g., start or weak precedences. See 3.5).

3.4 Requirements on expressiveness

As a consequence of the research results that have been attained, we can consider that a PML should provide the following features regarding expressiveness [SO97, JB99, FR99, JB96]:

- Support for modeling heterogeneous processes of different granularity.
- Definition of built-in expressive and high-level control-flow constructs.
- Support for both proactive and reactive control in process modeling.
- Definition of new control-flow constructs in a high-level manner.
- Decomposition of complex activities into simpler ones (which, in their turn, could be either composite or atomic).
- The resulting process model should be comprehensible. The construction of the model should be intuitive. Some graphical notation may be helpful.
- Object-oriented control-flow constructs.

From the study we have carried out in previous subsections, we conclude that it seems not to exist any PML that meets all the above-mentioned requirements.

3.5 PROMENADE

The contribution intended by PROMENADE to this matter is to model a SP by stating in a declarative way the various kinds of precedence relationships existing between the different activities taking part in that process. A basic set of such precedence relationships has been identified for this purpose (start, end, strong, feedback. See table 2). PROMENADE also provides a high-level notation to define new precedence relationships derived from the basic ones (weak, successfulEnd-end and grouping table 2 are examples of derived precedences).

Figure 1 shows an example with the definition of the weak precedence in terms of start and end). PROMENADE also defines a formalism to describe dynamic precedences (i.e., precedences that are fully known only at enactment time. See figure 2 for an example). Composite tasks may have an associated precedence diagram which describes the behaviour of the task in terms of the precedence relationships that exist between its subtasks. Such

1 Although Petri-nets provide a graphical language it is still quite low-level and unnatural.
subtasks may be both atomic and composite. A precedence diagram is shown in figure 3 of section 6.

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>s =&gt; t,1</td>
<td>t may start only if s has started previously</td>
</tr>
<tr>
<td>end</td>
<td>s =&gt; t,1</td>
<td>t may finish only if s has finished previously</td>
</tr>
<tr>
<td>strong</td>
<td>s =&gt; t,1</td>
<td>t may start only if s has finished successfully previously</td>
</tr>
<tr>
<td>feedback</td>
<td>s =&gt; t,1</td>
<td>t may be reexecuted after the unsuccessful end of s</td>
</tr>
<tr>
<td>weak</td>
<td>s =&gt; t,1</td>
<td>t may start only if s has finished previously and t may finish only if s has finished previously</td>
</tr>
<tr>
<td>Successful end</td>
<td></td>
<td>Each task of ( t_1, t_2, \ldots, t_n ) must be successfully finished in order to finish any of ( t_1, t_2, \ldots, t_n ).</td>
</tr>
<tr>
<td>Grouping</td>
<td></td>
<td>Tasks ( t_1, t_2, \ldots, t_n ) must be executed indivisibly.</td>
</tr>
</tbody>
</table>

**Table 2:** Some precedence relationships in PROMENADE

**Figure 1:** An example of a high-level definition of a derived precedence relationship

\[
\text{ImplementsComponent} \rightarrow \text{strong} \rightarrow \text{TestOperation}
\]

**Figure 2:** A dynamic precedence relationship in PROMENADE

In our opinion PROMENADE provides a more expressive and comprehensible approach than other presented systems. In first place, the use of precedence relationships instead of transitions provides a more declarative, high-level and less prescriptive approach. Also, it allows the definition of dynamic precedences (which we have not seen in other PMLs). Furthermore, although [JH99] also allow the definition of new precedences, in the case of PROMENADE, this definition is performed using a high level notation more than with low-level ECA-rules.

4. Flexibility and evolution of software process descriptions

A software process is prone to change due to the evolutionary nature of both software and software processes. There are many causes that may lead to a SPM evolution. As it is stated in [NC96] new and better methods and paradigms to develop software may arise; a SPM may be incorrect or should be optimized; delays in software development may be produced...

Changes may be introduced at three different levels of a process model: at the template model level, at the enactment model level and at the executing model level. Usually, changes in the first two levels are considered to be static, whereas those performed at the executing model level are dynamic, since they are carried out during model enactment (sometimes they are called changes on the fly). Usual changes in models may involve inserting/deleting tasks or other model elements (at the type or the instance level), inserting/deleting control or data flow elements into a task description (e.g. a feedback relationship).

**Table 3:** A comparison of different systems concerning expressiveness

<table>
<thead>
<tr>
<th>System</th>
<th>Kind of approach of underlying formalism</th>
<th>Kind of control-flow constructs</th>
<th>Support to define new control-flow constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARM</td>
<td>Activity oriented (low level)</td>
<td>Proactive control (activities)</td>
<td>No</td>
</tr>
<tr>
<td>APPLA</td>
<td>Activity oriented (low level) Process program Ada extended with triggers.</td>
<td>Proactive control (Ada controls constructs) Reactive control (triggers)</td>
<td>No</td>
</tr>
<tr>
<td>MERLIN</td>
<td>Document and rule oriented PROLOG</td>
<td>Reactive control (activities)</td>
<td>No</td>
</tr>
<tr>
<td>PACE</td>
<td>Goal oriented</td>
<td>Control-flow decided at enactment time</td>
<td>No</td>
</tr>
<tr>
<td>EPOS</td>
<td>Goal oriented</td>
<td>Control-flow decided at enactment time</td>
<td>No</td>
</tr>
<tr>
<td>APEL</td>
<td>Activity oriented (high level) ECA-rules</td>
<td>End-start transitions Proactive and reactive controls</td>
<td>No</td>
</tr>
<tr>
<td>JIL</td>
<td>Activity oriented (high level) Translated into a programming language (Julia-Ada)</td>
<td>Proactive and reactive control 4 control-flow constructs and 4 event categories</td>
<td>No</td>
</tr>
<tr>
<td>E3</td>
<td>Activity oriented (high level)</td>
<td>Proactive control with precedence relationships 2 precedences defined: preorder and feedback</td>
<td>2</td>
</tr>
<tr>
<td>J/H99</td>
<td>Activity oriented (high level) ECA-rules</td>
<td>No predefined set of control-flow constructs defined; although many may be defined</td>
<td>Yes Low-level definition of new constructs using ECA-rules</td>
</tr>
<tr>
<td>R/OD98</td>
<td>Activity-oriented (high level) Graph-rewriting</td>
<td>A set of control-flow constructs defined Proactive control</td>
<td>No</td>
</tr>
<tr>
<td>PROMENADE</td>
<td>activity oriented (high level) ECA-rules</td>
<td>Different kinds of basic, derived and dynamic precedence relationships. New precedences may be defined. Proactive and reactive control</td>
<td>Yes High-level language to define derived precedences in terms of the basic ones.</td>
</tr>
</tbody>
</table>

It is important to note that changes may involve inconsistencies (a task is introduced which leads to a deadlock situation or that is not reachable from the initial task of the control flow, a task instance which was being executed has changed...). Hence, some mechanisms should be provided to enforce consistency after a modification.

Finally, another related issue is that of flexibility. A software process should not be too much prescriptive. A software process is complex and a complex process should not be completely detailed before enactment. There should be means to refine it during enactment. This may involve changes to the model as we have stated or different choices (or realizations) when an activity is to be executed.

In order to cope with evolution and flexibility in a suitable way the following features should be provided:

- A metamodel for the PML should be given. Its metalevels should supply the structure of model elements and supply operations to modify them.
- The PML should be able to generate a model, both for the production process (the model for the production process is the SPM) and for the metaprocess (this is the process for changing a SPM). Notice that, while the production process is the result of the SPM
enactment, the metaprocess will use it as data to be modified. Therefore, the PML should be reflective.

- The metaprocess should be explicit.
- The PML should support changes in the three above-mentioned levels: template, enactable and enacting.
- The PML should supply tools for checking the resulting model for consistency and correctness. Moreover, it should only allow changes that keep consistency and correctness.

Within the family of activity-oriented systems, EPOS and SPADE were pioneers in the incorporation of evolution support for process models.

In order to achieve model evolution, EPOS relies on reflection and on the definition of a hierarchy of types which contain both metatypes and normal types. It allows changes-on-the-fly but it is not reported how they are checked. On the other hand, EPOS defines an explicit metaprocess to create and evolve SPM. Unlike EPOS, more recent approaches (e.g. UML) separate clearly the metamodel and the model levels.

SPADE also relies on the existence of a metamodel and reflective features in order to provide model evolution. SPADE does not define a metaprocess and does not report how to make changes into the process that deviate from the model.

Changes in all these systems are seen as something necessary to be taken into account but not necessarily an everyday operation. In some other systems changes and flexibility in model enactment are considered to be a major feature. This is the case of Peace+, in the field of SPM and of DYNAMITE and ADEPT-F, in the field of workflow management.

Evolution process in Peace+ [AL996] is based on reflection (both in the process and the evolution process are expressed in the same formalism and their execution is supported by the same engine). Constraints are represented as and checked by means of consistency graphs which simulate the consequences of a change in the process regarding consistency. Two kinds of inconsistencies have been considered: strong (they cannot be tolerated) and weak (they can be tolerated temporarily).

The high degree of interconnection between task enactment and task modification is one of the main advantages of DYNAMITE [HK996]. Therefore, model evolution is performed continuously and incrementally during model enactment. A process model in DYNAMITE is very basic at the beginning. It will be refined progressively by the dynamic incorporation of development and data flows (in particular feedback relationships) and also with new activities. This evolution was performed initially by means of graph rewriting rules (using a formal graph rewriting specification language called PROGRES). In a latter version DYNAMITE uses ECA-rules to perform change operations. These ECA-rules are responsible for checking the validity and possible inconsistencies of the changes they perform. Some consistency and correctness properties are not checked (e.g. deadlocks). In our opinion, the absence of a well-defined process model constitutes a drawback of this approach. For human beings it is important to have a (may not be fully detailed) description of the process to be followed.

The idea of graph rewriting is also used in ADEPT-F [RD98] approach for workflow management. It defines some correctness and consistency properties which are taken into account to ensure model correctness after dynamic changes.

PROMENADE

PROMENADE is a reflective PML. This is based on the facts that both the model and the metamodel are described in the same formalism (UML). See section 6) and that a metalevel (SPMetamodel) which instances are SPMs has been supplied to its definition. Therefore, PROMENADE is designed to support a metaprocess definition and also to allow process evolution. However, neither aspect has been implemented in the language yet.

On the other hand, PROMENADE is also designed to provide a high degree of flexibility in model enactment by means of task refinements. Intuitively, a task refinement is a concrete way to perform a task. A bit more formally, a task refinement of a composite task class T is a task class that expresses one specific way in which T may be decomposed into subtasks and precedence relationships that should be kept among them at enactment time. Since, in general, it is possible to think of several ways to perform a task, it makes sense to define several task refinements for a specific composite task. Any of these refinements could be selected at enactment time, or even a new one could be defined.

PROMENADE includes a list of task refinements that are the same that the ones in DYNAMITE and ADEPT-F, and it also includes the redefinition of some of them. PROMENADE also adds some new task refinements that are not present in the other languages.

<table>
<thead>
<tr>
<th>System</th>
<th>Explicit metamodel</th>
<th>Reflective PML</th>
<th>Flexibility in enactment</th>
<th>Levels of change</th>
<th>Consistency checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPOS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SPADE</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MENDEL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Peace+</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DYNAMITE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ADEPT-F</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PROMENADE</td>
<td>Two levels of metamodel (metamodel and reference model)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4: A comparison of different systems concerning flexibility and evolution.

5. Modularity and reusability

If software processes are software too, it makes sense to apply the notion of reuse to SPM. In fact, one of the most challenging issues in SPM in the last few years has been the ability to incrementally (modularly) construct a model by combining existing ones. This leads to a bottom-up approach that so far has not been completely attained by existing systems.

This column explores whether the system checks the consistency of dynamic (on-the-fly) changes.
Several levels of reuse in SPM can be considered: (1) Reuse of primary model components, (2) derivation of model components and (3) composition-combination of models. Clearly, the last is the most challenging one.

Model reuse raises several problems like name consistency [EHT97] and causal consistency [EHT97] (the constraints stated by a model or a model element may not be fulfilled anymore when reused elements are incorporated). Finally, another problem is how to find a component to be reused from a specification.

The features that a PML should accomplish regarding model and model element reuse are the following:

- It should allow different levels of reuse (from primary component reuse to model reuse).
- It should also allow model and component derivation.
- It should provide means for consistency checking of the model which has reused elements.
- It should provide a well defined component and model specification to ensure that a fragment to be reused is what is needed.
- It should provide intuitive and high-level operators in order to combine existing models. This feature would lead to a powerful way to construct models incrementally by the meaningful combination of existing ones.

5.1 Reuse of primary model components

The first step to construct SPMs modularly is clearly the reuse of primary model components such as documents, activities or roles in several SPMs. As we have stated, this idea can be extended with the notion of derivation. That is, the creation of new components (even entire models) by the modification of already existing ones on which they will be based. This derivation may be performed by means of inheritance, redefinition of some component constituents and definition of new ones. Reuse and derivation of model components requires consistency checking of the new created model.

This first step of reuse may be achieved in a straightforward way by using an object-oriented PML.

For the sake of an example we mention E3 [JPL98, Jacc96, JS00], a fully object-oriented PML which aims at providing some degree of reuse. It takes advantage of the object-oriented features to provide reuse of primary process components, template models, and also the construction of new model components by the derivation of existing ones. What these systems do not provide is the ability to reuse and combine existing models to construct modularly new ones.

A challenge of component reuse is the proper specification of reusable components in order to be sure that the reused components conform to the model that is being constructed. [Kal96] provides a formal approach to solve this situation based on the existence of a common redex of two specifications.

5.2 View-oriented approaches

To our knowledge there are not many approaches that aim at constructing models modularly by defining some methodology to reuse and compose (integrate) submodels. With the exception of [Cht94], which gives some guidelines for defining operators to combine models, all the approaches that follow this direction we are aware of rely on the notion of view. In general, a view may be considered as a projection of a process model according to a well-defined characteristic [AC96]. Usually, the characteristics based on which, the view is constructed are roles, activities, products...

Views suggest a top-down approach to SPM; i.e., they seem a good way to decompose a model into simpler ones in order, for example, to restrict the part of the process available to a given role or to show a more comprehensible model by focusing on one of its aspects. However, since views must be derived from an existing model, we believe that they are not the most natural way to perform model composition from submodels, which is clearly a bottom-up approach.

Among the view-based approaches, we think that OPISI [AC96] is one of the most competitive ones. It is a system based on views which may be generated by different views which may be used to model processes written in a Petri-net-based formalism. OPISI uses views for several purposes: on the one hand, it aims at managing the complexity of a model by partitioning it in several views; on the other hand, authors claim that model evolution may be made easier if it is expressed in terms of views instead of in terms of the whole model. Finally, views are also used in OPISI in order to construct models by adapting and composing existing views. Composition is performed by means of a composition operator which defines some functions to connect places and transitions from the two views to be composed.

In OPISI, views always come from an existing model. Therefore, composition cannot supply a pure bottom-up approach to construct complex models from simpler ones (which would be more natural in our opinion). On the other hand, view composition is limited to a superposition of existing views. More powerful and high-level operators to combine views in specific ways would be desirable. Finally, OPISI does not offer consistency checking (e.g., name collision).

Table 5 shows the results of other two view-based approaches to attain model reuse ([EHT97] and PYNODE [ABC96]).

5.3 PROMENADE

PROMENADE allows reuse both at primary component level and at model level. At the primary component level, reuse is a consequence of the fact that PROMENADE is a fully object-oriented PML. In particular, it allows component (and also model) derivation by inheritance. In order to provide reuse at the model level, PROMENADE defines three operations that may be applied to SPMs (or its elements) related with modularity: refinement, renaming and composition. Refinement operator allows model transformations; Renaming allows the application of a name substitution on a model. Finally, model composition allows the construction of a new SPM by the composition of some already constructed SPMs. These existing models may be seen as partial models or as particular views of a complete SPM, but they have the status of SPMs. The approach of PROMENADE has been chosen to deal with model composition consists roughly in building a model $m$ from the composition of a set of models $m_1, ..., m_n$ and a set of precedences $(a_i, a_j)$ in the following way:

- The static part of $m$ is the superposition of the generalisation hierarchies of $m_1, ..., m_n$, together with the union of their association and aggregation relationships.
- The dynamic part of $m$ is built by combining the preconditions of each model $m_1, ..., m_n$, with the precedences $(a_i, a_j)$. Hence, our approach to model composition is not based either on views that should be extracted previously from an existing model or on a mere superposition of submodels. Instead, PROMENADE provides an incremental and uniform way to construct complex models from simpler ones. Moreover, these simpler models are combined using precedence relationships among their main tasks, which leads to a meaningful, expressive and high level
model composition. Notice that the normal approach (from-the-scratch) to model construction is also supported by PROMENADE.

<table>
<thead>
<tr>
<th>System</th>
<th>Primary components reuse and component derivation</th>
<th>View or model composition</th>
<th>Consistency checking</th>
<th>Specification of reusable elements</th>
<th>High-level operators combine submodels to</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3</td>
<td>Yes (O.O appr.)</td>
<td>No</td>
<td>Not reported</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PYNODE</td>
<td>Yes (Component derivation not reported)</td>
<td>SPM is a role of object-oriented views (top-down appr.). Views not reusable. Views are sets of reusable components</td>
<td>Not reported</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OPSIS</td>
<td>Not reported</td>
<td>View composition (applied to Petri Nets) Very basic. No name consistency check.</td>
<td>Not reported</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>[EIT97]</td>
<td>Not reported</td>
<td>View composition. Top-down approach View reuse is not clear. Yes Name and constraint consistency check. manually</td>
<td>Not reported</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>[PTV97]</td>
<td>Tasks (workflows) may be reused</td>
<td>No</td>
<td>Transcational processes</td>
<td>Not reported</td>
<td>No</td>
</tr>
<tr>
<td>PROMENADE</td>
<td>Yes (O.O approx.)</td>
<td>Composition of reused models Yes Name and constraint consistency check. No (future work)</td>
<td>Yes Models combined by precedences between main tasks</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5: A comparison of different systems concerning modularity

6. Standardisation

The proliferation of languages and notations that we have just outlined in this article has hampered the wide use of software process technology within the software engineering community. One could wonder if it is time for the community to adopt a language and, in this case, which should be the chosen formalism. UML seems to be a natural candidate for such a standard process modelling formalism since it has become a standard de facto in the modelling of O.O. systems.

Some recent approaches have come up that use UML as a modelling formalism in the related fields of software and workflow processes ([AL98, ML98, UML98-2, FR98, JSW99]). A preliminary result of this research is that, although UML seems to be powerful enough to address those aspects concerning the static part of a process, it lacks some degree of expressiveness and flexibility in order to model its behavioural part. As we have seen, the control flow of software processes and workflow processes is modelled usually in an activity-oriented way, mainly by some sort of activity diagram that represents both the involved activities and the transitions from activity to activity. The UML diagram that conforms with this view of the process is activity diagram. But it can only show end-start transitions between activities which, as we have seen in section 3 is clearly not expressiveness enough to deal with complex processes.

Some work has been done aimed at using UML as a process language: Rational Software Corporation et al. have developed a UML extension for objectory process for software engineering [Rata]. Essentially, it extends some metamodel classes by means of stereotypes.

Neither structure nor behaviour is given to those stereotyped classes; no integrity constraints are defined; and no means to improve the UML features in order to deal with the dynamic process are provided. Therefore, this proposal seems to be insufficient to meet the requirements of SPM.

[JSW99] presents an approach to SPM based on UML which describes the behavioural part of the model using class diagrams with stereotyped associations for showing the control and data flow. It concludes that, although it has some limitations, UML is an adequate modelling language for software processes. We believe that this is not the most natural choice since UML associations are used to indicate that some specific instances of one classifier are structurally related to some specific instances of another. However, the information that both precedences and transitions convey is not structural and the attributes that the UML metamodel defines for associations are not applicable at all to them. Also, we have already mentioned the drawbacks of a stereotype-based solution.

6. Standardisation

7. Conclusions

This paper shows the state of the art in the field of SPM concerning four main issues: expressiveness; model evolution and flexibility in model enactment; modularity in model construction and standardisation. We have identified some challenges in these aspects and we have presented our approach PROMENADE and its proposals in these directions.
Figure 3: A precedence diagram in PROMENADE

The tables that appear in the different sections comparing several approaches show that, in most cases, PROMENADE overcomes some of the referred limitations. PROMENADE has been used to model the ISPW-4 software process and also a very detailed process aimed at constructing a library of software components.

Some work has been done yet in PROMENADE in order to cope with model evolution (specifically to allow changes on-the-fly) and to provide model specifications that may help in reuse. These are the issues on which we are currently working.

References


