

Variability of Process Models, and the EPOS Solution

Reidar Conradi*

M. Letizia Jaccheri

Cristina Mazzi

Dept. Computer Systems
Norw. Inst. of Technology
N-7034 Trondheim, Norway

Dipart. di Automatica ed Informatica
Politecnico di Torino
I-10129 Torino, Italia

Dipart. di Informatica
Università di Pisa
I-56100 Pisa, Italia

1 Introduction

Most PM systems are intended to serve in a hybrid machine-human environment [1]. Thus, *variability* through customization and evolution is crucial. As for interactive and knowledge-based systems, many PM systems have utilized interpretation and reflection, often combined with object-orientation. However, there is meagre methodological support for controlling variability e.g. of a Process Schema and its instances.

The paper will first define the term *process model*. Then follows a list of demands and existing technologies for process model variability. We also characterize some existing PM systems wrt. to such variability. Thereafter follows a short presentation of the EPOS support for PM variability, and some indications of further work.

The EPOS approach to PM variability consists of: 1) A common, reflexive, object-oriented data-model, available through the SPELL process specification language and offering types to describe the software process world; 2) an AI Planner to incrementally (re)generate task networks; 3) an underlying EPOSDB versioned database that stores both the Process Schema and its model instances.

2 Background

A (software) process consists of two main components: a *production process* to carry out production activities, and a *meta-process* to improve and evolve the whole process. An *internal* (computerised) *process model* provides a representation of *external* (real-world) process entities and their relationships. It can be similarly divided in a production process model and

a meta-process model (meta-model). A part of the model is called a *model fragment*.

The process entities may be of the following main categories: activities, products, tools, roles, and organization. Often a process model encompasses a *template model*, a Process Schema with types or rules in an enactable Process Modeling Language (PML). Such a Schema is used to describe, create and manipulate *enactable* and *enacting model* fragments (see below). The Schema and other parts of the meta-model may be explicitly manipulatable, allowing *reflection*.

We can identify the following variability issues in software process models:

1. Categories of variability:

- *Customization* according to project context (“variants”).
- *Evolution*: corrections and improvements (“revisions”).

We can also mention *Refinement* linked to specialization and life-cycle phases.

2. When can process models and their support be changed?, cf. the following PM life-cycle phases or meta-processes:

- (a) PM0. Provide Process Support: e.g. PML, pre-defined models, process tools.
- (b) PM1. Elicit reqmts for process model.
- (c) PM2. Analyse/design a *template* model (“symbolic program”).
- (d) PM3. Implement an *enactable* model (“load image”).
- (e) PM4. Executing an *enacting* model (“running image”).
- (f) PM5. Assess process performance.

3. Reasons for change: perfective, adaptive, or corrective – and their frequency and costs.

*Detailed address: Dept. of Computer Systems and Telematics, Norwegian Institute of Technology (NTH), N-7034 Trondheim, Norway. Phone: +47 7 593444, Fax: +47 7 594466, Email: conradi@idt.unit.no.

4. **Frequency of change:** Is it every minute, day or year – and for which model fragments?
5. **What can be changed?:** i.e. the variability axes and their **process parameters**?
 - *Process Support* (PM0): process tools and their platforms.
 - *Meta-model* fragments (PM2): e.g. policies for model changes.
 - *Template* model fragments (PM2): subschemas for activities, products etc.
 - *Enactable* model fragments (PM3): descriptions of unstarted activities, products and their dependencies, or tools (scripts and executables with switches).
 - *Enacting* model fragments (PM4): descriptions of running activities.

And how many of the changes can be *anticipated* (and thus “parameterizable”), and how many are *ad-hoc* (“raw” versioning)?

6. **Mechanisms** to support model fragment variability:
 - A *flexible architecture* to substitute system parts, cf. reconfigurable distributed systems.
 - *Dynamic binding*: cf. Lisp, Smalltalk, and Prolog – cf. reflection below.
 - *Restricting access rights*: Modularization, interfaces, scoping.
 - *Reflection* with explicit types and meta-types, so that we can reason upon, manipulate, and dynamically interpret a Schema.
 - *Variable types* (Schema):
 - versioned types (*between* transactions): what are the consequences for their instances; must we reformat them or temporarily passivate activities?
 - subtyping (*within* a transaction),
 - views (cf. versioned types),
 - generic/parameterized types,
 - instance-level delegation.
7. **Methodologies and principles to guide changes.**

We need meta-level rules e.g. to observe consistency constraints and apply impact analysis, supplemented by user views and roles.

The structuring and modularization facilities of the PML and the model fragments will influence the possibilities and mechanisms for variability. In PM systems with dynamic binding, the phase distinction between

design of a template model (PM2), and an enactable or enacting model (PM3 or PM4), is blurred.

Table 1 summarizes *three* variability aspects of PM systems: Schema level, Task Network level, and underlying mechanisms/methodologies. Given the heterogeneity in the above solutions, we should base our PM support on a few general concepts, expressed in a unified formalism. Then the same mechanisms and methodology for PM variability (subtyping, versioning, dynamic binding etc.) can be applied to most model fragments: product and activity descriptions, templates (types), and partly to the meta-Schema.

3 Variability in EPOS PM

There are four axes of structuring and variability in EPOS Process Schemas:

1. *Vertical* structuring by subtyping and other project specialization; i.e. the activity rule base is *hierarchical*.
2. *Horizontal* evolution inside projects and versioning between projects.
3. *Combined* vertical/horizontal, e.g. by dynamic binding semantics.
4. *Reflection*, e.g. by type-level operations and constraints.

EPOS stores the *entire* process model in a versioned software engineering database, EPOSDB. This has a structurally object-oriented data model and implements *Change Oriented Versioning* (COV). COV enables uniform versioning of entities and relationships, and their types¹. Nested and cooperative transactions are executed on the current database *version*.

The **SPELL** [2] process specification language combines and extends the DDL and DML of the underlying EPOSDB. SPELL supports user-accessible types and meta-types (reflection), type-level attributes, type- and instance-level procedures, triggers on procedures, and concurrency (tasking). The set of declared procedures and triggers in a type is open-ended, and with dynamic binding as in Smalltalk. Attributes and procedures can be **Private** or **Public**, giving some degree of modularisation. The type-level task attributes serve as shared *process parameters*, and can be redefined by subtyping. Most of the type-level procedures

¹NB: from a representation point of view, see Section 3 on change semantics.

An EPOSDB type corresponds to a *class* in the OMG terminology, and has an extent.

and -attributes instrument the Schema Manager, Execution Manager, and Planner (see below). Many procedures concern creation, update and deletion of types, as well as creation and evolution of instances.

A task type expresses an *activation rule* by the type-level attributes PRE_DYNAMIC, CODE, and POST_DYNAMIC. Activities (tasks) are connected in a *task network*, chained by products serving as actual parameters and decomposed into subtasks. Task execution is performed by the **Execution Manager** as a Process Engine (meta-process PM4), interpreting the above activation rules. Incremental instantiation and decomposition of networked tasks is done by the **Planner** (PM3), interpreting the type-level attributes PRE_STATIC, POST_STATIC, FORMALS, and DECOMPOSITION. It is also able to *replan* after product and type changes.

The **EPOS Schema Manager** incorporates a SPELL Editor to browse, edit, and analyse the types, and their customization and evolution (PM2). A SPELL Interpreter implements the access of procedures, triggers, and type-level attributes.

Types are stored as **TypeDescrs**, with meta-types. Type-changes can be **hard** (but simple!), involving changes in instance-level attributes or the subtype hierarchy. Hard changes are p.t. disallowed, cf. schema evolution in databases. However, type-changes can be **soft** or behavioral (and complex!). These are the PM-specific ones, and can be implemented in SPELL on top of the database. The type-level procedures **t_create** and **t_change** are used to introduce and change types, and these procedures can be subtyped. The feasibility of a requested type-change must be evaluated against its possible *impact* on the whole Process Schema and its instances. A list of *constraints* will be consulted. If the impact is too big, a new *subtype* might be defined, so that only a *subset* of the extent of the old type will be affected. E.g. we do not allow changing the CODE script of an executing task, but the PRE_DYNAMIC might be changed.

3.1 Projects and Their Sub-databases: A small Scenario

We will here describe the overall infrastructure of the EPOS PM support tasks, and omit roles and access rights (locking).

A long EPOSDB transaction is associated to a **Project** task. Its most important subtasks are the **Schema Manager** (meta-process PM2), a **Project Manager** (in PM3) to start and finish child projects, a **Coordination Manager** (mostly in PM3-PM4) with a Workspace Manager to coordinate with pos-

sibly overlapping sibling tasks and a **Develop** task (PM4) 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 type descriptions of local activities, products, production tools, and roles. Such schema evolution can also be done incrementally later, if the meta-model in the Schema Manager allows this. Thus, several parallel subprojects can co-exist and have customized Process Schemas, perhaps initially “borrowed” from a common superproject. This means that the *same*, shared products can be governed by *several* customized or versioned Process Schemas (views), belonging to different subprojects.

We then use the **Cooperation Manager** to establish cooperation protocols (negotiation and propagation rules and patterns) against possible overlapping neighbour projects/transactions. Such protocols can specify tight cooperation (automatic propagation), loose (manual propagation), or delayed until commit. Thereafter we check-out the relevant workspace files from database “long fields”.

Then **Develop** can start, and its subtasks can be gradually (re)planned and (re)executed – respectively in meta-process PM3 and PM4. This process depends on the actual production activities (PS changes) and meta-process activities (type changes). In parallel, we may have communication with neighbour projects, and merging may be required of exchanged and revised product parts. We can also start and finish new subprojects *manually* via the Project Manager.

Finally, the modified workspace files are checked-in to the database. The local project is closed by committing the database transaction, and possible remaining update conflicts resolved. The **Project Manager** of the parent project is notified.

3.2 Future work

- Better control of access rights, coupled to roles.
- Guidelines, as part of a methodology for PM variability.
- Extending the Schema Manager into a CASE Tool for PM?

Survey of Variability in some PM systems			
Name of PM system	Schema variability (PM2)	Task Network variability (PM3-PM4)	Methodology of Change (constraints, guidelines)
ADELE2	Evolving triggers and product types, dynamic binding	Implicit activities through triggers	Some guidelines, no meta-model
ALF	Customizable subschema in PCTE, late binding	Activity network is static	Customized parms, few guidelines
ARCADIA	Changed APPL/A programs	not after compilation	None? (not documented)
DARWIN	Static types, but rules may be added	Dynamic task network must obey Laws	The Law is mutable & self-regulatory
EPOS	Subtyping, "soft" type-changes, dynamic binding	Incremental (re)planning of task network	Type-level procs and constraints, versioned EPOSDB
HFSP	No type definitions	Task network can be changed	Meta-operations to change process state
IPSE 2.5	Evolving types, dynamic binding	Evolving network of tasks/roles	Type reflection, PM "Cookbook"
MARVEL	Activity rules can be customized	Via changed rules	Constraints on rule/type evolution
MELMAC	Static rules	Petri Net can be gradually refined	No meta-rules, manual net construction
MERLIN	Rule clusters can be customized	Via changed rules	
OIKOS	Static "theories" in Prolog, sub-blackboards (C++ programs)	Dynamic blackboards and their agents	None? (not documented)
Process WEAVER		Task network can be manually changed	Reconfigurable tasks, few guidelines
SPADE	Static types in DBMS (O2), may add new ones	Petri Net dynamically (re)reconstructed	By reflexive Petri Net

Figure 1: Variability in some PM systems.

- Considering changes also in process architecture/platform.
- And: getting more experience with practical applications.

References

- [1] Reidar Conradi, Christer Fernström, Alfonso Fuggetta, and Robert 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*, pages 3-17, September 1992.
- [2] Reidar Conradi, M. Letizia Jaccheri, Cristina Mazzi, Amund Aarsten, and Minh Ngoc 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*, pages 167-177, September 1992.