Software Process Modeling and Evolution in EPOS

Letizia Jaccheri[*], Jens-Otto Larsen, Reidar Conradi.

Div. of Computer Systems and Telematics (DCST)
Norwegian Institute of Technology (NTH),
N-7034 Trondheim, Norway

Abstract

Software process models are meant to describe software engineering activities around evolving software products. Process models will need modifications due to gained experience, changed policies or project-specific requirements, and a set of versions of the original models must be maintained. Thus, software process models are evolving products and should be under control of a configuration management system.

EPOS models Software Processes in an object-oriented data model. Both Process models and running instances are stored in a versioned database. EPOS Software Process Models may be evolved and reused by the same CM techniques as used for software systems.

1 Introduction

Configuration Management (CM) is the discipline of managing the evolution of complex systems. Software CM is needed to assist in designing and engineering software products. CM assumes a shared and versioned repository to store *software products*, consisting of general software components and their dependencies.

A software process is the total set of software engineering activities needed to produce and evolve software systems. Software *Process Modeling* (PM) consists of defining models that specify how some classes of software processes should, could, or have been performed [DNR91].

PM and CM are strongly related: PM depends on and is driven by CM, e.g. work breakdown and activity chains are deducible from the product structure. On the other hand, products are evolved as a result of PM activities. The PM information itself consists of structured and evolving objects, i.e. belonging to the CM domain.

The emerging paradigms in the PM community are essentially three: Rule based, Graph/net based [DG90], and Process programming [TBC$^+$88]. AI techniques found in *rule-based systems* have been applied to software process modeling, e.g. planning in [KFP88] or central blackboard structure in [ACM90]. The *Graph/net* approach adapts and extends techniques for real time systems specification, such as Petri Nets, to model software process, particularly w.r.t. dynamic triggering and concurrency. *Process programming* means to define the process by an executable specification in an *process programming language.*

EPOS [COWL91] is a multi-user software engineering environment kernel covering both CM and PM. EPOS uses rule-based techniques in the process models, Graph/net for enactment, and process programming to express the basic activity steps. The core of EPOS is the client-server based database system, EPOSDB. The database server implements an object-oriented data model, versioning and a long transaction mechanism. EPOSDB offers programming interfaces to C, C++ and Prolog. The EPOS PM system is an integrated set of tools covering the phases of the process life-cycle.

This paper outlines the EPOS process model life-cycle and the EPOS tools covering the different steps. We proceed by describing the EPOS software engineering database and the EPOS data model. We show how the process models and the software processes are described in the EPOS data model and how a process model can be evolved by the data modeling and versioning mechanisms offered by the EPOS database.

2 EPOS Software Process Life-Cycle

The software process meta-cycle, also called software process life cycle, consists of the phases of design-

[*]Current Address: Politecnico di Torino, Corso Duca degli Abruzzi n.24, 10129 Torino, Italy.

ing, tailoring and evolving process models and generating software processes from process models. This section will give an overview of the EPOS software process life-cycle following the terminology suggested in [FH91] and emphasizing the connections to the EPOSDB. Figure 1 illustrates the software process life-cycle for EPOS.

The EPOS *Process Architecture*, or framework, consists of the object-oriented data model, EPOS-OOER, the EPOS database with its versioning mechanism, and a set of PM tools. The *Process Elements* that constitute a Process Models are defined within this framework.

An EPOS Process Model consists of several sub-models: an *activity model* describing the tasks carried out by humans or tools, a *product model* for structuring software systems, an *organization model* for representing the organization of humans into teams and a *tool model* to describe the engineering tools at hand. These sub-models are inter-related to represent generic connections between process elements.

To describe these sub-models, EPOS uses different kinds of process elements: *activities* (tasks), *products tools* and *resources*, and subtypes of these. Each group forms the root of a specialization hierarchy of type definitions in EPOS-OOER. Both the types and the process instances are stored in the database. The versioning mechanism aids the management of the total set of process elements.

The EPOS *Process Design* (or Process Model) consists of a structure of process elements in addition to those offered by the general data model. The *Process Design* phase in EPOS consists of designing a consistent set of process elements and storing them in the database. Process elements are defined in the EPOS *Process Modeling Language* (PML) and placed in the correct element category by subtyping. In this way, process elements will be refinements of the existing framework. The design process resembles schema design for ordinary database applications and the resulting set of process elements constitutes a database schema – a valid configuration of EPOS-OOER types.

The PML translator takes as input textual PML schemas and translates them into the database internal representation. The PML translator performs syntactic and static semantics checks. It is written in Prolog and it is heavily based on the EPOS database Prolog interface. The PML translator works in an incremental mode.

A generic process model may be specialized into a more refined or project-specific one, also called a *Process Definition*. An EPOS *Process Definition* consists of a set of process elements intended for use in a specific project.

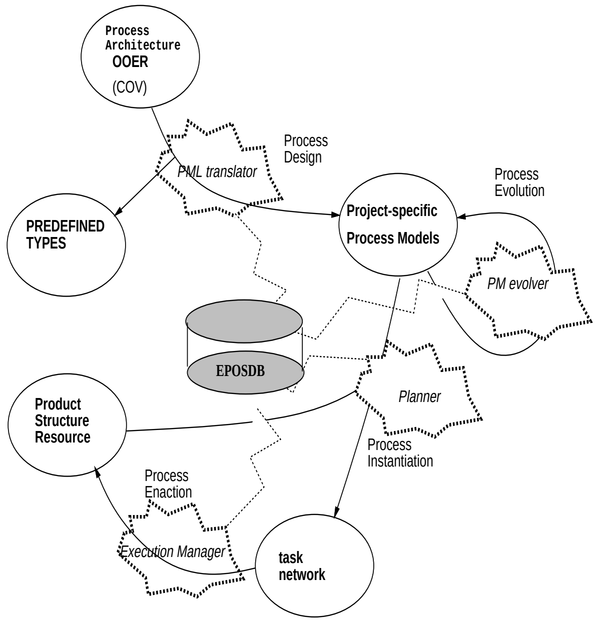

Figure 1: EPOS Software Process Life-cycle

A process definition may be further refined by subtyping or changed within the context of a given project. *Process Tailoring* is the act of obtaining (sub-)project-specific process definitions from more general or system-defined ones. *Process Evolution* concerns the process of detailing and correcting existing process definitions. Both these activities result in modified process definitions. The EPOSDB versioning mechanism is used to distinguish the modified process definitions from the original ones.

The PML translator works in collaboration with the EPOS *PM Evolver* that is in charge of refining and changing process models. EPOS also provides support for *Dynamic Evolution*, i.e. evolving process models with active processes. Facilities for automatic instance conversion are provided. The *Evolver* is also written in Prolog on top of the database interface.

Software Processes (tasks) are modeled as instances of the types representing the process elements. *Process Instantiation* depends on project-specific matters as project goals and policies; product structure; organization of the development team; project schedules and availability of human, tool and other resources. Process instantiation is done automatically by the EPOS Planner. The Planner takes as input both project-specific process definitions and the actual product description to produce executable task networks. During instantiation, the EPOS Planner generates a network of task and product instances and stores it in the EPOSDB. Specific conditioning for triggering and terminating activities are set as well as parallelization and serialization of activities.

A task network is *executed* by the EPOS *Execution Manager* (EM), which interacts with the EPOS Planner. The instantiation phase is highly dynamic and it is performed on demand in an incremental manner. The binding between process models and the respective software process is done at run time.

3 EPOS Framework

The integration framework in EPOS is the EPOS data model, common to PM and CM. The EPOS database provides persistent storage for instances of the data model. The versioning model is the base of EPOS CM and provides uniform versioning of both software systems, software processes and models of these.

3.1 EPOS data model

The data model of the EPOS database, EPOS-OOER is a variant of the Entity-Relationship model [Che76], extended with object-oriented concepts: a strong notion of object identity, subtyping and complex objects.

Entities are objects with a unique identity. Entities are created as instances of an *entity type* which is defines a set of *attributes* for its instances and a set of *supertypes* from which attribute definitions are inherited. The instances of an entity type will also be instances of the supertypes. Entities can be converted to sub- or supertypes of its original type while still preserving their object identity.

Relations represent associations between entities – the *roles* of a relation. All kinds of inter- and intra-object associations are declared in the object model through relation types. EPOS-OOER only allows binary relations. Relationships can have attributes. Relationships are created as instances of a *relation type*. Relation types have the same subtyping semantics as entity types and in addition the role specification is inherited. A relation subtype may constrain the roles to subtypes of the entity types.

The entity and relation types are represented in the database by *type-descriptor* objects which store the type name and a structural description of the instances. In their basic form, type-descriptor objects are instances of the predefined *meta-types* `enttd` or `reltd`. Users may create subtypes of these meta-types and in this way add more information to the type-descriptor objects. This mechanism can be utilized to store attributes and relationships common to all instances of a type. Type-level attributes may be versioned just as ordinary objects.

The EPOSDB allows dynamic extensions of the type system, i.e. new subtypes of the existing type hierarchy can be added to the database at any time.

3.2 Versioning

The EPOSDB implements the *Change-Oriented Versioning* model (COV) [LCD$^+$89]COV is independent of the data model and enables uniform versioning of entities and relationships.

In the following, a *logical change* is a named set of *physical changes* that result in a product with distinct properties. A physical change is a sequence of database updates. Two important properties of COV are:

Uniform version selection: This is done by including or excluding logical changes and results in a uni-version view of the database. The version selection is done independently of object access and results in a set of consistent configurations from which the product selection can be made.

Variable scope of changes:
A physical change may be a part of several logical changes. COV lets the user specify the range of a physical change in terms of logical changes. When later version selections are made, a physical change is included if the version selection is within the range of the change. The range can be specified so one physical change may be visible in several later version selections.

3.3 Transactions

EPOSDB offers a long transaction model with nested and non-serializeable transactions. Transactions may survive several application sessions. The transactions are user-controlled - they may be started, committed or aborted interactively. All database operations must be performed within the context of a long transaction.

Transactions also provide a framework for versioning. A transaction represents a consistent set of physical changes to database as defined in the previous section. All changes in a transaction have the same version range.

4 Software Process Modeling in EPOS

Our goal has been a system interpretable formalism to describe the process models and the related software activities. Activities and products are described as types in an extension of the basic EPOS-OOER data model. The extended model allows definition of behavioral properties, *procedures*, in addition to the attribute definitions. The same subtyping semantics as described in section 3.1 applies to the extended model.

The activity model consists of a set of *task types* and the product model of *data types*. Two predefined types, `taskentity` and `dataentity`, form the roots of two type hierarchies, the task type hierarchy and the product type hierarchy.

Figure 2 shows a part of the EPOS type hierarchy. Types are represented by rectangles, ordinary instances are represented by circles, arrowed lines represent the subtype links. The names in brackets under each object denote the type of the respective object.

EPOS models the following properties with a single task type:

Pre- and postconditions: are formulas in first-order predicate logic. The precondition is split into two parts: *Static precondition* is used at task instantiation time for backward and forward reasoning without executing the task. The *dynamic precondition* is tested by the Execution Manager at execution time and it is used for dynamic triggering of tasks. (similarly, for postcondition).

Code: This part of a task description defines the steps that are performed when the task is executed. A tasks' code is responsible for causing its Postconditions to become true. The code is written in an appropriate language, e.g. Prolog is used in the EPOS prototype.

Decomposition: specifies type-level task aggregates to constrain the *vertical* `SubTasks` relationships. A task type has *decomposition* relationships to the set of task types the Planner can choose among, when decomposing instances of the given type. Task instances have `SubTasks` relationships to its sub-tasks in the task network.

Formals: The formal input and output parameters of a task are described by relationships from task types to product model types. `Formals` defines the types of the inputs and outputs of the task execution. Task instances are related to actual product instances.

Software Process steps, or tasks, are modeled as instances of task types. All tasks have an instance level attribute denoting their execution state..The task state may be manipulated by ad hoc rules such as `start`, and `stop`. The individual task behaves according to the definition in its task type.

4.1 Process Model Representation

Task types are modeled as special EPOS-OOER types and tasks are created as instances of these types. To accommodate the storage of the task type properties, special type-descriptor objects are created as described in section 3.1. The meta-types `tasktd` and `datatd` defines the necessary attributes and relations to store the type properties for activity model and product model types. The pre- and postconditions and the code are stored as attributes in the type-descriptor objects representing task types. Decomposition and input/output are defined as relation types between these meta-types and the actual relationships will be created between type-descriptor objects when they are defined.

Procedures are divided into two groups: instance level and type level procedures. Examples of type level procedures are creation of new types by specialization of existing ones, change of existing types by versioning, and instance creation. Procedures are written in Prolog and stored as attributes in the type-descriptor objects, but interpreted at the PM level.

Some of the task type properties have special inheritance semantics, e.g. the static precondition are concatenated by conjunction with the static precondition of the supertypes (the same applies for all kinds of pre- and postcondition, both static and dynamic). Pre- and postcondition for the topmost `TaskEntity` type all have the value `true`. The code part have execution semantics as defined for Simula [DMN70].

An inherited procedure body may be redefined, but the interface has to be kept stable. The search and binding mechanism is implemented by the message passing mechanism `call_proc (Caller, Receiver, ProcName, InputPar, Results)`.

4.2 Software Processes Instantiation and Execution

`Project` is a special task type that emphasizes the productive aspect that result in a set of database logical changes. A project runs in the context of a database transaction. Changes performed in a project are kept local until the corresponding transaction is committed, thus allowing type changes and relative

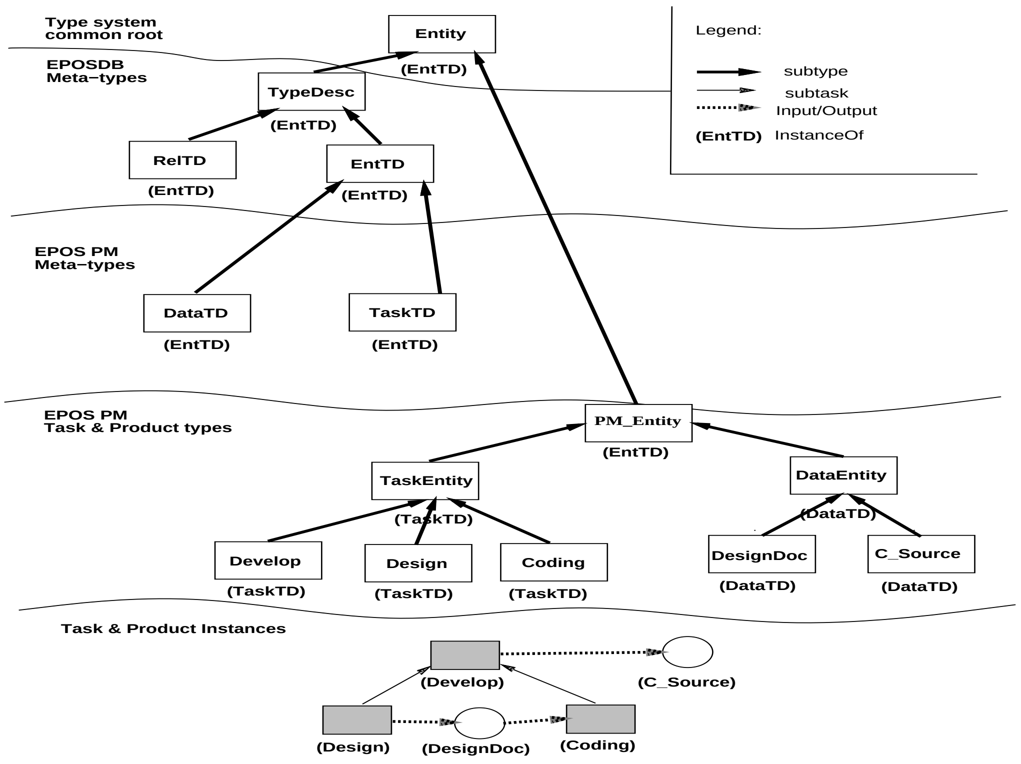

Figure 2: Meta-types, Types, and Instances

customization local to a project. A project may consist of subprojects. Project breakdown (decomposition) is modeled by transaction hierarchies. Projects provide contexts for executing software processes.

The EPOS *Execution Manager* (EM) and *Planner* are working in close cooperation. The EM schedules, interprets, executes, controls and records tasks. It cooperates with external tools and users. The EM is responsible for invoking the Planner, and to execute and re-execute the plans generated by this.

The Planner works on a knowledge base consisting of type-descriptors and a Product Structure as its *World State Description*. It offers *product-level* assistance such as construction of a derivation graph, and *project-level* assistance about job decomposition, and work agendas. The generated plan is handed over to the EM for later execution. Then, it offers replan upon Product Structure changes and/or execution failures, i.e. unexpected plan results. Finally, it provides learning by putting generated plans back into the Project database.

The reasoning of the Planner is based on the static pre- and postconditions of task types. It applies *hierarchical* and *non-linear* planning. Hierarchical planning is accomplished by coupling the EM and the Planner. That is, when the EM meets a task with empty *code*, it calls the Planner to decompose this "high-level" parent task based on its decomposition relationships. The Planner will take the current world state as the initial world state, the postcondition of the parent task as the goal, and the tasks set in the parent's decomposition as the candidate subtask pool. It will then build a subplan to achieve the postcondition, and add it to the original plan through the parent's SubTasks relationship. The generated plan will be a partially ordered task network, not a linear sequence of tasks. Thus the Planner can deal with parallel processing and handle possible goal interactions.

5 Software Process Models Evolution

Process Models evolve as experience accumulates and as the environment being modeled changes. Process models may need to be customized to project specific requirements. Software Process enactment may reveal errors and insufficiencies in the models. Then, Software Processes may change during their enactment, either adding detail not prescribed by the model (refinement) or modifying (and possibly creating) detail to accommodate unpredictable contingencies. Finally, Software Processes may be adapted to obey to modified process models.

Planned process evolution, or Process Tailoring is the act of obtaining project specific process definitions, from generic process designs. The bulk of the tailoring work usually happens before process instantiation.

Both Process Models and their instances become evolving products: changing process models means changing type definitions, changing software processes means either operating at the instance level or reflecting into the instance level changes occurred at the type level.

5.1 Subtyping and Versioning

Subtyping allows us to specialize type definitions in an incremental manner. This enables us to start from a generic model and to add knowledge about the process being modeled in a controlled way.

Versioning enables the user to change each represented object on the fly retaining the history of changes and representing alternatives. Versioning is a flexible and powerful basic mechanism, but more high-level control mechanisms are needed in order to know, control, and forbid certain changes. Configuration Management (CM) includes maintaining consistency among the different parts of an evolving system.

The challenge of system evolution does not rely in the technical change mechanisms. The system must know the semantics of the changed objects as well as the semantics of the changes in order to forbid certain changes, and manage the allowed ones. EPOS includes a set of rules that handle evolution. EPOS exploits *reflection* because types (and even meta-types) are objects themselves that can be manipulated. Both the enaction status and the definition of each activity may be operated on.

5.2 The Evolver

The *Evolver* manages type evolution. This is accomplished by rules for changing and creating types. These rules are specified as type-level procedures along with the type definitions.

We have defined the type-level procedures `t-create`, `t-delete`, and `t-change` together with `TaskEntity` type and they may be redefined for `TaskEntity` subtypes. Each type is created by specialization of its supertype. Procedure `t-create` creates a sub-type of the type it is associated with, by redefining type level attributes and adding/redefining procedures. We do not foresee structural changes for active type like deletion or addition of instance level attributes, i.e. we believe the structure of the instances to be stable.

Obsolete types may be deleted (rule `t-delete`), but of course attention has to be paid not to create inconsistencies. The rule `t-change` updates a type definition. In the context of this rule, we make an attempt to check the feasibility of the existing instances conversion invoking the meta-rule, `i-convert`.

The rule `t-change` defines legal changes to a type. The changes are kept local to the embedding project. If a project, or sub-project, is intended to produce durable changes to the database, either product, process or process model, the changes will usually be represented by a new, named logical change. When the project finishes and its changes are released, either to the parent project or to the global database, these changes can be retrieved by including the logical change in the version selection.

Instances may be converted downward in the type hierarchy. After a subtype has been created, the existing instances of its supertype may be selectively converted to be instances of the newly created type. On the other hand if a type has been modified, all the visible instances of that type are automatically bound to the changed type.

The `TaskEntity` predefined type attributes may be divided into two groups: dynamic and static. The dynamic properties, i.e. dynamic pre- and postcondition, and code are read and executed at run time by the run time support. The static properties act as constraints at instantiation time of a task network. Changing the static type properties means changing the constraints that regulated the shape of an existing network.

The rule `i-convert` attempts to convert an active task to a modified type definition, see CLOS' `update_instance_for_redefined_class` [Kee89]. A task must be treated with regard to its state. The core of rule `i-convert` does not depend on whether the type change has been done by subtyping or by versioning. In fact we believe that the most frequent changes involve redefinition of the predefined type level properties. Then rule `i-convert` selectively considers the redefinition of such properties.

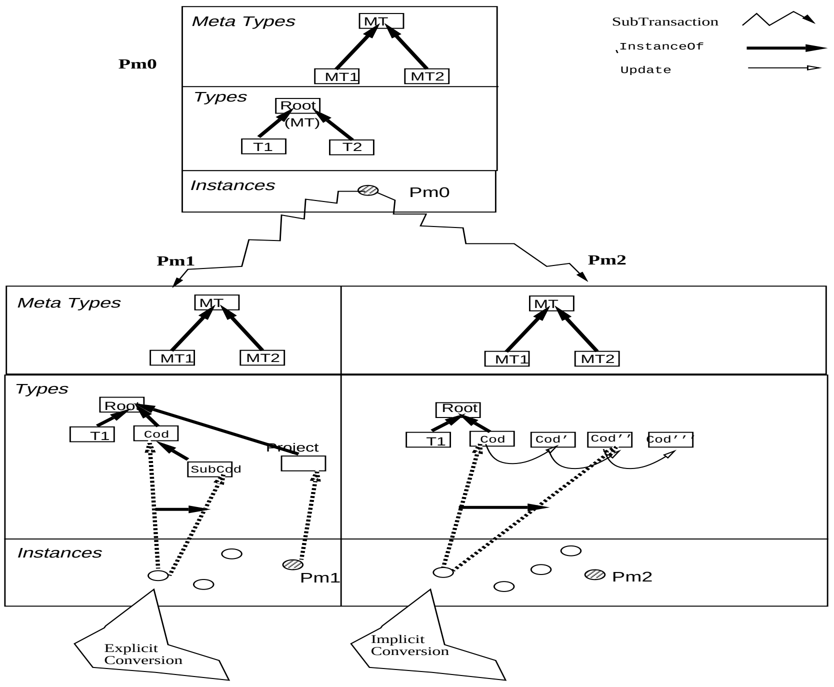

Figure 3: A scenario based on sub-typing and versioning

5.3 An Example

As an example we consider the the solution of the example problem as proposed for the 7th International Software Process Workshop.

Let us suppose to have a process description of a `Coding` task, stating that it is possible to begin coding before design is approved. Suppose that it is decided to change the process and to tighten the requirements detailing when coding can begin in order to require that the design be approved before coding may begin. Further, assume that this process change is to affect one (only one) currently "executing" or "instantiated" process.

In the EPOS framework, this means that we have to modify the dynamic precondition of the `Coding` type. We show how EPOS can solve the problem both by subtyping and by versioning.

In figure 3 we depict a scenario where two projects, Pm_1 and Pm_2, run as subprojects of a project Pm_0. The type *Coding* (Cod in the picture) has been defined in the context of the project Pm_0. In project Pm_1 the change is performed by subtyping, in project Pm_2 by versioning.

A set of types (T_1, T_2, Coding) has been created as instances of the respective meta-types in the context of a Pm_0 project, bound to a database transaction t_0. Then, at a later time, a project Pm_1 is created as a subproject of Pm_0. Thus, Pm_1 is bound to a subtransaction of t_0. The same set of objects (meta-types, types, and instances) is available to Pm_1.

In the context of Pm_1, it is possible to refine (*customize*) the knowledge inherited from Pm_0, by subtyping the type definitions, still retaining the types as defined in the super project. On the other hand, in project Pm_2, also created as a subproject of Pm_0, the type definition is changed, and when the subproject finishes a distinct set of changed objects is available to the super transaction.

Finally, concerning the instances, when the subtype is created as specialization of type `Coding`, the existing instance may, on demand, be explicitly convert ed to be an instance of the subtype. On the other hand, when T_2 is updated in the context of project Pm_2, the existing instance is automatically converted to be an instance of the modified type.

The rule `i-convert` selectively considers the task state. If task state is `Initiated`, then the instance is convertible. Otherwise the new dynamic precondition has to be re-evaluated.

6 Conclusions

The originality of the EPOS solution is that we are merging CM techniques with AI techniques to manage both software processes and software products. The database has been designed to provide the right granularity levels for both PM and CM. Software definitions, software processes, and software products are designed, evolved, and instantiated on top of a versioned database that enables evolution of processes as well as products. Static reasoning and active triggering occur on top of the database.

The EPOS data model is the basis for integration: The EPOSDB provides persistence for instances of the data model, the CM mechanisms work on instances of the model and PM models processes as types and instances. As a result, process model and process evolution can be managed by the CM system. The ability to dynamically add EPOS-OOER types to the EPOSDB allows flexible extension of process models. A changed process model can be stored as a distinct configuration, a version of the original process model, thus facilitating reuse of process models.

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, including the interface to Prolog. The entire PM framework including the is implemented on top of the multi-user EPOSDB by 7000 SWI-Prolog lines. The User Interface uses the PCE Prolog-based graphical package.

EPOS is now being used for medium-scale experiments. The challenging issue to be pursued is how to cope with design, structuring and evolution of real-world PM models.

References

[ACM90] 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*, pages 183–192, 1990.

[Che76] P. P.-S. Chen. The Entity-Relationship Model — Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[COWL91] Reidar Conradi, Espen Osjord, Per H. Westby, and Chunnian Liu. Initial Software Process Management in EPOS. *Software Engineering Journal (Special Issue on Software process and its support)*, 6(5):275–284, September 1991.

[DG90] Wolfgang Deiters and Volker Gruhn. Managing Software Processes in the Environment MELMAC. In *Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, California. In ACM SIGPLAN Notices, Dec. 1990*, pages 193–205, December 1990.

[DMN70] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. SIMULA Information — Common Base Language. Technical Report 145 p., S-22, Norwegian Computing Center, Oslo, 1970.

[DNR91] Mark Dowson, Brian Nejmeh, and William Riddle. Fundamental Software Process Concepts. In *[FCA91]*, pages 15–37, 1991.

[FCA91] Alfonso Fuggetta, Reidar Conradi, and Vincenzo Ambriola, editors. *Proceedings of the First European Workshop on Process Modeling (EWPM'91)*, CEFRIEL, Milano, Italy, 30–31 May 1991, 1991. Italian Society of Computer Science (AICA) Press.

[FH91] Peter H. Feiler and Watts Humphrey. Software process definitions, May 1991. (Draft document).

[Hen88] Peter B. Henderson, editor. *Proc. of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Boston), *257 p.*, November 1988. In ACM SIGPLAN Notices 24(2), Feb. 1989.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison Wesley, 1989. 266 p.

[KFP88] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, pages 40–49, May 1988.

[LCD+89] Anund Lie, Reidar Conradi, Tor M. Didriksen, Even-André Karlsson, Svein O. Hallsteinsen, and Per 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)*, pages 56–65, November 1989.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michael Young. Foundations for the Arcadia Environment Architecture. In *[Hen88]*, pages 1–13, November 1988.