Techniques for Process Model Evolution in EPOS

Maria Letizia Jaccheri
Dipartimento di Automatica e Informatica
Politecnico di Torino
10129 Torino
Italy

Reidar Conradi
Dept. of Computer Systems and Telematics
Norwegian Institute of Technology (NTH),
N-7034 Trondheim,
Norway

5 May 1993

Abstract

This paper categorizes some aspects of software process evolution and customization, and describes how they are handled in the EPOS PM system. Comparisons are made with other PM systems.

A process model in EPOS consists of a Schema of classes and meta-classes, and its model entities and relationships.

There is an underlying software engineering database, EPOSDB, offering uniform versioning of all model parts and a context of nested cooperating transactions.

Then, there is a reflective object-oriented process specification language, on top of the EPOSDB. Policies for model creation, composition, change, instantiation, refinement and enaction are explicitly represented and are used by a set of PM automatic tools. The main tools are a Planner to instantiate tasks, an Execution Manager to enact such, and a PM Manager to define, analyze, customize and evolve the Process Schema.

Contents

1 **Introduction** 1

2 **The Process Model Life cycle** 2
- 2.1 The PM meta-process 2
- 2.2 Evolution 3

3 **EPOS** 5
- 3.1 The Layered EPOS Architecture 5
- 3.2 The SPELL Language 6
- 3.3 The Process Tools 7
  - 3.3.1 The Execution Manager 7
  - 3.3.2 The Planner 8
  - 3.3.3 EPOSDB: The Change Oriented Versioning Model 8
  - 3.3.4 EPOS Project Context 9
- 3.4 Methodologies of Change in EPOS 9
- 3.5 An example: the ISPW7 Reference Problem 13

4 **Comparisons and Related Work** 14

5 **Conclusions** 16

1 Introduction

πάντα ῥεῖ

Heraclitus of Ephesus[1]

The Software Process and its support has attracted increasing attention in the last 15 years. A *Software Process* is the total set of software engineering activities needed to transform user requirements into operative software and to evolve it. It is composed of two main components: a *software production process* to carry out software production activities, and a *software meta-process* to improve and evolve the whole software process.

A *Process Model* is a description of one or more software processes, and it is composed by a production process model and a meta-process model (meta-model). A part of the model is called a *model fragment*. Software[2] *Process Modeling* (PM) is the discipline of describing process models [1] [2]. [3] [4].

In this paper, the term *process model* is used to denote the *internal* computer representation of an *external* process. The term process model denotes *both* a process abstract description (as a schema) and a more concrete description of the external process elements to be supported. The external process elements constitute the real world production environment that cannot be totally represented in a computerized form, e.g. human behavior. However, several authors use the term process model only about a *process schema* (templates, classes, rules).

A *process schema* provides a *template* description of a group of *process elements*, e.g. software production activities, products (artifacts), tools, human roles, projects, organizations etc. – with interconnections. The schema may consist of related sub-schemas, e.g. one for describing activities, etc.

The *Process Support Environment* consists of a *Process Modeling Language (PML)*, possibly a library of schemas expressed in the PML, and various *process tools* to support definition, instantiation, evolution, and enactment of process models. It is similarly divided in production process support and meta-process support. If the underlying PML is *reflective*, the schema defines both the production process model and the meta-process model.

Software processes are typically life-cycle *activities* such as requirement analysis, design, coding, testing, installation, maintenance etc.. Few activities are

[1] All things are in a state of flux.

[2] The "software" prefix may often be omitted in the following.

atomic; the majority being compositions of more concrete activities. Activities may communicate, operate on input products to produce output products, and share the same products. Two or more activities may be carried out by the same human role or use the same tool.

Software products consist of all the product artifacts (usually documents) produced during the software life cycle, such as requirements and design specifications, source codes, released programs, libraries, test packages, bug reports, and documentation. Each artifact may exist in many versions.

A *tool* is an executable software program, often consisting of a set of cooperating sub-tools in a tool set. Tools are invoked by activities and communicate with each other. Typical *production tools* are those for requirements specification, tracing, prototyping, reuse, modeling, program generation, compilation, maintenance support, and documentation generation.

The *user* applies the production tools, assisted or enforced by the process support. Different kinds of users are programmers, designers, quality engineers, project managers etc.. A *project* is the work context where the software processes occur and encompasses users, tools, and products, plus the process model that is actually governing it.

A Process Support Environment (PSE) is a human-oriented system [3], intended to serve interacting computerized tools and humans. Ideally it should serve as an intelligent and cooperative assistant in the daily chores of the project workers. However, users tend to modify and improve the process they are carrying out. This is due to better understanding of, and creativity towards, their objectives. It is also that they may find the process faulty, ineffective, or no longer valid with respect to its requirements or its supporting technology.

A process model must therefore be continuously maintained during its life time. Software Process Model *Evolution* is the act of changing existing models in a controlled way [5] [6]. This includes Software Process *Customization*: reusing existing process model fragments and adapting them to different contexts.

The paper is structured as follows: section 2 defines a process model life cycle, and elaborates the meta-process for process evolution and customization. Section 3 presents the EPOS support for process model evolution. Section 4 discusses some related work and tries to compare EPOS features with those offered by some existing systems. Conclusions are given in section 5, with indications of further work.

2 The Process Model Life cycle

Process models are themselves produced by an engineering process. Such engineering (creating, changing etc.) consists of a set of phases, called PM *meta-activities* [3], and constitute the **meta-process**. The meta-process of producing process models clearly resembles the software process of creating normal executable software products.

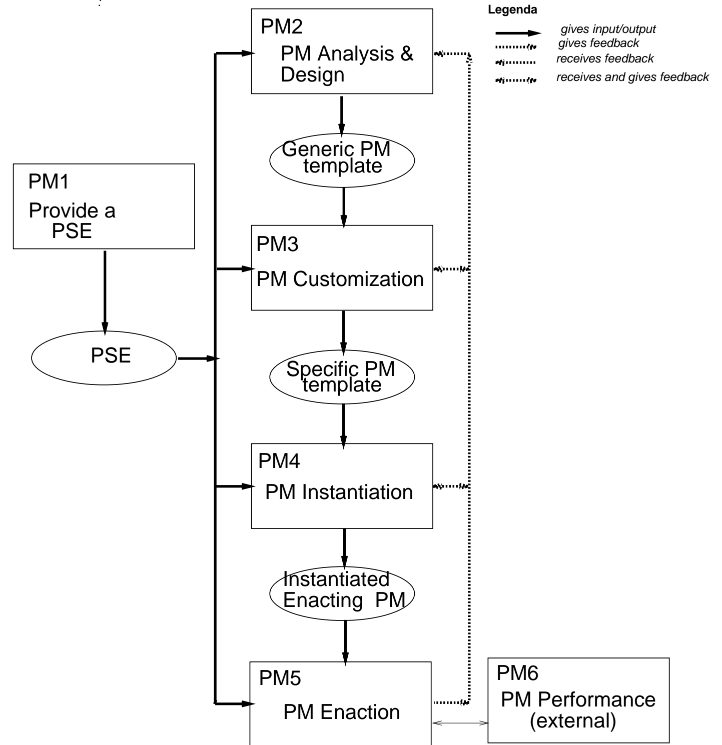

Figure 1: The general meta-process, with meta-activities.

2.1 The PM meta-process

Fig. 1 shows six meta-activities, depicted by boxes, and their respective inputs and outputs, depicted by ovals. Bold arrows denote input/output relationships; dashed arrows denote feedbacks from a meta-activity to the ones above it.

The initial meta-activity (PM1) must provide a PSE. A PSE offers an enactable PML with precise syntax and semantics, libraries of reusable process models, a PM methodology, and various process tools for process model creation, composition, refinement/ customization, instantiation, enaction, and evolution.

The second meta-activity (PM2) is the Analysis and Design phase of a *generic*, template process model (schema). Such a generic schema is an abstract process model, such as the waterfall or spiral model, for use in many projects.

The third meta-activity (PM3) or customization step reuses the generic schema to obtain a more specific schema to accommodate project- or application-related information by adaptation and refinement.

The forth instantiation meta-activity (PM4) produces an instantiated software process model, with concrete descriptions of activities, connected to input/output products and with attached roles (actors) and tools.

This model is gradually made enacting by the fifth enactment meta-activity (PM5), which also executes and monitors it.

Finally comes the sixth meta-activity, being continuous assessment of external process performance (PM6). This goes in parallel with PM5 on enactment.

There is no assumption that the above meta-activities must be executed in a strict water-fall fashion for all components of the process model. Further, not every PSE allows the distinction or formalization of all these meta-activities.

2.2 Evolution

Process models must be created so that they can be customized to different project contexts. This means that process models contain a certain number of *parameters* to facilitate reuse through customization. However, customization before instantiation is not always sufficient. In fact, during and after enaction, the external software process is assessed for correctness and performance. This evaluation produces feedbacks to the earlier meta-activities. This may result in changes either to the instantiated or template process models (generic/specific), or even to the PSE. These changes are driven by feedbacks produced at the enaction level, and were not anticipated by the model designer. They may thus be regarded as process model maintenance, performed by the overall meta-process.

Solving the problem of process model evolution requires an answer to the following questions: which process model fragments should be changed, how and when? And how to analyze and guide change?

Fig. 2 answers the question "Which model fragments to change?". It depicts the different categories of process fragments. At the lowest level, there are the instantiated/enacting model fragments. Lines denote data flows between product fragments (circles) and activity model fragments (rectangles). The next level shows the generic or specific schema. This consists of Sub-Schemas with relationships and constraints. At the top level, there is the *meta-model* (including the Meta-Schema), i.e. the encoded rules and procedures for process model definition and manipulation.

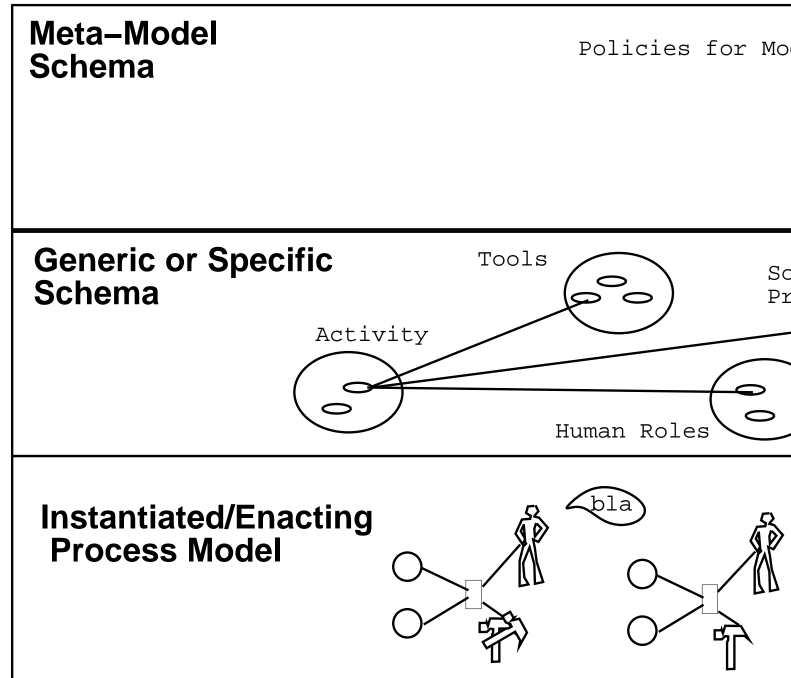

Figure 2: Process model fragments: candidates for change.

Each model fragment in fig. 2 may prove to be inadequate and need to be changed.

Instantiated/Enacting process model
Starting from the bottom level of fig. 2, the product model fragments must always be changeable, since evolution of products is the main aim of the external software production environment.

Additions or changes to activities are more difficult, as they may impact existing work. Changes to tools or human work allocation must also be considered.

Such changes will result in either feedback and respective changes to the process schemas, or in temporary changes (patches) to the instantiated/enacting model.

Generic or specific schema Changes to the generic or specific process schemas consist in changes to descriptions of single items, or the

constraints on their interactions. As items at this level describe items at the instantiated/enacting process model level, a change to one of this item may impact not only items at the same level but also items at the lower level (Instantiated/Enacting process model).

Meta-model schema The meta-model may be found inadequate due to feedbacks from the lower levels. These changes are very delicate as they impact the way in which items are manipulated both at the lower levels and at the meta-model level itself, e.g. how to change a procedure regulating meta-model changes?

Traditionally, Configuration Management (CM) needs PM to control activities related to change control, change propagation, consistency maintenance, auditing, re-building etc.. On the other hand, the entire process model constitutes a versioned and composite object, thus it should itself be under CM control. However, there are some additional problems in evolving enacting process models.

Fig. 3 gives a CM perspective of PM change. Here, the terms revision and variant (branch) are given the classical CM semantics [7]. A process model may therefore be modified as sequential revisions, or as alternative/parallel variants that evolve independently. Revision and variant are commonly termed *version*.

On the horizontal dimension, PM.1.1 is created as a revision of PM.1.0. On the vertical dimension, PM.2.0 and PM.3.0 are obtained by alternative refinements of PM.1.0.

The technology to facilitate change of model fragments varies between available PSEs, and also between different categories of fragments. The underlying PML is decisive here.

A *reflective* PML and PSE architecture will generally be advantageous to handle model changes. All process-relevant information can then be explicitly and uniformly manipulable (as in Lisp), and the meta-model can be explicitly represented, reasoned upon, enacted and evolved in a controlled way. Proper access control is of course needed here, as for general database operations.

We can define the following skeleton *meta-process* for process model changes: 1) submit a request for model change; 2) assess (validate, simulate etc.) the request; 3) reject or accept a possibly adjusted change request; 4) carry out the accepted change; 5) propagate it to a subset of the affected internal fragments and possibly to their external process elements; 6) re-establish internal and external consistency.

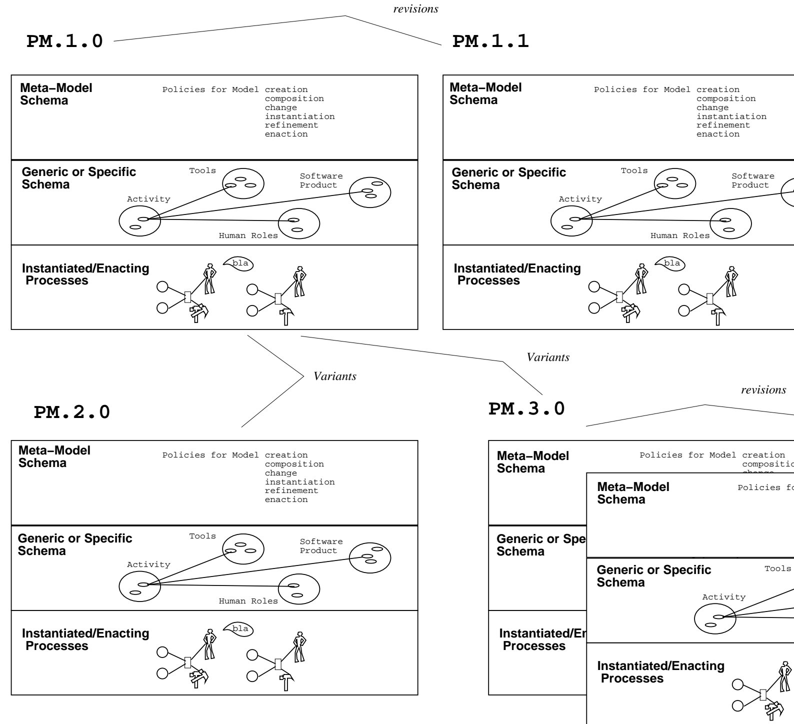

Figure 3: Changing Process Models.

Such a meta-process should encode aspects of a change methodology to guide process model evolution. The overall methodology can be rather independent of the actual PML and its process tools.

Change propagation may be *eager* (changes are propagated immediately), *opportunistic* (changes are propagated at some later convenient time), *lazy* (each fragment is checked for consistency upon later access). To facilitate precise forward analysis and propagation, and similar backward traceability, we need to explicitly represent external process elements and their dependencies in an internal process model.

3 EPOS

EPOS[3] [8] is a process support environment that offers a PML called SPELL (Software Process EvoLutionary Language) [9], an initial process schema, and a set of process tools.

In EPOS, the *internal* process model is a network of activity descriptions (tasks), being linked to descriptions of other tasks, products, tools, and roles. The activities interact with each other and with tools and humans.

The process model schema is represented as a set of entity and relation classes that constitutes template fragments. The meta-model part of the process schema is represented as a set of meta-classes. The instantiated and enacting process models consist of instances representing external process elements. The EPOS model fragments are meta-classes, classes, and instances; both entity and relation.

The main process tools operating on the above models are: a *PM Manager*, an *Execution Manager* (Process Engine), an *AI Planner*, and *EPOSDB*, a versioned object-oriented database.

3.1 The Layered EPOS Architecture

PSEs that rely on an object-oriented database, e.g. PMDB [10] and ADELE [11], often have a PML as a layer around the underlying database. EPOS extends this by having three layers around the database. Thus, the EPOS layers are:

A client-server EPOSDB with *change oriented versioning* [12] [13] in a context of long, nested and cooperating transactions. EPOSDB offers a structurally object-oriented data model and its DDL to define entity and (binary) relation classes[4]. Entities (objects) have unique and immutable identity (an OID). There is a system-defined `entity` root class. Both entities and relationships can have scalar attributes with inheritance. Entities can also have longfield attributes to describe external files. This data model is close to the *object-relation* model suggested by [15]. A free-standing Prolog based DML is offered.

All entities, relationships, and their classes and meta-classes (as `class_descriptor` instances) are stored in the database and they are thus uniformly versionable.

[3] EPOS: Expert system for Program and ("Norwegian Og") System development.

[4] In the EPOS literature they are called types, but here we adhere to the OMG [14] terminology.

A reflective and object-oriented PML
SPELL unifies and extends the underlying DDL/DML, and offers class-level attributes and instance/class-level procedures. Active procedures, or triggers, may also be defined. Meta-classes are used reflectively as in Smalltalk [16] to store class-level information.

This EPOS layer supports meta-activities Analysis & Design (PM2) and Customization (PM3), and later Evolution through the PM Manager.

A tasking framework for concurrent enaction of process models. This is done by the Execution Manager that interprets special class-level attributes defined in a predefined `task_entity` class whose definition is shown in fig. 6. The Execution Manager cooperates closely with the Planner, that incrementally (re)instantiates task networks, and with external tools.

That is, this EPOS layer supports meta-activities PM5 (Enaction) and partially PM4 (Instantiation).

Application-specific process models are domain-specific and include both schemas, e.g task templates, and instances, e.g. production tools.

Instances often constitute a network of tasks and products, connected to production tools and human resources. Relationships to describe subtasking and dataflow between tasks and products are commonly used.

Fig. 4 shows how the six meta-activities from fig. 1 are implemented by the EPOS process tools.

3.2 The SPELL Language

SPELL is a persistent object-oriented language with a reflective architecture.

Classes and Model Structuring

Fig. 5 displays some system-defined and predefined classes, showing the same layering as in Section 3.1.

SPELL supports several levels of abstraction and composition to model the external process elements. It supports definition of classes with both an *instance-level* and a *class-level* part, enabling specific and general information to be naturally represented. Class level relations can be explicitly modeled and they can be used to model subclassing, or some internal relations that encode the `formals` (dataflow) and task `decomposition`. E.g., relation class

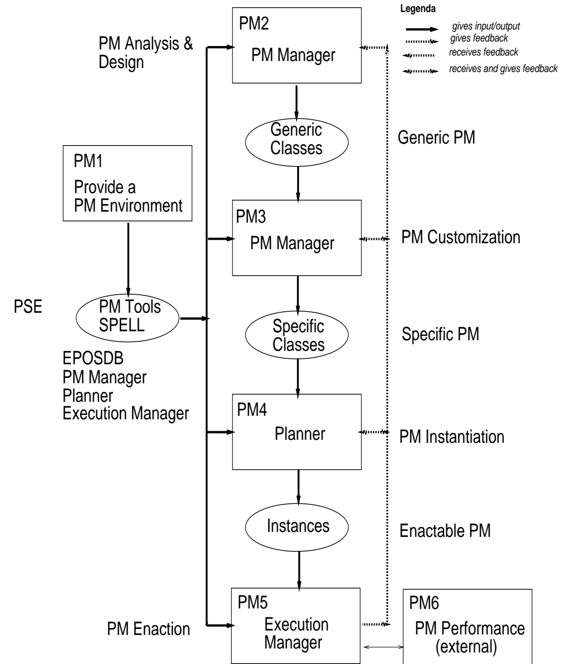

Figure 4: The actual meta-process in EPOS.

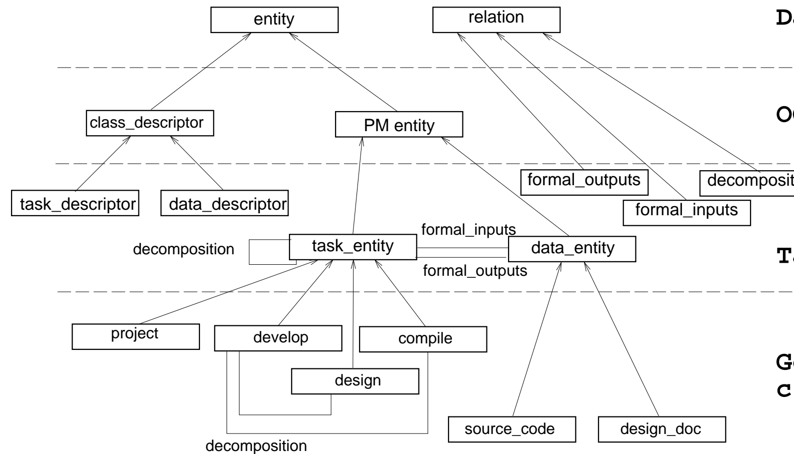

Figure 5: The EPOS Classes.

`formal_inputs` is defined between `task_descriptor` and `data_descriptor`, and it may be instantiated between `task_entity` and `data_entity` subclasses.

Inheritance and Protection

Single *inheritance* is provided for all properties (attributes, procedures). A subclass may redefine class-level attributes and instance/class-level procedures. Three kinds of inheritance are available for class-level properties and for procedures: `redefine`, `append`, and `concatenate`. `redefine` means overwriting (the default), while `append` applies only to class-level attributes and means logic conjunction (used on the predicates in fig. 6). `concatenate` is inspired by the Simula [17] *inner* mechanism. It means, that if the code of a superclass is defined as `step2`, and the code of a subclass is defined as `(step1,inner,step3)`, the concatenated subclass code is `(step1,step2,step3)`. Redefined procedures in a subclass may be inherited by either `redefine` or `concatenate`. Attributes and procedures can be declared `private` or `public`.

The SPELL Interpreter: Dynamic Binding

The access and binding of procedures, and attributes, both instance- and class-level, are dynamic as in Smalltalk. It is implemented by the SPELL Interpreter that consists of one Prolog predicate[5]:

```
call_proc(?Caller,+Called,+Procedure_Name,
          +Input_Parameters,-Output_Parameters).
```

The predefined DML procedures `read, write, read_relation`, and `write_relation`, are used as procedure parameters for accessing both instance- and class-level attributes, e.g.:

```
call_proc (?Caller, +Called, read,
           +Attribute_Name, -Attribute_Value).
```

Related entities may be accessed by:

```
call_proc (?Caller, -Called, read_relation,
           +Relation_Name, -Relation_Item).
```

The `task_entity` class

A simplified definition of this basic task class is presented in fig. 6. Generally, a task has actual input/output parameters to facilitate dataflow chaining. The classes of these "product" parameters are constrained by the `formals` class-attribute. Likewise, a `decomposition` class-attribute describes the classes of possible subtasks. Static and dynamic pre/post-conditions and a `code` script are also defined. The meaning of all these class-level attributes will be further explained in the next subsections.

Task subclasses with empty `decomposition` model low-level activations of atomic tools. Such classes serve as tool envelopes. Task subclasses with non-empty `decomposition` model high-level activities, whose main work is delegated to their generated subtasks.

[5] In the specification of Prolog procedures, parameters prefixed by ? are optionals, by + are mandatory input parameters, and by − are output parameters.

```
class task_entity: entity
{
  instance_level_attributes:
    task_state:= created.

  instance_level_procedures:
    instance_delete(+SelfO);
    instance_convert(+SelfO, +New_Class);
    start(+SelfO);
    restart(+SelfO);
    stop(+SelfO).

  class_level_attributes:
    pre_static    := true      (inheritance=append);
    pre_dynamic   := true      (inheritance=append);
    code          := inner     (inheritance=concatenate);
    post_static   := true      (inheritance=append);
    post_dynamic  := true      (inheritance=append);
    formals       := data_entity -> data_entity ;
    decomposition:= repertoire(task_entity);
    executor      := nil;
    role          := nil.

  class_level_procedures:
    instance_create(+SelfC, +Attribute_Values, -Instance_
    class_create(+SelfC, +Defined_Properties, -Class_Id);
    class_delete(+SelfC);
    class_change(+SelfC, +Defined_Properties).
}
```

Figure 6: The task_entity class.

3.3 The Process Tools

In the following, the Execution Manager, and the Planner will be described. Then, the versioning model of the EPOSDB will be introduced. Finally, the PM Manager will be described.

3.3.1 The Execution Manager

Tasking is realized by the Execution Manager that utilizes three class-level attributes:

- pre_dynamic, specifying the condition on when to enact an instance of the given task class. The condition is combined with local task information about task state and goal-directed vs. opportunistic execution.

- code, being a sequential program to perform the intended job of the given task class. Thus, enaction of a task means interpretation of its code.

  For an *atomic* task class like compile, the code contains all the relevant actions. For a *composite* or high-level task class like develop, the middle part of code is empty, causing the Planner to be invoked to instantiate subtasks.

- post_dynamic, e.g. to treat errors.

The Execution Manager also maintains the instance-level attribute task_state (line 4 in fig. 6), whose value domain is: created during Planner instantiation, waiting on its pre_dynamic condition, active during code enaction, waiting_children denoting an expanded (planned) composite task waiting for its children to terminate, forked denoting an atomic task waiting for its associated operating system task to stop, or terminated.

The attribute executor refers to the logical name of the tool that should execute the task, while role refers to the role or responsibility description of a generic human actor, e.g. a software developer.

3.3.2 The Planner

The AI Planner [18] is technically a procedure in meta-activity PM4. It is implicitly and incrementally invoked by the Execution Manager to detail a composite task at the time. That is, the Planner will automatically generate a new subtask network for each composite task. The EPOS Planner uses a domain-independent, non-linear AI planning algorithm, as in TWEAK [19] and IPEM [20]. Dynamic and incremental instantiation is achieved by this collaboration between Planner and Execution Manager.

The Planner starts with a composite task and its desired output which is the *goal*. It applies backward chaining and hierarchical decomposition, combined with domain-specific knowledge, to build a proper subtask network (a plan in AI terms). The planning is based on the process schema as a knowledge base, and a representation/model of the product structure as a world state description.

The Planner consists of two layers: the core layer is a domain independent planner, while the outer layer is domain-specific to PM. The Planner transforms the class-level attributes pre/post_static, formals, and decomposition of task class into AI pre/post-conditions of nodes, so the core layer can work. This transformation also considers the product structure.

The Planner utilizes four class-level attributes:

- pre_static and post_static express necessary conditions that must hold, respectively, before and after enaction of a task.

- formals specifies the legal "product" classes of actual parameter instances (Inputs/Outputs) of the given task class.

- decomposition specifies a pool of candidate task classes for subtasks of the given composite task.

These class-level attributes together specify legality constraints on the structure of the task network.

Clearly, changes to these schema attributes and to the product structure imply replanning. An incremental algorithm for replanning is presented in [21].

3.3.3 EPOSDB: The Change Oriented Versioning Model

In the following, we first describe transactions and then versioning.

Transactions

EPOSDB offers long and nested transactions with checked-out workspaces. Transactions may survive several application sessions, and are represented by special transaction objects in the database, connected to project tasks. The transactions may be started, committed or aborted interactively. All database operations must be performed in a given transaction context.

Each transaction selects the current *version*, i.e. the visible sub-database. It also specifies the "scope" of the local changes. Local changes are made visible to the parent transaction (and its children) upon commit, and possible conflicts must be resolved.

Change Oriented Versioning

EPOSDB implements Change Oriented Versioning or COV. COV considers a set of physical changes to a set of (related) fragments as *one* logical (or functional) change. These changes may result from a sequence of update jobs, using the above transaction mechanism. COV is largely independent of the data model and enables uniform versioning of entities and relationships,

including Schema-level information. Thus, COV controls the *version space* at any granularity of data.

COV can be used together with normal check-out/in towards workspaces in a long transaction context. Checked-out configurations are bound in the *product space*, according to the given product model (entities and relationships).

A logical change is described by a boolean and global *option*. Logical changes can be freely combined, possibly constrained by stored version-rules (predicates).

COV generalizes conditional compilation on the entire database. In traditional versioning models, changes (deltas) are computed as the differences between versions, being atomic data objects. In COV, a version is not an explicit object, but can be evaluated as a combination of the selected changes (deltas). It applies to any granularity: atomic objects, subsystems, configurations, entire databases.

Traditional versioning models use a version tree/graph for each versioned object, thus creating a version group of "similar" objects. However, making version selections that combine (merge) changes done at different "parts" (branches) in the version tree is not automatic. In COV, the changes have no "history", and can in principle be freely combined. The version tree is flat, and version history is (at a low level) recorded by version-rules that regulate valid version combinations (see below).

In a version selection in COV, we must first specify an option binding of true/false values for the relevant options. This binding is called a *version-choice*, and will select the visible *version* of the entire database. Then a selection in the product space can be done. This is the inverse binding sequence of that in most other CM systems, although ADELE has an intermixed product/version binding sequence.

In a transaction, we must therefore give a version-choice (a filter) for reading the database. We must also give a possibly partial option binding, called an *ambition*, for writing back to the database. The ambition specifies the scope of changes done in a transaction. Many more "versions" may thus be affected by the local changes, than the one given by the version-choice. Hence, changes is automatically propagated (merged) into *many* sub-databases. The ambition will also "lock" a part of the version space for concurrent updates.

If ongoing transactions have overlapping ambitions and sub-products, we should encourage such sibling transactions to cooperate, to avoid merges afterwards or even loss of information by over-write or rollback.

In spite of many similarities between COV and traditional versioning, the main difference lies in: intentional and flexible version selection based on logical changes, explicit representation of the version space (e.g. for locking), and versioning being orthogonal to the stored data and its granularity.

3.3.4 EPOS Project Context

The overall infrastructure of the EPOS process support tasks are as follows: a long EPOSDB transaction is associated to a `project` task.

At start up, a transaction, that defines a database version of the entire process model, and a local Project governing this are started. This may require negotiation and delegation from a parent project, e.g. according to an incoming change-request. Under the local `Project` task, the Planner will generate the infrastructure of subtasks.

The PM Manager is used to refine and adapt the process schema that has been inherited from the parent project into a more specific one. This meta-activity results in appropriate class descriptions of local activities, products, production tools, and roles. Facilities for impact analysis of changes may be modeled by procedures in special task classes. Such schema evolution can also be done incrementally later.

Cooperation protocols, i.e. negotiation and propagation rules and patterns against possible overlapping neighbor projects/transactions may also be established. Thereafter relevant workspace files are checked out from the database.

Then, subtasks can be gradually (re)planned and (re)enacted – respectively in meta-activities PM4 and PM5. This process depends on the actual production activities, e.g. product structure changes, and meta-activities (class changes).

Finally, the local Project task will check-in the modified workspace files to the database, and close itself after committing the database transaction. The Project Manager of the parent project is notified about all this.

3.4 Methodologies of Change in EPOS

A process model always stands in a local project context. Thus, the meta-process of changing fragments of such model also occurs in such a context. If some changes are found to be useful elsewhere, these may be propagated to other projects.

We define a process model to be *inconsistent*, if the meta-classes, the classes, and their instances may lead to an unpredictable behavior of the PM tools.

E.g. if a task class has no `pre_dynamic` condition, the Execution Manager will never be able to enact the corresponding tasks.

Changing Instantiated/Enacting Models

Instantiated/Enacting models consist of objects representing products, activities, tools, and human roles. EPOS relies on the CM facilities offered by EPOSDB to update product configurations. Tasks may also be manipulated, i.e. they may be created, started, suspended, restarted, and killed. Tool descriptions may as well be changed, e.g. new switches may be added or the executable files may be substituted.

Hard and *soft* schema changes are distinguished. The *hard* changes imply changes in the structure of either the instance-level attributes or the subclasses. The *soft* schema changes are the process-specific ones, as they modify the behavioral part of a class. Hard changes are prohibited by EPOSDB. Thus, we may have to create a new subclass of the actual class, and explicitly convert a subset of the instances to the new subclass definition.

Changes to one instance may affect instances related to it, and may therefore leave the system in an inconsistent state. E.g. if the relation instance connecting a task to its parent task is deleted, the child task execution may lead to unpredictable results. However, such updates are not prohibited, but it is the responsibility of the (privileged) user performing the changes, to reinstate system consistency. The distinction among different kinds of users and relative access right mechanisms is still under design.

Changing Process Schemas

A schema *change request* and an actual schema class change are distinguished. Fig. 7 depicts the meta-process of evaluating the impact of a schema change request and optionally performing the change on a class definition. The feasibility of a requested class change has to be evaluated against its possible impact on the whole process model.

A class change may affect the *extent* of the class, i.e. the instances of the modified class, but also the related classes of the modified class and their extents. The *related classes*[6] of a class are: its subclasses and the entity classes related by class level relations, e.g. `formals`.

Thus, when a schema change request is issued, the PM Manager checks, for each of the possibly affected

[6] The related classes are connected to the given class by class-level relationships.

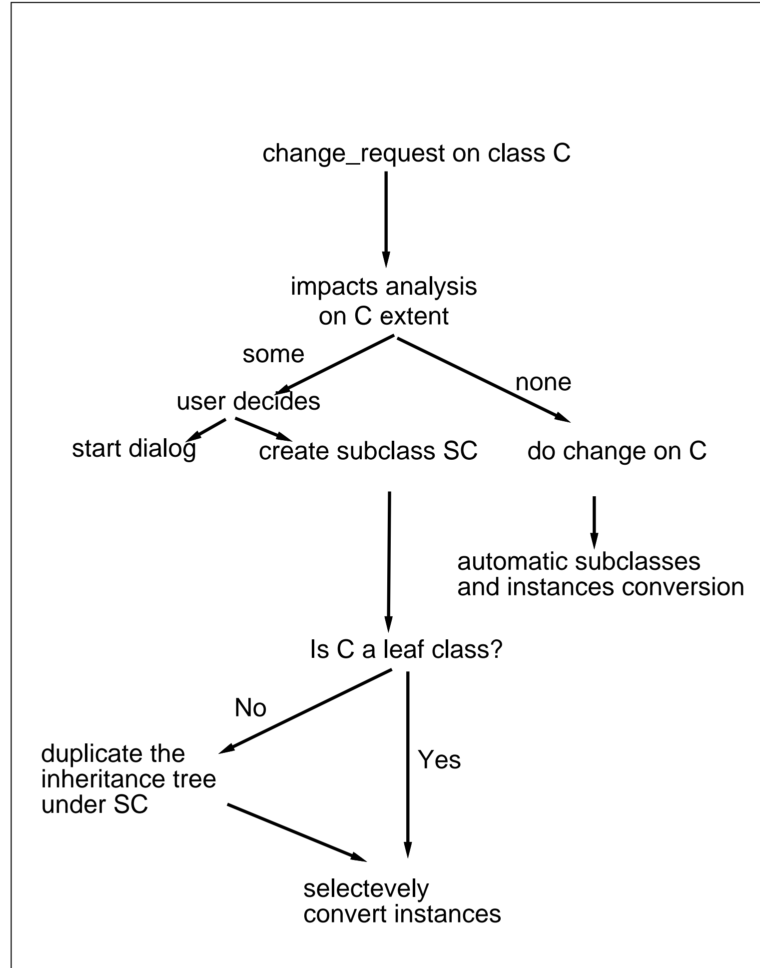

Figure 7: The meta-process of Process Schema change.

related classes and instances, the possible effects of the requested change.

If a requested change does not have any impact, the PM Manager is free to perform the requested change. All the existing instances and subclasses are automatically converted to the new definition.

If the requested change has some impact, i.e. some actions will be needed to reinstate consistency, the possible consequences are then displayed to the user, and a dialog is started. The user may decide to create a new subclass by *refining* the class so that only a *subset* of the extent of the old class will be affected. Then, some existing instances may be converted to the new subclass definition, thereby possibly adding and initializing new attributes. If the modified class is not a leaf node in the inheritance tree, i.e. if it already has subclasses, the whole inheritance tree has to be duplicated under the new created class. This is of course a disadvantage of refining by subclassing versus modifying. On the other hand, the advantage is that the old class is retained and the conversion of the existing instances to the new definition may occur in a selective and controlled way.

If the user wants to modify the class despite the suggested impact, again he interacts with the system to selectively perform some of the suggested actions.

The interactive dialog with the user is performed through a *dialog window* composed by two bottoms and one menu as displayed in fig. 11.

The only changes that are not allowed are those that violate the following class invariants:

- A class name must be *unique* over all projects.
- A class must have *one* super class (single inheritance).
- The class and all super classes of an instance must be visible in the actual project.
- Likewise for the version visibility of the related entities of a relationship.
- Likewise, the related classes mentioned in the `decomposition` and `formals` must exist.

For task classes we can further distinguish between the following soft class changes:

- Changed `code`: this assumes that no task instance is currently enacting it, i.e. the affected tasks are all passive. This affects the Execution Manager.
- Changed `pre_dynamic` or `post_dynamic`: this can technically be done as soon as no task instance is not evaluating these. This also affects the Execution Manager.
- Changed `formals`, `decomposition`, `pre_static`, or `post_static`: this forces the Planner to reexamine the affected tasks in the task network to check the constraints, and possibly rebuild parts of the network (if feasible).
- Changed procedure: this also assumes that no task is executing this.

Changing the Meta-Schema

The knowledge of how to create a subclass of a given class, to change the class, to create instances of it, and to convert instances to a new behavior is defined in its meta-class defined in the Meta-Schema. It happens that the first time we create a class, we do not want to bother with how to reuse and maintain it. That is, we do not redefine the `class_create`, `class_change`, and `instance_convert`, but let them be inherited from the superclass. Only when a class has been used and found useful, class-level procedures telling how to change and reuse the class may be added.

The PM Manager implementation

The PM Manager is in charge of defining, refining, and modifying classes. The main procedures implementing the PM manager are `class_create`, `class_change`, `instance_convert`, and `restart`.

Procedure class_create

Procedure `class_create`, defined in the root `entity` class and possibly redefined for its subclasses, implements meta-activity PM2–PM3 concerning the Schema. This corresponds to definition and compilation. The PM Manager invokes the `class_create` of the given superclass with the following subclass data as parameters:

- name of new subclass;
- definition of new instance-level attributes;
- definition of new instance-level procedures or redefinition of existing ones;
- redefinition of values of pre-defined class-level attributes;
- definition of new class-level procedures or redefinition of existing ones.

The definition of a new subclass cannot, of course, affect the definitions of existing classes.

Procedure class_change

Procedure `class_change` attempts to update procedures or class-level attributes of a class, and has the same parameters as `class_create`. All the instances of the old class are implicitly converted to be instances of the modified class.

Procedure `class_change` can change a `PM_entity` subclass, and tries to preserve the consistency of the system state.

A class change may affect: the *related* classes of the modified class; and the instances of the modified class, including the instances of related classes.

Procedure `class_change` operates a single change on a `PM_entity` subclass. Since the Planner and the Execution Manager relies on class-level attributes to manage process model instantiation and enaction, such changes may lead to inconsistent situations.

The procedure `class_change` is in charge of evaluating the impact of the proposed change and to find those actions that are necessary to put the system in a consistent state again.

However, as the proposed actions may have a deep impact, a dialog is started with the user who may choose not to carry out all the proposed actions, or even to cancel the change request.

Fig. 8 shows the template model fragment of the procedure `class_change` that implements the modification of dynamic pre-conditions. Lines 2-3 specify that for each instance of class `SelfC`[7], the state is read. If the `task_state` is `waiting` or `created`, the given instance is not affected. Otherwise, it means that the pre-condition was evaluated to true before, and the pre-condition should be re-evaluated (line 8). If this evaluation leads to true, it means that the task is not affected; if not, the task has to be restarted.

Procedure restart

Procedures `restart` and `stop` are heavily used by the PM Manager for stopping, converting, and restarting instances of evolving classes.

The instance-level procedure `restart` suspends the work done by a task instance and put its state to `initiated`. This procedure must rollback the actions that have been done by the task itself and its subactivities, and by the tasks that operate on the data that it has produced.

Performing a possible rollback of the actions means to reset the output data to the state they had before the task was started, to kill possible forked operating

[7] `SelfO` and `SelfC` are used to denote the instance or class a procedure is invoked on.

```
1   class_change(Sender,SelfC,pre_dynamic-X):-
2      call_proc(SelfC,SelfC,read_relation,instance_of,I)
3      call_proc(SelfC,I,read,task_state,S),
4      (member(S,[waiting,created])->
5           call_proc(SelfC,Sender,notify,
6                     ["task",I,"in state",S,"not affecte
7       member(S,[active,waiting_children,forked,terminat
8           (eval(X)->
9              call_proc(SelfC,Sender,notify,
10                     ["task",I,"in state",S,"not affect
11             call_proc(SelfC,I,restart,[],_),
12             call_proc(SelfC,Sender,notify,
13                  ["task",I,"in state",S,"restarted"]))
14  fail;
15  call_proc(SelfC,SelfC,write,pre_dynamic,X),
16  call_proc(SelfC,Sender,notify,["work done"]).
```

Figure 8: The class change procedure for dynamic pre-condition.

```
1   restart(Sender,SelfO):-
2     call_proc(SelfO,SelfO,read,task_state,S),
3           (member(S,[created,waiting])->
4                true;
5            S=active->
6                (call_proc(SelfO,SelfO,read_relation,act
7                 call_proc(SelfO,Out,reset,[],_),
8                 fail;
9                 true);
10            S=waiting_children->
11               (call_proc(SelfO,SelfO,read_relation,sub
12                call_proc(SelfO,T,restart,[],_),
13                fail;
14                call_proc(SelfO,SelfO,read_relation,act
15                call_proc(SelfO,Out,reset,[],_),
16                fail;
17                true),
18            S=forked->
19               (call_proc(SelfO,SelfO,read,os_pid, Pid)
20                kill_os(Pid),
21                call_proc(SelfO,SelfO,read_relation,act
22                call_proc(SelfO,Out,reset,[],_),
23                fail;
24                true),
25            S=terminated->
26               (call_proc(SelfO,SelfO,read_relation,act
27                call_proc(SelfO,Out,reset,[],_),
28                fail;
29                true)),
30    call_proc(SelfO,SelfO,write,task_state,waiting).
```

Figure 9: Procedure restart.

system jobs, and to eliminate all the side-effects associated to `code` enaction. Otherwise, the conditions on which the Planner is based to perform automatic instantiation would not hold.

However, it is not possible to determine automatically the effects of `code` enaction. Thus we assume that an instance-level procedure `fail` specifies actions to "undo" the results from the `code` part.

The implementation of procedure `restart` is given in fig. 9. Procedure `restart` reinitiates the task state and possibly backtracks the actions performed by the task. Procedure `restart` first inspects the state of SelfO (line 2). If this state is `created` or `initiated` no action is taken; otherwise the procedure `fail` is invoked.

The procedure `fail` may be alternatively implemented by an interaction with the user. For instance, it is not wise to cancel all the effects performed by very long transaction tasks.

3.5 An example: the ISPW7 Reference Problem

We here give the EPOS solution to the reference problem of software process model change, as proposed for by 7th International Software Process Workshop (ISPW7) [22].

Let us suppose we have a process schema that includes a `coding` task class, stating that it is possible to begin coding before the design is approved. Suppose that it is later decided to tighten the schema requirements detailing when coding can begin, so that the design must be approved before coding begins. Furthermore, assume that this schema change affects only one currently instantiated or enacting process model fragment (task object).

This example does not explore all the possibilities discussed before, because there is only one instance is affected, there is only one project, class `coding` does not have any subclasses. In EPOS, this means that the dynamic pre-condition of the `coding` class must be modified. The requested change does not violate any consistency constraints, thus the class `coding` may be changed by modification. However, we show how the problem can be solved in EPOS by using either `class_change` (overwrite) or `class_create` (subclassing). In fig. 10, we depict a scenario where two parallel subprojects, `PMB` and `PMC`, co-exist under a superproject `PMA`. The class `coding` has originally been defined in the context of the project `PMA`. The set of classes created in project `PMA`, is available to both subprojects `PMB` and `PMC`. In subproject `PMB` the Process Schema change is performed by subclassing (`class_create`), while in subproject `PMC` by overwrite (`class_change`).

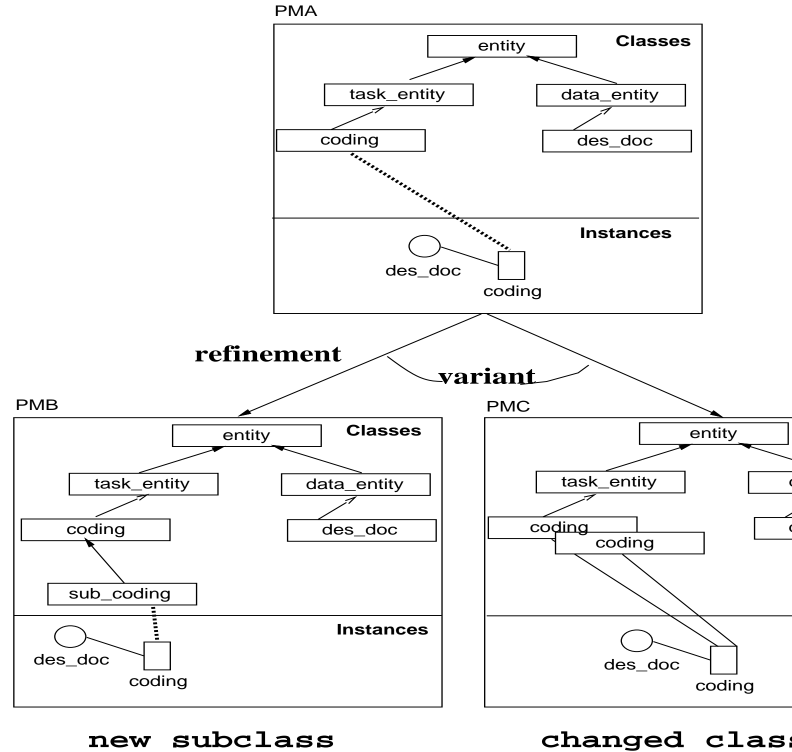

Figure 10: A scenario based on subclassing and versioning.

In subproject `PMB`, the knowledge (process model, database) of its superproject `PMA` is customized by creating a subclass `sub_coding`, but still retaining the classes as defined in the superproject. In subproject `PMC` the class definition is changed.

Lastly, when subclass `sub_coding` is created from class `coding` in subproject `PMB`, the existing instance is implicitly converted to an instance of the new subclass. On the other hand, when class `coding` is updated in subproject `PMC`, the existing instance is automatically converted to an instance of the modified class. In both cases, the task may have to be restarted.

Fig. 11 shows the change dialog window that is displayed, in case the dynamic pre-condition of class `coding` is requested to be changed, and there is one `coding` instance with two existing subtasks, one of class `edit` and one of class `compile`. As displayed in fig. 11, the suggested actions are: (1) updating of the static pre-conditions of class `coding`, (2) restart of the `edit`, `compile`, and `testing` instance, (3) reset of both the `edit` and `compile` output. The user may or may not choose the suggested actions. Concerning re-

setting of products, we have chosen not to delete their file attributes, if these are not produced by automatic tools, only to reset the state of the objects. This is because the work done has not to be automatically destroyed, but the responsible user has to be notified that some inconsistencies may have been introduced.

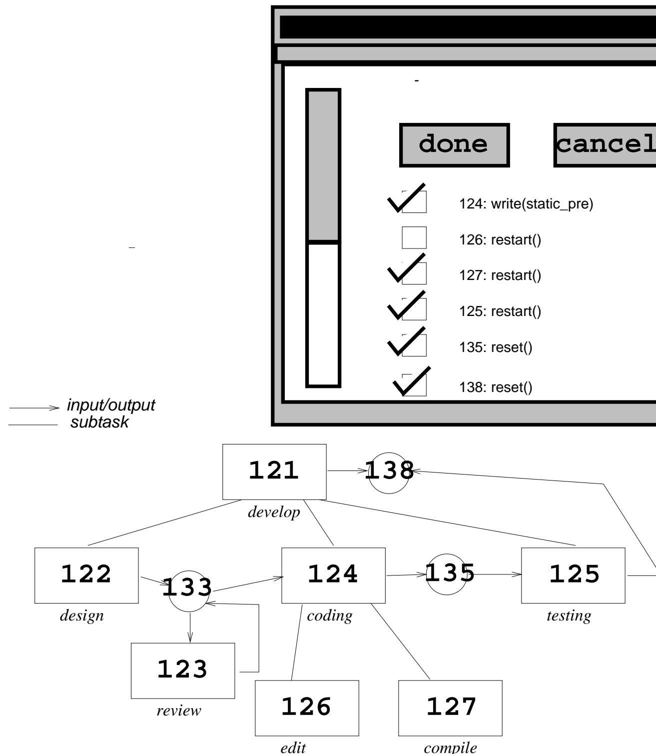

Figure 11: The EPOS task network and change dialog window.

4 Comparisons and Related Work

Many PSEs have been prototyped and documented over the last 5 years and some experiments in realistic external production environments, have been reported (Process Weaver [23], IPSE 2.5 [24]). In addition, some large examples have been run by the development teams, and several PSEs can assist in maintaining themselves (ADELE [11], MARVEL [25]).

In the following, the main characteristics of the EPOS system are compared against those offered by some other systems. During this comparison process we take as parameters both the general process evolution issues and the specific EPOS solutions. Among the general issues, 1) reflection and meta-process, 2) change assessment simulation and validation, 3) when process evolution may happen, 4) items of change, are taken into consideration. Among the specific EPOS choices, 1) Object Orientation, 2) Database support and versioning, 3) Automatic and incremental instantiation by planning, are considered.

Reflection and meta-process The EPOS meta-process is explicitly represented by meta-classes. This is strongly influenced by the meta-class mechanism in Smalltalk [16]. CLOS [26] also offers reflective features and pre-defines procedures to change class definitions, and to convert the affected instances. Similarly, SELF [27] provides rules to control evolution.

Further, IPSE 2.5 offers reflection, while MARVEL has a fixed meta-process expressed in another (non-reflective) language. SPADE [28] offers only task-level reflection. Laws to control evolution of both product and of the rules themselves may be defined DARWIN [29].

The use of reflection to manage process model evolution is not new, but EPOS exploits an integrated, object-oriented architecture for managing class changes.

Change assessment, simulation, and validation In most PSEs, relationships can be used to control and propagate the impacts of change. At the moment, EPOS does not provide facilities for simulation of changes, and weak mechanisms for formal verification of changes. MARVEL [25], Merlin [30] and IPSE 2.5 [24] can offer some formal verification support, and MELMAC [31] can perform simulations.

When process evolution may happen

Process model changes fall into two main categories: refinement/customization *before* enaction, as in Process Weaver, and MELMAC [31]; or correction *after* enaction, as in MARVEL, IPSE 2.5, SPADE, and EPOS. To implement correction after enaction either late/dynamic binding or rebuild mechanisms are needed. Generally the second category includes the first.

Items of Change HFSP [32] enables to define meta-rules for changing the enaction state. This can be done also in EPOS. ARCADIA [33] can add triggers and turn on/off predicates at runtime whereas EPOS does not offer the possibility of imposing global constraints, such as predicates.

Adding new (production) tools is easy in most systems and it corresponds to the tool installation and subsequent addition of a tool envelope. A tool envelope corresponds to a task class in EPOS. To change a tool interface or to remove it, is much harder and may lead to loss of functionality.

Among PSEs that have an explicit representation of team structure, Merlin [30] allows this to change. Other PSEs integrate with an external project management tool to perform such actions, as for Process Weaver. EPOS can offer only some initial functionalities and it is not integrated with a project management tool.

Process model schema fragments are items of change in EPOS, as in SPADE, IPSE 2.5, and PRISM whereas HFSP [32] enables to change enacting models, but not schemas.

PRISM [6] offers a Dependency Structure for describing change items and a Change Structure for describing change related data. The structure of EPOS instances, classes, and meta-classes connected by relations resembles the PRISM Dependency Structure whereas EPOS does not manage change related data, i.e. maintenance reports or history of changes. As PRISM, EPOS provides a way for incrementally defining or refining class level procedures to implement changes.

Object-Orientation Among the PSEs exploiting object-orientation, the closest to EPOS is IPSE 2.5 using PS-Algol, and partly MARVEL. ADELE has a hybrid object-oriented model, with run-time binding (delegation [34]) towards product and/or project contexts for customization. EPOS has derived dynamic binding and class-properties from Smalltalk [16] to gain flexibility.

Database Support and versioning Many of the PSEs have their own and partially proprietary Object Management Systems (OMSes), e.g. MARVEL, ARCADIA (CHIRON). Others use existing OMSes, e.g. PS-Algol used in IPSE 2.5, PCTE in ALF [35], O2 in SPADE, and LDL in OIKOS [36].

ADELE and EPOS are the only systems that rely on a fully versioned OMS, thus integrating CM and PM. Both exploit triggers and nested transactions. However, ADELE does not apply the same versioning on the schema as on product descriptions: the Adele schema is bound to substitutable project contexts, and classes are not first order objects.

Planning The EPOS Planner uses domain-independent, non-linear planning to incrementally (re)construct task networks. As mentioned, this is inspired by TWEAK [19], and also by IPEM [20]. Other PSEs using goal-oriented AI techniques are GRAPPLE [37], and for a similar purpose as EPOS. SPADE uses reflection to incrementally construct its task network (a PetriNet). However, neither of them have facilities for incrementally rebuilding ("replanning") the network after product or class changes.

5 Conclusions

An EPOS template process model consists of a process schema of classes and meta-classes and its instances, describing the external process entities and their relationships. An instantiated and enacting process model is represented by task instances, linked to product, tool, and role instances.

The originality of the EPOS approach to process model evolution lies in three parts: 1) a uniformly versioned database to store the entire process model and offering nested cooperative transactions under PM control; 2) a reflective and fully object-oriented data model accessible through SPELL to flexibly define and evolve a Process Schema and its instances; 3) a Planner to incrementally and dynamically (re)generate task networks.

The EPOSDB is based on C-ISAM, with client-server protocols using Sun RPC. The server is implemented by 22,000 C lines and the client by 6,000 C lines. EPOSDB offers a Prolog based interface. The PM tools including the Planner are implemented by 7000 SWI-Prolog lines. The User Interface uses the PCE Prolog-based graphical package.

EPOS has been demonstrated on a set of examples, covering software systems with some dozens of modules, including parts of the ISPW7 example. We are now starting to apply EPOS on itself, and on prototyping external applications in several domains together with three Norwegian software companies. Facilities for process model evolution is judged crucial for these test examples.

The EPOS system has not yet been demonstrated to be "open-ended" versus other platforms and application domains, i.e. it lacks distribution and federation aspects. Also, COV is promising, but unproven technology. However, EPOS PM should be portable on top of other object-oriented DBMSes with a different versioning model.

Of the specific drawbacks of the EPOS process model evolution support, we can mention: poor support for assessment, simulation and validation of changes. In addition, the exploitation of CM techniques for process evolution should be improved.

Our future work will try to rectify the above drawbacks with particular emphasis on change assessment, simulation, and validation. Further, a more high-level PML supported by a small CASE tool for PM is under design as an extension of the PM Manager. Finally, the relation between processes and meta-processes has to be better elaborated, both on the conceptual and on the technical level.

References

[1] M. Dowson, B. Nejmeh, and W. Riddle, "Fundamental Software Process Concepts," in *[38]*, pp. 15–37, 1991.

[2] N. H. Madhavji, "The process cycle," *Software Engineering Journal*, vol. 6, pp. 234–242, Sept. 1991.

[3] R. Conradi, C. Fernström, A. Fuggetta, and R. Snowdon, "Towards a Reference Framework for Process Concepts," in *J.-C. Derniame (ed.): Proc. from EWSPT'92, Sept. 7–8, Trondheim, Norway, Springer Verlag LNCS 635*, pp. 3–17, Sept. 1992.

[4] P. H. Feiler and W. Humphrey, "Software Process Development and Enactment: Concepts and Definitions," Jan. 1992. 12 pages (Second version).

[5] M. M. Lehman and L. A. Belady, *Program Evolution — Processes of Software Change*. Academic Press, 538 p., 1985.

[6] N. H. Madhavji, "Environment evolution: The prism model of changes," *IEEE Trans. on Software Engineering*, vol. SE-18, pp. 380–392, May 1992.

[7] P. H. Feiler, "Configuration management models in commercial environments," tech. rep., Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, Mar. 1991. 53 pp.

[8] R. Conradi, E. Osjord, P. H. Westby, and C. Liu, "Initial Software Process Management in EPOS," *Software Engineering Journal (Special Issue on Software process and its support)*, vol. 6, pp. 275–284, Sept. 1991.

[9] R. Conradi, M. L. Jaccheri, C. Mazzi, A. Aarsten, and M. N. Nguyen, "Design, use, and implementation of SPELL, a language for software process modeling and evolution," in *J.-C. Derniame (ed.): Proc. from EWSPT'92, Sept. 7–8, Trondheim, Norway, Springer Verlag LNCS 635*, pp. 167–177, Sept. 1992.

[10] M. H. Penedo and C. Shu, "Acquiring Experiences with the modelling and implementation of the project life-cycle process: the PMDB work," *Software Engineering Journal (Special Issue on Software process and its support)*, vol. 6, pp. 259–274, Sept. 1991.

[11] N. Belkhatir, J. Estublier, and W. Melo, "Software Process Model and Work Space Control in the Adele System," in *Leon Osterweil (ed.): Proc. from 2nd Int'l Conference on Software Process (ICSP'2), March 1993, Berlin. IEEE Press*, pp. 2–11, 1993.

[12] A. Lie, R. Conradi, T. M. Didriksen, E.-A. Karlsson, S. O. Hallsteinsen, and P. Holager, "Change Oriented Versioning in a Software Engineering Database," in *Walter F. Tichy (Ed.): Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, USA, 25-27 Oct. 1989, 178 p. In ACM SIGSOFT Software Engineering Notes, 14 (7)*, pp. 56–65, Nov. 1989.

[13] B. Gulla, E.-A. Karlsson, and D. Yeh, "Change-Oriented Version Descriptions in EPOS," *Software Engineering Journal*, vol. 6, pp. 378–386, Nov. 1991.

[14] Object Management Group, *Object Services/Data Model - Request for Information*, Sept. 1990.

[15] J. Rumbaugh, "Relations as semantics constructs in an object-oriented language," in *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, (Kissimmee, Florida), pp. 466–481, Oct. 1987. In ACM SIGPLAN Notices 22(12), Dec. 1987.

[16] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[17] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "SIMULA Information — Common Base Language," Tech. Rep. 145 p., S-22, Norwegian Computing Center, Oslo, 1970.

[18] C. Liu, "Software Process Planning and Execution: Coupling vs. Integration," in *Proc. of CAiSE'91, the 3rd International Conference on Advanced Information Systems, Trondheim, Norway, 13–15 May 1991* (R. Andersen, J. A. B. jr., and A. S. lvberg, eds.), pp. 356–374, LNCS 498, Springer Verlag, 578 p., 1991.

[19] D. Chapman, "Planning for conjunctive goals," *Artificial Intelligence*, vol. 32, pp. 333–377, 1987.

[20] J. A. Ambros-Ingerson and S. Steel, "Integrating planning, execution and monitoring," in *Proc. of AAAI'88*, pp. 83–88, 1988.

[21] C. Liu and R. Conradi, "Automatic Replanning of Task Networks for Process Model Evolution in EPOS," in *Ian Sommerville (Ed.): "Proc. from the 4th European Software Engineering Conference (ESEC'93)", Garmisch-Partenkirchen, FRG. Forthcoming as a Springer LNCS. 17 p*, Sept. 1993.

[22] M. I. Kellner, P. H. Feiler, A. Finkelstein, T. Katayama, L. Osterweil, M. Penedo, and H. D. Rombach, "Software Process Modeling Problem (for ISPW6)," Aug. 1990.

[23] C. Fernström, "Process WEAVER: Adding Process Support to UNIX," in *Leon Osterweil (ed.): Proc. from 2nd Int'l Conference on Software Process (ICSP'2), Berlin. IEEE-CS Press*, pp. 12–26, Mar. 1993.

[24] R. Snowdon, "An example of process change," in *J.-C. Derniame (ed.): Proc. from EWSPT'92, Sept. 7–8, Trondheim, Norway, Springer Verlag LNCS 635*, pp. 178–195, Sept. 1992.

[25] N. S. Barghouti and G. E. Kaiser, "Scaling up rule-based development environments," *International Journal on Software Engineering and Knowledge Engineering, World Scientific*, vol. 2, pp. 59–78, Mar. 1992.

[26] S. E. Keene, *Object-Oriented Programming in Common Lisp.* Addison Wesley, 1989. 266 p.

[27] D. Ungar and R. B. Smith, "Self: The Power of Simplicity," in *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, (Kissimmee, Florida), pp. 227–242, Oct. 1987. In ACM SIGPLAN Notices 22(12), Dec. 1987.

[28] S. Bandinelli and A. Fuggetta, "Computational Reflection in Software Process Modeling: the SLANG Approach," in *Proc. ICSE'15, Baltimore, USA, IEEE-CS Press (forthcoming)*, May 1993.

[29] N. Minsky, "Law-Governed Systems," *Software Engineering Journal*, vol. 6, pp. 285–302, Sept. 1991.

[30] W. Emmerich, G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf, "MERLIN: Knowledge-based Process Modeling," in *[38]*, pp. 181–187, 1991.

[31] V. Gruhn, "The Software Process Management Environment MELMAC," in *[38]*, pp. 191–201, 1991.

[32] T. Katayama, "A Hierarchical and Functional Software Process Description and its Enaction," in *Proc. of the 11th Int'l ACM-SIGSOFT/IEEE-CS Conference on Software Engineering, Pittsburgh, PA*, pp. 343–352, 1989.

[33] S. M. S. Jr., D. Heimbigner, and L. Osterweil, "Language Constructs for Managing Change in Process-Centered Environments," in *Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, California. In ACM SIGPLAN Notices, Dec. 1990*, pp. 206–217, Dec. 1990.

[34] L. A. Stein, "Delegation is inheritance," in *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, (Kissimmee, Florida), pp. 138–146, Oct. 1987. In ACM SIGPLAN Notices 22(12), Dec. 1987.

[35] F. Oquendo *et al.*, "A Meta-CASE Environment for Software Process-centred CASE Environments," in *Proc. Int'l Conf. on Advanced information Systems Engineering (CAiSE'92), Manchester, UK. Springer Verlag LNCS 593*, pp. 568–588, May 1992.

[36] V. Ambriola, P. Ciancarini, and C. Montangero, "Software Process Enactment in Oikos," in *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, California*, pp. 183–192, 1990.

[37] K. E. Huff and V. R. Lesser, "A plan-based intelligent assistant that supports the software development process," in *Proc. of the 3rd ACM Symposium on Software Development Environments*, (Boston, Massachusetts), pp. 97–106, Nov. 1988.

[38] A. Fuggetta, R. Conradi, and V. Ambriola, eds., *Proceedings of the First European Workshop on Process Modeling (EWPM'91)*, (CEFRIEL, Milano, Italy, 30–31 May 1991), Italian Society of Computer Science (AICA) Press, 1991.