Techniques for Process Model Evolution in EPOS, V1-rev.

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

May 25, 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 enactment 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

πάντα ῥεῖ

Heraclitus of Ephesus

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. Software² Process Modeling (PM) is the discipline of describing process models [?] [?]. [?]

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).

¹All things are in a state of flux.
²The “software” prefix may often be omitted in the following.
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 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 [?], 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 [?] [?]. This includes Software Process Customization: reusing existing process model fragments and adapting them to different contexts.

The paper is structured as follows: section ?? defines a process model life cycle, and elaborates the meta-process for process evolution and customization. Section ?? presents the EPOS support for process model evolution. Section ?? discusses some related work and tries to compare EPOS features with those offered by some existing systems. Conclusions are given in section ??, 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 [?], 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. ?? 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 en-actable 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, enactment, 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 enactment, 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 enactment 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. ?? 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 be-
tween 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. ?? may prove to be inadequate and need to be changed.

**Instantiated/Enacting process model** Starting from the bottom level of fig. ??, 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. ?? gives a CM perspective of PM change. Here, the terms revision and variant (branch) are given the classical CM semantics [?]. 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.

Such a meta-process should encode aspects of a change methodology to
Figure 3: Changing Process Models.
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\textsuperscript{3} [?] is a PSE. It offers a PML called SPELL (Software Process Evolutionary Language) [?], an initial Process Schema, and a set of process tools.

In EPOS, the main \textit{internal} process model is a network of chained and decomposed activity descriptions (tasks), coupled to descriptions of other activities, products, tools and users.

The template process model, a Process Schema, is a set of entity and relation classes. The meta-Schema part of the Process Schema is represented by a set of meta-classes. The instantiated and enacting process models are instances of Process Schema classes, and make up the above network.

Thus, EPOS process model fragments are meta-classes, classes, and their instances. (The communication protocols mentioned in Section ?? are technically “instances”, but express overall control information.)

The main process tools operating on the above models are: a \textit{Schema Manager}, an \textit{Execution Manager} (Process Engine), and an \textit{AI task network Planner}. In addition comes \textit{EPOSDB}, a versioned object-oriented database.

3.1 The Layered EPOS Architecture

PSEs that rely on an object-oriented database, e.g. PMDB [?] and ADELE [?], often have a PML as a layer around the underlying database. The EPOS layers are:

\textsuperscript{3}EPOS: Expert system for Program and ("Norwegian Og") System development.
1. A client-server EPOSDB, with change oriented versioning [?] ([?]) 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. 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 [?].

A free-standing DML using Prolog is offered at the client side. All entities, relationships, and their classes and meta-classes (as class descriptor instances) are stored in the database. They are thus uniformly manipulatable and versionable (at least technically).

2. A reflective and behaviorally object-oriented data model, available through SPELL. SPELL unifies and extends the underlying DDL/DML, and offers class-level attributes and instance/class-level procedures and triggers. Meta-classes are used reflectively as in Smalltalk to store class-level information, and the underlying EPOSDB meta-classes are instrumented for SPELL extensions.

SPELL is implemented on top of Prolog, and available through the SPELL Interpreter [??]. This EPOS layer supports meta-activities Analysis & Design (PM2) and Customization (PM3), and later Evolu-

\footnote{In the EPOS literature they are called types, but here we adhere to the OMG [?] terminology.}
tion?? through a Schema Manager (??).

3. A **tasking framework** for concurrent enaction of process models. This is done by the Execution Manager, our Process Engine. This interprets special class-level attributes defined in a predefined **task_entity** class (??). The Execution Manager cooperates closely with the Planner to incrementally (re) instantiate task networks, and with external tools.

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

4. **Application-specific process models and process tools**, relying on SPELL and its Process Engine. Such models are domain-specific and “user”-defined, and include both Schemas and instances. For instance, they can describe the user’s software product structures, production tools etc.. Relationships to describe vertical aggregation and horizontal dependencies of both activities and products are commonly used.

The **project** class describes EPOSDB transactions, and the **develop** class describes a general software development process. Of additional process tools we can mention a Project Manager and a Cooperation Manager for meta-activity PM4 (??).

Figure ?? shows how the six?? meta-activities from Figure ?? are implemented by the EPOS process tools.
Figure 4: The actual meta-process in EPOS.
3.2 The SPELL Language

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

Classes and Model Structuring

Figure ?? displays some system-defined and predefined classes, showing the same layering as in Section ??.

Figure 5: The EPOS Classes.

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. For instance, we can explicitly model relations between classes (meta-relations). Such relations can be used to ex-
plicitly model subclassing relations, or some internal relations that encode the formals and decomposition class-level attributes in the task_entity class (??).

Inheritance and Protection

As mentioned, single inheritance is provided for all properties (attributes, procedures, triggers). A subclass may redefine class-level attributes and instance/class-level procedures/triggers. 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 Figure ??). concatenate is inspired by the Simula [?] 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, achieving some degree of modularization.

The SPELL Interpreter: Dynamic Binding

The access and binding of procedures, triggers, and attributes, both instance- and class-level, are dynamic as in Smalltalk. It is implemented by our
**SPELL Interpreter**, consisting 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 for Task Network Nodes**

A simplified definition of this basic task class stands in Figure ???. Generally, a task has actual input/output parameters to facilitate horizontal chaining. The classes of these “product” parameters are constrained by the **formals** class-attribute. Likewise, it has a **decomposition** class-attribute to constrain the classes of its vertical subtasks. It also has static and dynamic pre/post-conditions (four in all), and a **code** script that can be enacted (see Execution Manager in ??). The meaning of all these class-level attributes will be explained in the next subsections.

---

\(^{5}\)In the specification of Prolog procedures, parameters prefixed by ? are optionals, by + are mandatory input parameters, and by – are output parameters.
Low-level activations of atomic tools are modeled by a task subclass with empty decomposition. Such classes serve as tool envelopes, where extra actions serve to update actual parameters, thus “triggering” neighbor tasks.

More high-level activities, whose main work is delegated to their generated subtasks (see Planner in ??), are described by a task subclass with non-empty decomposition.

3.3 The Process Tools

In the following, we will describe the Execution Manager (??), Planner (??), and partly the Project Manager and Cooperation Manager (??). We will also explain the transaction and versioning model of the EPOSDB (??). The Schema Manager will described later in Section ??.

All these process tools use the XPCE User Interface package.

There are also some utility tools: an Extractor tool to install a product model from a set of file catalogs, ...??

3.3.1 The EPOS Process Engine: the Execution Manager

As mentioned, tasking is realised by the Execution Manager. It utilizes three class-level attributes:

- \texttt{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 (see below) and goal-directed vs. opportunistic execution (lazy vs. busy). This condition is (re)evaluated using polling, and can be optimized using triggers activated from preceding tasks (not
class task_entity: entity
{
    instance_level_attributes:
    task_state := created.

    instance_level_procedures:
    instance_delete(+Self0);
    instance_convert(+Self0, +New_Class);
    start(+Self0);
    restart(+Self0);
    stop(+Self0).

    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_Id);
    class_create(+SelfC, +Defined_Properties, -Class_Id);
    class_delete(+SelfC);
    class_change(+SelfC, +Defined_Properties).
}

Figure 6: The task_entity class.
explained here). The evaluation can have side-effects, e.g. by reading a mailbox.

- **code**, being a sequential program to perform the intended job of the given task class. Thus, enactment 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 prepare subtasking (??).

- **post_dynamic**, e.g. to treat errors, not mentioned here for simplicity.

The Execution Manager also maintains the instance-level attribute **task_state** (line 4 in Figure ??), whose value domain is: created during Planner instantiation (the default), waiting on its pre_dynamic condition, active during code enactment, 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 job to stop, or terminated upon goal-oriented enactment. Note, that waiting_children and forked are special cases of waiting.

**executor** refers to the logical name of the tool that should execute the task. **role** refers to the role or responsibility description of a generic human actor, e.g. a Software Developer. This description will be bound dynamically to certain persons or teams, and may define access rights.
3.3.2 The Planner

The AI Planner [?] is technically a procedure in meta-activity PM4. It is implicitly and incrementally invoked by the Execution Manager to detail (or plan) composite tasks. That is, the Planner will automatically generate a new subtask network for such tasks – and so on in due time. In practise, the Planner is mostly used to generate the subtasks of Develop and the Cooperation Manager (??).

It uses a domain-independent, non-linear AI planning algorithm, see TWEAK [?] and IPEM [?]. The integration of planning, execution and monitoring is at a coarse granularity through hierarchical planning.

The Planner starts with a composable Task and its desired Output, being 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 (KB), and a representation/model of the Product Structure (PS) as a World State Description (WSD).

The outer layer of the Planner is domain-specific to PM. It 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 PS. Indeed, the PS “drives” most of the task network.

The Planner utilizes four class-level attributes:

- pre_static and post_static express necessary conditions that must hold, respectively, before and after enactment of a task of the given task
class.

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

- **decomposition** specifies an unordered pool of candidate task classes for subtasks of the given composite task.

Note: An empty middle part in code assumes a non-empty decomposition.

These class-level attributes together specify legality constraints on the structure of the task network, cf. graph grammars. None of the above attributes are supposed to have any side-effects upon (re)evaluation, facilitating goal-oriented reasoning.

Clearly, changes to these Schema attributes and to the PS will affect the Planner. [?] describes an incremental algorithm for replanning that readjusts the existing task network to comply with the changes (which are usually local and small), rather than replanning from scratch.

### 3.3.3 EPOSDB: Change Oriented Versioning and Transactions

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

**Transactions**

EPOSDB offers long and nested transactions. Transactions may survive several application sessions, and are represented by special transaction objects in the database, connected to project tasks (??). The transactions are user-controlled – they may be started, committed or aborted interactively. All
database operations must be performed in a given transaction context. This selects the current version, i.e. the “visible” sub-database. It also specifies the “scope” of the local changes (see under COV below). Local changes are made visible to the parent transaction (and its children) upon commit, and possible conflicts must be resolved. Logging points for internal rollback can be implemented by a succession of subtransactions.

Actual production work is often in a checked-out and uni-version workspace, containing files and Prolog facts.

Change Oriented Versioning

As mentioned, EPOSDB implements Change Oriented Versioning or COV. 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.

It 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).

COV considers a set of physical changes as 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. Thus it fits well with a process-oriented view of software production.

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

25
In a sense, 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 inter-mixed 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. I.e. the ambition of a subtransaction must be a subset of its parent’s.

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

**Conclusion:** In spite of many similarities between COV and traditional versioning, the main difference lies in: intentional and very 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 and Meta-process

The overall infrastructure of the EPOS process support tasks are as follows: A long EPOSDB transaction is associated to a **project** task. Its most important subtasks are the **Schema Manager**, a **Project Manager** to start and finish child projects/transactions, a **Cooperation Manager** with a Workspace Manager to coordinate with possibly overlapping sibling tasks [?], and a top-level **Develop** task that contains the real production subtasks.

We get started by manually generating a transaction, that defines a database version of the entire process model, and a local **Project** governing this. This may require negotiation and delegation from a parent project, e.g. according to an incoming change-request. Under our local **Project** task, the Planner will generate the above infrastructure of subtasks.
First, we use the **Schema Manager** to refine and adapt the “inherited” Process Schema of 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, if the meta-model in the Schema Manager allows this.

We then use the **Cooperation Manager** to establish cooperation protocols (negotiation and propagation rules and patterns) against possible overlapping neighbor projects/transactions. Thereafter we check-out the relevant workspace files from database “long fields”.

Then the **Develop** task can start, and its subtasks can be gradually (re)planned and (re)enacted – respectively in meta-activities PM4 and PM5. This process depends on the actual production activities (PS changes) and meta-activities (class changes). In parallel, we may have communication with neighbor projects, and we can start and finish new subprojects via the Project Manager.

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.

NB: We have purposefully left out roles and access rights (locking).??
3.4 Methodologies of Change in EPOS

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

Changing Instantiated/Enacting Models

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.

We define a process model to be inconsistent, if the existing classes and their instances may lead to an unpredictable behavior of the process tools. E.g. if a task class has no pre\_dynamic condition, the Execution Manager will never be able to enact the corresponding tasks.

Hard and soft Schema changes are distinguished. The hard (but simple!) ones imply changes in the structure of either the instance-level attributes or the subclasses. The soft (but complex!) Schema changes are the process-specific ones, as they modify the behavioral part of a class. Hard changes are p.t. prohibited by EPOSDB, see however [?]?[?]. 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 there-
fore leave the system in an inconsistent state. E.g., if we delete a relationship connecting a task to its parent task, or a relationship connecting a product instance to its consuming task, we loose traceability. However, such updates are not prohibited per se, 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 has not yet been designed??.

**Changing Process Schemas**

A Schema change request and an actual Schema (class) change are distinguished, as described in Section ???. Figure ?? 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. This resembles classical CM authorization procedures.

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, two “role” entity classes if it is a relation class, and the classes “related” by formals and decomposition class-attributes if it is a task class.

Thus, when a Schema change request is issued, the Schema Manager checks, for each of the possibly affected related classes and instances, the possible effects of the requested change.

\(^6\)The related classes are connected to the given class by class-level relationships.
Figure 7: The meta-process of Process Schema change.
If a requested change does not have any impact, the Schema 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 s/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 Figure ??.

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 version- *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/trigger: this also assumes that no task is executing 
  this.

All in all, there are two axes of structuring and variability in EPOS Process 
Schemas. There *Vertical* refinement by subclassing inside projects and by
nested subprojects, and *horizontal* evolution inside projects and versioning between projects.

**Changing the Meta-Schema**

The knowledge of how to create a subclass, to change the class, to create instances of it, and to convert instances to a new behavior is defined in its meta-class standing in the Meta-Schema. It is realistic to foresee 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 then 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.

**3.4.1 The Schema Manager**

The Schema Manager is in charge of defining, refining, and modifying classes. It incorporates a class Browser and Editor, and uses the class-level procedures `class_create`, `class_change`, and `instance_convert`.

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 Schema 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/triggers or redefinition of existing ones;

• redefinition of values of pre-defined class-level attributes;

• definition of new class-level procedures/triggers or redefinition of existing ones.

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

Procedure `class_change` attempts to update procedures/triggers 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.

As the Planner and the Execution Manager ...

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_delete` attempts to delete a class. A class having either related classes or instances cannot be deleted. The procedure `instance_convert` converts `Self0` to a new class identified by `New_Class`. Procedures `restart` and `stop` are heavily used by the Schema Manager for stopping, converting, and restarting instances of evolving classes.
3.5 Implementation

Procedure class\_change

Procedure class\_change operates a single change on a \texttt{PM\_entity} subclass, as described before. 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 very profound 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.

Figure ?? 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 \texttt{SelfC}\(^7\), the state is read. If the \texttt{task\_state} is \texttt{waiting} or \texttt{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

The instance-level procedure \texttt{restart} suspends the work done by a task

\(^7\text{Self0 and SelfC are used to denote the instance or class a procedure is invoked on.}\)
class_change(Sender,SelfC,pre_dynamic-X):-
call_proc(SelfC,SelfC,read_relation,instance_of,I),
call_proc(SelfC,I,read,task_state,S),
(member(S,[waiting,created]) ->
  call_proc(SelfC,Sender,notify,
    ["task",I,"in state","not affected"]);
  member(S,[active,waiting_children,forked,terminated]) ->
    (eval(X) ->
      call_proc(SelfC,Sender,notify,
        ["task",I,"in state","not affected"]);
      call_proc(SelfC,I,restart,[],-),
      call_proc(SelfC,Sender,notify,
          ["task",I,"in state","restarted"])))
fail;
call_proc(SelfC,SelfC,write,pre_dynamic,X),
call_proc(SelfC,Sender,notify,"work done").

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

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 system jobs, and to eliminate all the side-effects associated to code enactment. 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 enactment. Thus we assume that an instance-level procedure fail specifies actions to “undo” the results from the code part.
restart(Sender,Self0):-
call_proc(Self0,Self0,read,task_state,S),
    (member(S,[created,waiting])->
        true;
        S=active->
            (call_proc(Self0,Self0,read_relation,actual_outputs,Out),
             call_proc(Self0,Out,reset,[],_),
             fail;
             true);
        S=waiting_children->
            (call_proc(Self0,Self0,read_relation,sub_tasks,T),
             call_proc(Self0,T,restart,[],_),
             fail;
             call_proc(Self0,Self0,read_relation,actual_outputs,Out),
             call_proc(Self0,Out,reset,[],_),
             fail;
             true),
        S=forked->
            (call_proc(Self0,Self0,read,os_pid, Pid),
             kill_os(Pid),
             call_proc(Self0,Self0,read_relation,actual_outputs,Out),
             call_proc(Self0,Out,reset,[],_),
             fail;
             true),
        S=terminated->
            (call_proc(Self0,Self0,read_relation,actual_outputs,Out),
             call_proc(Self0,Out,reset,[],_),
             fail;
             true)),
call_proc(Self0,Self0,write,task_state,waiting).

Figure 9: Procedure restart.
The implementation of procedure **restart** is given in Figure ??.

Procedure **restart** reinitializes 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.

### 3.6 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) [?].

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 etc..

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 Figure ??, we depict a scenario where two parallel subprojects, **PMB** and **PMC**, co-exist under a (super)project **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**).

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.

Figure ?? 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 Figure ??, 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**
Figure 10: A scenario based on subclassing and versioning.
output. The user may or may not choose the suggested actions. Concerning resetting 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 created.

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 [?], IPSE 2.5 [?]). In addition, some large examples have been run by the development teams, and several PSEs can assist in maintaining themselves (ADELE [?], MARVEL [?]).

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 [?]. CLOS [?] also offers reflective features and pre-defines procedures to change class definitions, and to convert the affected instances. Similarly, SELF [?] 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 [?] offers only task-level reflection. Laws to control evolution of both product
and of the rules themselves may be defined DARWIN [?].

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 [?], Merlin [?] and IPSE 2.5 [?] can offer some formal verification support, and MELMAC [?] 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 [?]; 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 [?] enables to define meta-rules for changing the enaction state. This can be done also in EPOS. ARCADIA [?] 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 [?] 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 [?] enables to change enacting models, but not schemas.

PRISM [?] 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 [?]) towards product and/or project contexts for customization. EPOS has derived dynamic binding and class-properties from Smalltalk [?] 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. P-SAlgol used in IPSE 2.5, PCTE in ALF [?], O2 in SPADE, and LDL in OIKOS [?].

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 [?], and also by IPEM [?]. Other PSEs using goal-oriented AI techniques are GRAPPLE [?], 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

In EPOS, a template process model is a Process Schema with a set of classes and meta-classes, abstractly describing the external process elements and their relationships. An instantiated and enacting process model is a set of instances of the Schema classes. These instances form a chained and decomposed network of concrete descriptions of interacting activities (tasks), coupled to products, tools and roles, and other activities. The activities interact with each other and with performing tools and humans.

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 process 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 (re)generate task networks incrementally and dynamically.

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 are judged crucial for these test examples.

Of the general EPOS PSE drawbacks, we can mention:
• The EPOS PSE is not enough “open-ended” wrt. other platforms and application domains (e.g. distribution and federation aspects).

• A proprietary SPELL language. That is, could some other (reflective and concurrent) Prolog, Lisp, or imperative language have done the job? Like most PSEs, we have made our own interpretative PML, although the implementation effort is only a few thousand lines of extensions to SWI-Prolog.

• COV is a promising but unproven versioning technology.

Of the specific drawbacks wrt. process model evolution we can mention:

• Poor support for assessment, simulation, and validation of process model changes;

• No overall methodology for maintaining (and developing) process models.

• How to use versioning (and COV) properly on such changes?

• Too much focus on (rather trivial) Schema changes, while ignoring more difficult changes, e.g. in the architecture or platform of the external process?

Some ideas of future work, aside from rectifying some of the above drawbacks:

• A more high-level PML, supported by a development method and embedded in a small CASE tool for PM – e.g. an expanded Schema Manager?
- Better understanding of the relation between meta-processes and production processes.

- Proper assessment of process performance vs. process model enactment.

- Gaining more experience with practical applications ...

- Developing a better conceptual framework for (software) process modeling.