Software Process Modelling in EPOS

Reidar Conradi[*]
Anund Lie, Espen Osjord, Per H. Westby,
Norwegian Institute of Technology, Trondheim, Norway
Vincenzo Ambriola, Univ. of Pisa and Udine, Italy
Maria Letizia Jaccheri, TecSiel, Pisa, Italy
Chunnian Liu, Beijing Polytechnic Univ., Beijing, P.R. China

Presented at CAISE'90, Kista, Stockholm, Sweden, May 8-10 1990.

Abstract

EPOS[1] is an *instrumentable, kernel* Software Engineering Environment (SEE). It consists of facilities for management of versioned **products** (configurations) through file-based workspaces attached to a versioned DBMS. EPOS will also manage the associated software development **processes** (tasks), being the subject of this paper.

The **EPOS-OOER** semantic data model can describe deriver tools, human actors, tasks and subtasks, projects, and triggers/notifiers; as well as normal software products. EPOS-OOER incorporates object-oriented ERA modelling, extended with tasking (PRE, POST, CODE) and simple type constructors (FORMALS, DECOMPOSITION). Customization is done through versioning of task types in project-specific workspaces.

Static task knowledge is expressed by types and subtyping, and is used for reasoning, planning, scheduling and execution of activities. Dynamic task knowledge is expressed by a *versioned* task network with a horizontal (temporal) and a vertical (decomposed) dimension. Tasks are connected to products by normal relationships.

Keywords: Object-Oriented ERA Model, Planning, Software Configuration Management, Software Process Management.

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

[1]EPOS, Expert System for Program and ("Og") System Development, is supported by the Royal Norwegian Council for Scientific and Industrial Research (NTNF) through grant ED0224.18457. Do *not* confuse with the German real-time environment of the same name [Lem86]!

1 Introduction

Experience with SEEs indicates that an *open* architecture is crucial to avoid strait-jacketing effects. Both new and old tools must be accommodated, and relevant company or project policies should be explicitly stated, enforced, reasoned about – and changed. This puts high demands on the expressivity and flexibility of the underlying formalism.

A software product is described by many interrelated and evolving software components. A software configuration management (CM) tool is therefore needed to control the evolution of such systems. Most CM tools are marginally aware of the underlying software processes and their management (PM). Such a development process is often described in terms of the product (*what*), while the process (manual or automatic) can specify *how* and *why* a version of a product is constructed. In other words: the CM and PM areas should be integrated. Much work has recently been spent on PM in order to understand, model, execute ("enact") and record the operations performed on a product. Such operations or processes range from simple tool invocations to high-level design and project-related activities performed by human actors. A perpetual argument in the PM area has been human creativity vs. automation [Leh87].

Several *generic* or meta-models for PM have been introduced. These can be *instantiated* or customized into a more *specific* process model, e.g. a waterfall, spiral or project-specific model. This puts high demands on the dynamics and generality of the underlying type system.

The EPOS PM approach is to integrate:

- Static process programming as in ARCADIA [TBC*88] and IPSE 2.5 [OR86].
- Dynamic (sub)contracts as in ISTAR [Dow87].
- Rule-based reasoning as in MARVEL [KF87] and ALF [B*89],
- Networked tasks with dynamic triggering à là OSMOSE (Petri net model) [DGS89], and to some extent PCMS [HM88] and NOMADE [BE87].
- Subtype refinement as in Process-Oriented CM [BL89].

Some more high-level structuring is needed, but we are only describing the kernel facilities and basic type system.

The ensuing sections of this paper are as follows. First the EPOS architecture and basic CM model of EPOS are summarized. Then the overall PM model and the associated Activity Manager are presented. Then follows a more formal treatment of our object-oriented ER model, EPOS-OOER, with emphasis on PM-relevant type properties, type constructors and project-specific versioning. Lastly, some present problems and ideas on future work are given.

2 EPOS Background

2.1 EPOS Architecture

The main EPOS components are:

- A semantic data model, *EPOS-OOER.*
- An advanced *CM system*, p.t. on top of the INGRES relational DBMS [SWKH76].
- An *Activity Manager* and *Planner* for PM support. The Activity Manager includes a *Builder.*
- A set of EPOS support tools, such as a *Product Editor* and a *Maintainer's Assistant.*
- A *User Interface*, based on the X Window System.
- *Local, checked-out workspaces* or configurations (files, databases in special formats), accessed by misc. *programming tools.*

EPOS will run on Unix workstations, using INGRES, X Windows, C++, and Prolog. See [CDrG*89] for more details.

2.2 Change-Oriented Versioning and Related CM

We have adopted the *change-oriented* model (COM) to versioning [Hol88] [LDC*89] [LCD*89]. Here, a *functional change* involving several (related) components is described by a single, *global option.* COM resembles and generalizes conditional compilation, and is fundamentally different from more conventional, *version-oriented* models (VOM). Most most database (DB) items can be **uniformly versioned** from a technical point of view. Some COM concepts are:

- *Option*: Essentially a boolean variable to describe a functional change (external property), such as `MachineSun` or `BugFixCommandA`. It is not an object attribute as in VOM, rather a non-versioned(!) entity with its own attributes: `DateTime`, `Validity`, `Name`, etc.
- *Validity*: Boolean expression over options to describe valid combinations of options. It corresponds to attribute constraints in VOM.
- *Fragment*: Basic information item, such as a relational tuple or a text line.
- *Visibility*: Boolean expression attached to each fragment of `VersionedObj` instances.
- *Version*: It consists of all fragments where the visibility evaluates to `True` for the given version-choice. A version is not an instance itself, but the result of a functional evaluation!
- *Version-choice*: A set of (option,value) bindings to describe which version an application task wishes to see. A version-choice is *complete* if a unique version of the DB is produced.
- *Version-description*: High-level DB query, that maps to a low-level *version-choice*, which defines a specific *version-view* or version of the DB.
- *Product-description*: Tuple of (Root objects, ER types), that maps to a *product-view* or product *closure* on the DB. The product is described by `SysBody` and similar entity

types. Note, that all **DataEntity** instances have a long **Contents** field, represented by an external file.

- *Config-description*: *Generic* (Product-description, Version-description), that maps to a *config-view* closure on the DB.
- *Specific, primary configuration*: a config-view to be used as a *workspace* by developers or tools.
- *Specific, derived configuration*: can be produced by the Builder as *Deriv-Config := Build(Product-view(Version-view(DB)))*.
- *Change job*: An edit-update *task* that controls a long *transaction* associated with a workspace, thus linking CM to PM.
- *Ambition*: A non-complete version-choice associated with a change job. It identifies the set of version-choices where new changes are going to be visible.

To interface old tools, the only viable solution is to *check-in* and *check-out* local workspaces of files and related DB information. This corresponds to the *copy-modify-merge* paradigm in NSE and PACT [Sun88] [Bul87]. An example of two coordinated workspaces is shown in figure 1. Here, workspace WS_1 belongs to Project P_C. This is a subproject of project P_A that controls workspace WS_2. Each

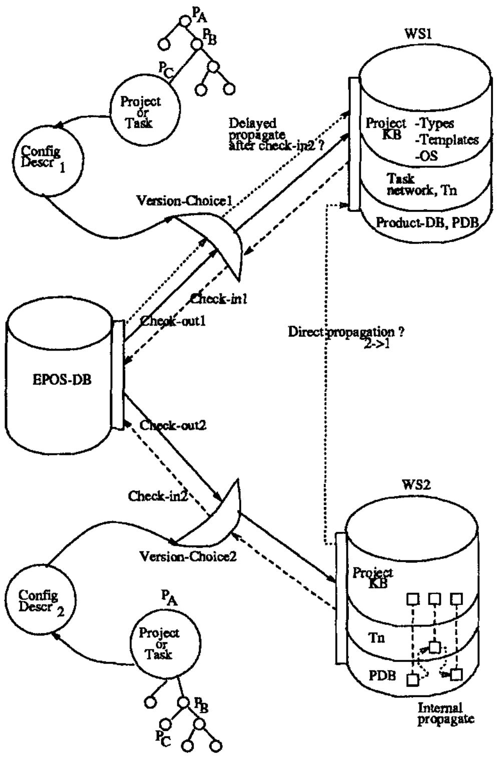


Figure 1: Central DB with checked-out workspaces, WS_i

workspace or configuration is controlled by a change job and its configuration description. A workspace is divided in three: product DB, task network, and project knowledge base (KB, with types and various other descriptions).

3 PM in EPOS

3.1 General Demands for Generic PM Support

Our goal is a common, system-interpretable formalism to describe software development processes. The model must cover:

- Deriver *tools*: their description (pre/post-conditions, inputs/outputs, tool switches) and aggregation.
- *Human actors* in an open-ended way.
- "Active" relationships to express change propagation by triggers.
- Chained tasks for *horizontal* life-cycle phases, revision lines, or derivation graphs.
- Subtask hierarchies to describe *vertical* work decomposition.
- Complex interactions between tasks and tools/users.
- Special project tasks to control workspaces with project-specific information (e.g. types).

3.2 A Survey of the PM Model

Our activation mechanism is coarse-level and mainly *descriptive*, based on PRE- and POST-conditions in types. Such conditions seem to flexibly express the activity rules for PM. They are also well-suited for *static*, forward and backward reasoning without executing the CODE.

The CODE associated to a task instance is responsible for causing its POST-condition to become True, and thus cause ("fire") other PRE-conditions to become True etc. The POST/PRE coupling therefore serves as a *dynamic synchronization mechanism.* The CODE of a task may re-execute, repeating the PRE/POST pattern above.

The experience with unrestricted "firing" of unbound rules and *triggers*[2] in databases, AI applications, and syntax editors [HN86] made us sceptic to such solutions. In our case, direct task communication is limited to relationship-connected *neighbor tasks* (4.5.1), and can thus describe traditional message passing and *notification.* Note the analogy with Petri nets [Rei85]. Note also that task execution can occur at any network node, not only at the leaves – cf. ISTAR. This distinguishes the network from a finite state machine, where only one node at a time is active.

[2]We may, of course, consider a task with a PRE-condition and an imperative CODE as a "trigger", with syntax IF — THEN — . Note also, that the EPOS *task type* corresponds to a conventional AI *rule*.

Dynamic subtask creation allow for considerable freedom in organizing the software work, and alleviates the limitations of static CODE in instantiated tasks. There are FORMALS and DECOMPOSITION properties (constraints) to regulate the *structure* of the task network.

3.3 The Activity Manager and Planner

Management of task types and instances is done by the EPOS *Activity Manager (AM).* and its associated Planner [Mul89] [LC89]. High-level or more complicated tasks must be delegated to human actors. Generally, we will have a high-level *goal* to achieve, and a product DB and a project KB as the starting point.

The relations between the Activity Manager and types, tasks and products are shown in figure 2:

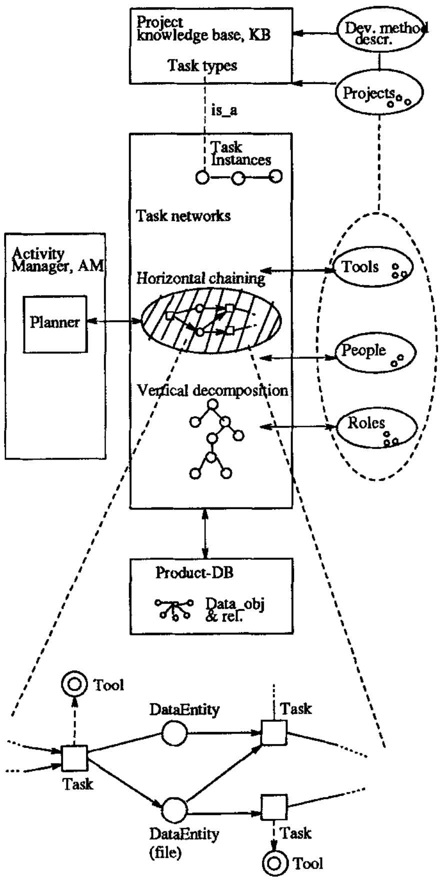


Figure 2: **AM and data**

The Activity Manager consists of:

- A *Type Manager (TM)* to construct and modify specific `TaskEntity` subtypes.

  The TM cooperates with a project manager or another meta-user.
- An *Instance Manager (IM)* to dynamically create, edit, and delete task instances; and to manage horizontal and vertical composites of such. The FORMALS and DECOMPOSITION information from task types will be used to automatically construct new and type-validated task structures.

  The IM cooperates with the Planner or a user to handle simple tasks (tool activations), and with a meta-user for more complex tasks (e.g. projects).
- An *Execution Manager (EM)* to interpret, execute, control and record task instances – while obeying the static task knowledge. E.g., tools must be called correctly, with dynamic checking of parameters if necessary.

  The EM cooperates with tools, users, the Builder (an EM component), and the Planner. Possible error situations must be monitored for alternative actions, and may include replanning and re-execution.

The Planner will:

- Offer *product-level* assistance such as construction of the empty derivation graph – a *plan* – for the Builder, cf. [HC88]. The plan may be *incomplete*, and a `TaskState` attribute of the planned tasks may reflect this.
- To some extent offer *project-level* assistance about task or work decomposition, work plans, etc.
- Utilize *static*, method-specific type knowledge about legal task communication and combination patterns. This is expressed by PRE, POST, FORMALS and DECOMPOSITION properties. This knowledge will be combined with information from the product DB. Only the non-temporal aspects of PRE- and POST-conditions will be considered, excluding comparison of `DateTime` timestamps, or scheduling rules like `<Compile between 24:00 and 06:00>`.
- Replan upon execution failures.
- Assess the impact of changes.
- "Learn" by putting generalized PM types back into the KB.

The Planner will borrow ideas from MARVEL, the AGORA PLANNER [BLA88], the domain-independent IPEM [AS88], and case-based planning.

4 EPOS-OOER, The Semantic Data Model

A common, semantic data model for CM and PM is sought, since there is much interaction between the two areas. **EPOS-OOER** represents a unification of ER [Che76] and *object-oriented* (OO) modelling, and allows general subtyping. The use of low-level pointers instead of general relationships in OO models has been remedied [Rum87]. From semantic network models we have been inspired by uniform handling of entities and relationships [TL82].

4.1 Available Type Properties

Multiple subtyping is specified through SIMULA-style prefixing. We will not define the *schema* notation formally, as the examples should be self-explanatory. The repertoire of type properties is explained in the following sections.

4.1.1 DOMAINS and ATTRIBUTES

Attributes belong to given domains, and represent passive *variables* or active *PRO-Cedures* (methods). Domains are declarable, and include scalars, text strings, pointers (REFs, see below), PROCs, and pre-defined ones like `Bool`. A REF-value is the value of the `ObjId` attribute of the referred object (see `RootER` type below). REFs are restricted to non-`PersistentObjs`, temporary PROC parameters, and access functions for CONNECTION data (4.1.2). In task types externally callable PROCs are constrained to functions without side-effects, i.e. functional attributes. Initialization is provided by `= <init-value>`. CONST means read-only or system-maintained variables.

Inheritance means that the supertypes' global attributes and domains are inherited into the subtypes. Multiple names in the supertypes are resolved by bottom-up, left-right search. A local name will always hide or overload the global one.

An example of a type definition with domains and attributes is:

```
TYPE RootER =                                % No prefixing supertypes.
DOMAINS
  DT = String;                               % A domain definition.
ATTRIBUTES
  % Four system-maintained attributes:
  ObjId    : CONST <64-bits>  = ...;         % Unique and immutable.
  TypeId   : CONST 'REF TYPE' = ...;         % 'Is-a' relationship.
  DateTime: CONST DT          = ...;         % Create time.
  Primary  : CONST Bool       = ...;         % Prim/Deriv?
  New      : PROC (--)=--;                   % In meta-type?
  Delete   : PROC ()  =--;                   % Similarly.
  Read     : PROC (--)--=--;                 % On attrs also
  Write    : PROC (--)=--;                   % Make version?
  Select   : PROC (--)=--;                   % Very general.
  ---
END-TYPE RootER;
```

4.1.2 CONNECTION

Types with no CONNECTION property are implicitly called **entity** types. Inversely, a CONNECTION implicitly identifies a *binary* **relationship** type, that connects two entity types[3]. Only single type inheritance is allowed at the user

[3]In ER terminology, *roles* should have been used here. – And the debate whether relationships should have identity and TypeId or not, is still going on!

level, starting from the Relationship type below. Inheritance may constrain cardinalities and related entity subtypes, and may rename the access functions.

An example of a basic relationship type is:

```
VersionedObj, PersistentObj, GenRelationship TYPE Relationship = % NB!
CONNECTION
  Acc1 : Entity1(l1..u1) <-> Acc2 : Entity2(l2..u2);
   % Defines a binary relationship between Entity1 and Entity2
   % with access functions Acc1 and Acc2, and
   % with cardinalities l1..u1:l2..u2 - or u1:u2 in short.
   % ui = * means unbound cardinality, and SEQ after ui means ordering.
   % NB: The suffixes in Entity1 and Entity2 only serve to distinguish
   %     these from one another!
   %     Also, li = 0 and ui = * at this general type level!
ATTRIBUTES
  Closure: PROC (--)=--;     % Special DB op.
  ---
END-TYPE Relationship;
```

Named access functions (Acc1 and Acc2 above) are implicitly available to the "related" entities, but formally defined in the connecting relationship type. Assuming a cardinality of `*:1`, Acc1 could be defined in Entity2 with domain 'SET(REF Entity1)'. With a cardinality `1:*`, the domain of Acc1 could be 'REF Entity1'. "CONNECTION data" may be stored inside the related DB entities as pointers or sequences of such for reasons of efficiency.

A short-hand notation for simple relationship types, implicitly prefixed by Relationship is offered:

```
RELATIONSHIP-TYPE
  RelType = Acc1 : Entity1(l1..u1) <-> Acc2 : Entity2(l2..u2);
```

4.1.3 CODE

This is a piece of program code expressed in an imperative programming language, executed by the Activity Manager. CODE is primarily used in TaskEntity types, but may be used for initialization of non-tasks. However, such instances are not "active" tasks in the OO sense.

The CODE language is *not concurrent*, as all triggering is implicitly expressed by PRE- and POST-conditions. The CODE language is a restricted Unix shell language. STOP means task termination. Shell variables such as $<name> can be accessed. Into(AccFunc,X) means insertion of X into a relationship. INNER means execution of the subtype's CODE. It will be appended in the CODE part, if missing. Type inheritance implies concatenation, using SIMULA's INNER mechanism.

4.1.4 PRE- and POST-conditions

As mentioned, a task waits for its PRE-condition to become `True` before (re-)activation of its CODE part. After each activation, the POST-condition must be fullfilled. PRE and POST are intended for task types, but can be envisaged as initialization and termination constraints elsewhere. Type inheritance is by conjunction ($\wedge$), starting from `True` in the `TaskEntity` type.

PRE- and POST-conditions are formulas in first-order predicate logic, with the following predicates and functions:

`ALL(x:accfunc!cond)`	`True, if valid for all x`
`SET(x:accfunc!expr)`	`set of all expr's`
`CLOSURE(accfunc)`	`closure from curr.obj (def. in Relationship)`
`CLOSURE(x,accfunc)`	`closure from x obj`
`NOTIFY(set)`	`DB change-message`
`t1 SUBTYPE_OF t2`	`subtype test`
`set1 IN set2`	`subset member test`
`x WITHIN (.values.)`	`scalar range test`
`MIN(values),MAX(--)`	`minimum, maximum`
`ANDIF, ORELSE`	`McCarthy and/or`

For technical reasons, the PRE-condition is split into two parts: PRE_STATIC used by the Planner and PRE_DYNAMIC used by the Execution Manager. We have not yet found it necessary to split up the POST-condition similarly.

The PM types (4.5) contain examples of PRE- and POST-conditions.

4.1.5 FORMALS and DECOMPOSITION constructors

Both specify simple type templates, and are restricted to `TaskEntity` type and subtypes. Type inheritance rules are as for attributes, although we have considered various schemes of subtype-constraining semantics [CW85].

FORMALS constrains the indirect, *horizontal chaining of tasks*, i.e. the legal types of actual task parameters. These parameters are expressed by the `GenInputs` and `GenOutputs` relationships between `TaskEntity` instances and their inputs and outputs (4.5.1).

An example of a FORMALS specification is:

```
TaskEntity TYPE TaskX =
FORMALS
  a:C_Source * $b:C_Include -> c:DerivEntity
```

This means that `TaskX` instances take one input of type `C_Source` and a variable number of inputs of type `C_Include` (because of the starting `$`-sign in the parameter name), and produce one output of type `DerivEntity` – or subtypes thereof for all these types. All parameter types must be subtypes of `Entity`, i.e. no relationships can be processed by tasks. The default FORMALS specification is:

```
$in:Entity -> $out:Entity
```

i.e. no constraints at all. CHECK_IN_FORMALS and CHECK_OUT_FORMALS predicates are available for use in PRE/POST-conditions, see Sec. 4.5.1.

DECOMPOSITION constrains ***vertical task breakdown***, i.e. the SubTasks relationship between parent and children tasks (4.5.1).

An example of a DECOMPOSITION specification is:

```
TaskEntity TYPE TaskY =
DECOMPOSITION
  CHOICE(SEQ(ta:T1,tb:T2), PAR(tc:T3,td:T4), tseq:REPERTOIRE(T5,T6,T7));
```

This means that possible children of TaskY instances may be either instances of T1 and T2 executing in sequence, instances of T3 and T4 executing in parallel, or any number of instances of T5/T6/T7 executing in parallel (PAR is implicit). The default is REPERTOIRE(TaskEntity), i.e. no constraints.

The Activity Manager and Planner uses DECOMPOSITION for automatic generation of children tasks after creating a parent task, and so on. To avoid redundancy with Into(Children, NEW TaskEntity(...)) in the CODE part, the DECOMPOSITION should be empty, if the parent task explicitly generates or kills its children tasks. In case of a SEQ specification, synchronization of children tasks will be checked and possibly enforced by the Planner.

4.1.6 EXECUTABLE

This identifies the *logical* name of the associated OS-tool of a deriver task. Such a tool can be considered an "external" PROC, possibly shared by many similar task types. Upon workspace initialization, a "soft link" between the *type* object(s) and the selected Executable *instance* will be established.

4.1.7 INVARIANT

This is a formula in first-order predicate logic, specifying an assertion over certain DB instances. It should always be True. Type inheritance is by conjunction ($\wedge$). Sec. 4.5.5 contains an example of an INVARIANT.

4.2 Pre-defined Types

A type semi-lattice of some of the *pre-defined* types is shown in figure 3, with the *system-defined* ones above the dotted line:

For a real project, all the main types have much more subtypes than indicated.

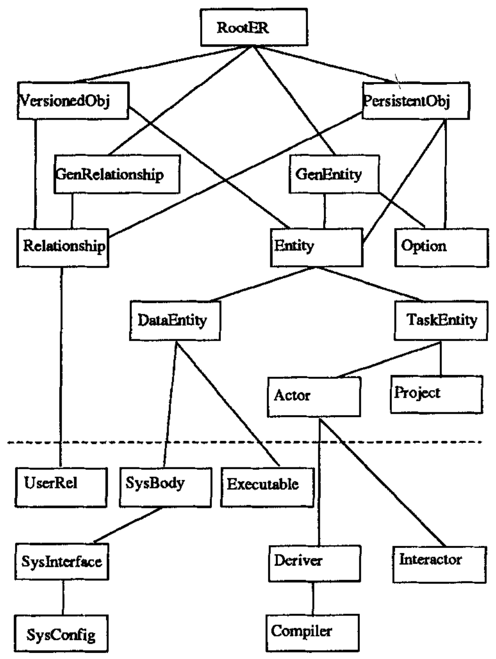


Figure 3: EPOS type semi-lattice

4.3 General Comments on EPOS-OOER

DB *instances* of any type can be created, by `NEW Type(List of <AttrName = Value>)`. Entity instances are often called "objects".

The types are represented as normal DB objects or `TypeDescrs` (4.4), available for dynamic interpretation. There are no meta-types.

The non-PROC ATTRIBUTES and CONNECTION part of EPOS-OOER is called the *CM-part.* The rest is called the *PM-part,* since such properties are stored at the type level and are implemented by the Activity Manager. This PM-part can be *versioned* (4.6).

Note, that EPOS-OOER is not quite "unified": There are limitations on the use of pre-defined types as supertypes. E.g. `PersistentObj` and `VersionedObj` (2.2) mostly serve as *keywords,* `GenRelationship` subtypes are restricted to simple inheritance, and `Option` cannot be used as a supertype at all. Type properties beyond domains and attributes are only relevant for `TaskEntity` subtypes.

An alternative conceptual base would have been more specialized type categories – DATAENTITY, RELATIONSHIP and TASKENTITY – and a more closed data model.

4.4 On Task / Tool modelling

The OO ERA modelling in EPOS-OOER follows conventional patterns. The problem has been the modelling of tasks and tools, where the solutions have changed fundamentally over the last year. We will repeat the relevant arguments and rationale for the uninitiated reader.

The main PM type is now `TaskEntity` (4.5.1). A `TaskEntity` instance describes a potentially *active* process or task. Its CODE part may execute its own program, and indirectly that of its children. The low-level, more specialized `Deriver` tasks may directly call *passive* OS-tools, so-called `Executables`, through a procedural envelope with given file parameters. Static tool aggregation is done by DECOMPOSITION of associated task types. Versioning of OS-tools is done by the normal CM mechanisms, and versioning of task types is explained in Sec. 4.6.

We have also had problems in modelling formal parameters and task/tool templates. Both these imply *type-level* "relationships" or constructors to express constraints, which are not easily expressible in an ER framework:

- *Formal parameters* were initially expressed by ad-hoc `ProcInputs` and `ProcOutputs` *"meta-relationships"* to connect a `TaskEntity` *type* with its legal input and output *types*. The type information of such formal parameters could then be matched against the types of the actual parameters.

  It remains a problem to compactly express *shared* formal parameters between similar deriver tasks. We may need a separate *signature* type ("arrow" type) for this.
- *High-level templates* for task/tool aggregates and task networks may assume *type composites* at the meta-level.

Instead of having a more sophisticated type apparatus on the *meta-level*, we have introduced the FORMALS and DECOMPOSITION properties. However, we have no separate `Template` type to describe more generalized type composites.

FORMALS and DECOMPOSITION are internally implemented by system-maintained "shadow types", `TypeDescrs`, at the *instance level*. Such `TypeDescrs` facilitate, in principle, any kind of *high-level typing* through arbitrary type relationships or *constructors*[4].

Some more technical arguments have been:

- An EPOS `Deriver` task is an abstraction of an activation of a real *OS-tool*. Such an OS-tool is called by issuing a command line (a script) to the OS, with proper tool name, tool switches and file parameters.
- Tool *envelopes* are needed to hide OS details and to provide project instrumentation and error treatment.
- Tools cannot be expressed as traditional OO "methods" (PROCs) in such types. This is because the tools may be shared by several task types and

[4]EPOS-OOER only supports `''Subtype_Of''`, FORMALS and DECOMPOSITION type constructors.

possibly independently produced and versioned by the surrounding OS – cf. EXECUTABLE property and next item.

- *Tool switches* cannot easily be modelled by traditional parameters, due to varying number and special semantics. E.g., some switches cause extra output to be produced, so that different "deriver variants" must be defined with appropriate FORMALS[5]. Centralized control over default tool switches is also desirable.
- Tool *aggregates* are needed to hide details, using DECOMPOSITION to express super-tools.
- Lastly a reminder: We do *not* want to model *all* the "hairy" semantics (to put it mildly!) of the OS-tools. Only the essential parts for basic tool management need be covered.

There are *three* different task/tool breakdowns, all N:M:

- The static type semi-lattice.
- Static task/tool type aggregates through the DECOMPOSITION constructor.
- Dynamic decompositions of task instances through a SubTasks relationship, obeying the DECOMPOSITION constraints.

4.5 The main PM Types

4.5.1 The TaskEntity Type

TaskEntity has an Actor subtype (4.5.2) to execute an interactive or automatic tool, a Project subtype (4.5.5), and more specialized subtypes.

As mentioned, the CODE in a task instance is executed when the PRE-condition evaluates to True. A task serves as a *coroutine* – with an implicit, embedding loop around the outermost CODE part. The task execution environment is assumed to be cheap – probably co-routines in the Activity Manager, plus forking of real OS processes to execute OS-tools.

The TaskEntity definition is:

```
Entity TYPE TaskEntity =
DOMAINS
  TaskStates = ENUM(Created, Initialized, Waiting, Ready,
                    Active,  Terminated,  Deletable);
  Eagerness  = ENUM(Busy, Periodic, Opportunistic, ..., Seldom, Lazy);
ATTRIBUTES
  TaskState :       TaskStates = Created;
  BatchTool :CONST Bool = ...;                   % Deriver?
  Parallel  :CONST Bool = True;                  % PARallel?
PRE
  CHECK_IN_FORMALS(Inputs)                       % Type-check inputs.
```

[5]Indeed, the Unix CC-compiler may be used both as a pre-processor, compiler, linker, assembler, ... – and with different FORMALS within these categories!

```
CODE % Coroutine:
% DO                                                % Implicit DO.
    <Initial CODE part>;
    INNER;
    <Final CODE part>;
% OD;                                               % Implicit OD.
POST
  CHECK_OUT_FORMALS(Outputs)                        % Type-check outputs.
  AND ALL(o:Outputs ! o.Primary = NOT BatchTool) % Deriver condition.
  AND ('Success' OR 'Failure')                      % Outcome predicate.
END-TYPE TaskEntity;

RELATIONSHIP-TYPE
  SubTasks  =Parent   :TaskEntity(0..*) <-> Children:TaskEntity(0..*SEQ);
  GenInputs =GInNode  :TaskEntity(0..*) <-> Inputs   :Entity(0..*SEQ);
  GenOutputs=GOutNode:TaskEntity(0..*) <-> Outputs  :Entity(0..*SEQ);
```

Name attributes of GenInputs and GenOutputs actual parameters are omitted for sake of clarity.

4.5.2 The Actor Type

Actor is a trivial TaskEntity subtype; not shown. It has two subtypes, an Deriver (4.5.3) and Interactor (4.5.4).

4.5.3 The Deriver Type

As mentioned, the Deriver type represents a deriver tool. Its CODE part is a shell script to prepare, activate, control, and record an OS-tool activation. This is done through "INNER"-subtyped *envelopes* of Pre/Post-Actions. Such actions may include "invisible" control and accounting functions, as in GENOS [GEC87]. The logical name of the OS-tool is given by the EXECUTABLE property. *Default* tool switches will be provided from the global CurrProject. *"Crucial"* tool switches, affecting the FORMALS, are contained in a local DeriverSwitches attribute. This attribute and the matcher PROC attribute (see 5.1.1) will be redefined by subtypes.

A Deriver definition with a *busy* rebuild rule is:

```
Actor TYPE Deriver =
ATTRIBUTES
  Analyzed        : DateTime;            % Last derivation.
  DeriverSwitches: String = 'xx';        % Redefined in subtypes.
  Matcher: PROC () = ---;                % Make derivation graph.
  TooOld : PROC () Bool =
  BEGIN
    MAX(SET(i:Inputs  ! i.DateTime)) >  % Checking timestamps within
    MIN(SET(o:Outputs ! o.DateTime))    % a configuration.
  END TooOld;
```

```
PRE                                      % Busy build in local config:
  TooOld() AND CurrProject.Rederiv_Policy() = Busy
EXECUTABLE 'xx'                          % Redefined in subtypes.
FORMALS    ---                           % The same.
CODE
  <Check derivation cache>;
  <PreAction>;                           % Shell script.
  Call-OS-Tool(<EXECUTABLE-name>,
               DeriverSwitches, CurrProject.DefaultSwitches,
               <file parameters>);
  INNER;                                 % Extra subtype actions.
  <PostAction>;                          % Shell script.
  <Update derivation cache>;
  <Assemble product statistics>;
  <Report to PM recording tool>;
POST
  <Extra assumptions on output parms> AND <Success and failure modes>
END-TYPE Deriver;
```

A coarse description of success and failure modes must be supplied. Note, that each project may define its own rederivation policies (PRE-conditions) or CODE parts.

A deriver is typically a compiler, link editor, text formatter etc. We can envisage a `Compiler` subtype, with a `CC-Compiler` subtype of this etc.

4.5.4 Interactor Type

`Interactor` is an `Actor` subtype representing an interactive tool activation, coupled to a `Role` instance. The `Interactor` definition may look like:

```
Actor TYPE Interactor =
PRE
  <Input changed>
CODE
  <Call some role/tool pair, e.g. a Maintainer + Assistant>;
END-TYPE Interactor;
```

A `Role` represents a "canonical" person, emphasizing authorization, job position, and general project attachment. The `Role` will again be connected to specific `Person`(s), having responsibility of certain software components.

4.5.5 Project Type

`Project` is a `TaskEntity` *subtype* to emphasize the productive aspect. A project specifies a *project KB* connected to a workspace. A new project instance, e.g. `CurrProject`, may be created in the workspace of its parent project. The new project will inherit the parent's workspace, and in addition create a *nested* workspace of its own. In principle, the project KB can be changed on-the-fly! See Sec. 4.6 for binding mechanisms to achieve flexible project tailoring.

The project KB should contain the following information:

- A *config-description* (2.2) for the current DB transaction in the local workspace. This may include a traditional DB view, with rights and capabilities.
- A *coarse OS description*: specific or low level OS-tool information, such as file bindings, environment flags, default OS-tool set including tool switches (e.g. `-I <directory_name>` to compilers), ...
- *Subprojects*, and allocation of *persons and resources* – as in ISTAR.
- *Project-specific rules and policies*, through various PM types, task/tool templates, invariants, project-pervasive attributes, ...

Ex. Project policies for *CurrProject* = `ProjectX`.

```
Project TYPE ProjectX =
ATTRIBUTES
  DefaultSwitches : String;                     % Used above.
  Rederiv_Policy  : PROC () Eagerness = BEGIN Busy END Rederiv_Policy;
PRE
  ProjectAccount <> NIL AND ProjectLeader <> NIL  % Def. elsewhere.
CODE
  <Create subtasks>;
POST
  ---
DECOMPOSITION
  REPERTOIRE('Approved-task-types')             % Special INVARIANT.
INVARIANT
  % Expresses policies: All documents created shall bef of (sub)types
  % approved by the project.
  ALL(x:CLOSURE(Children).Outputs
        ! x.TypeId SUBTYPE_OF 'Approved-document-types')
END-TYPE ProjectX;
```

4.5.6 Executable Type

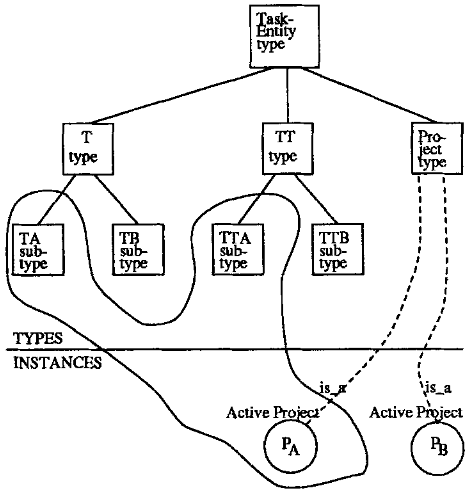
This is a `DataEntity` subtype, and specifies `BinaryProgs`, `ShellScripts`, or other OS-tools.

4.6 Project Customization of Typing

Some possible binding mechanisms are:

- *Static inheritance* in the task type hierarchy, i.e. project-specific *subtyping*. This may lead to a proliferation of subtypes and mutual constraints on correct subtype selection, cf. option validities and figure 4.

- *Type parameterization* and instantiation of generic project types, as an alternative to subtyping.
- Project *versioning*: a generalization of subtyping, which is easy to achieve in our COM model from a technical point of view. However, we want essentially non-versioned types/DB-schemas to prevent a semantic explosion of sub-universes in the DB [SZ86].
- *Dynamic inheritance* in the task instance hierarchy, or along any given relationship.
- *Dynamic instrumentation* of the Activity Manager through special couplings to `CurrProject`, cf. `CurrProject.Rederiv_Policy` used by the Builder.



PA selects TA and TTA, and PB selects TB and TTB.
That is, a coupling from the instance world to the type world.

Figure 4: Consistent choice/version of subtypes

The EPOS solution is to use the system-maintained `TypeDescr` instances (4.4) to express limited *type versioning* within the workspace of a project, i.e. of the PM-part only. This guarantees some minimum DB stability.

A comment on the EPOS model of type versioning: It may seem primitive to let a set of *boolean* or scalar options "parameterize" our type system. However, we want to version an *entire* collection of types and other control information at the same time. Alternative approaches with sophisticated type parameters in addition to subtyping (cf. Eiffel [Mey88]) are more complex but still insufficient. In contrast, our scheme uses the *existing* versioning and workspace mechanisms in EPOS, i.e. it has a low human and computer cost.

5 Applications of the PM Model

5.1 Active Relationships, Derived Objects and the Builder

5.1.1 Active Relationships and the Derivation Graph

An intuitive modelling of "active" relationships is to associate triggers (rule-coupled tools) with relationship types. Since we do not allow *n-ary* relationships with possible task decomposition, we must insert extra task entities between the "N:M"-related objects, see next paragraph.

The conversion from a *dependency* graph with "pure" relationships to a *derivation* graph with inserted task nodes is *language-specific*. It is taken care of by the `matcher` PROC in the `Deriver` subtypes. For instance, `GenInputs` in a derivation graph must represent the *transitive closure* of the relevant inputs for programming languages like C, Pascal and Fortran (large, but shallow graph). That is, the `Deriver` step to compile a C-program `X.c` will require the "body" file `X.c`, the "interface" file `X.h`, and *all* `.h` files that these two files transitively include. This closure is not necessary for languages like Ada and Modula, having separately compiled interfaces (smaller, but deeper graph).

5.1.2 Derived Objects

Derived objects are different from *primary* objects. A derived object of type `Entity` is stored as a versioned, functional `DerivResult` "attribute" in an instance of `DerivEntity` type (not defined here). This instance represents a possibly empty version group of derived objects. Different tools operating on the same input objects must be described by *different* derived objects and derivation graphs.

When requested by an application, the versioned `DerivResult` attribute may be regenerated, using the available derivation graph. This corresponds to *lazy* build. The `DerivResult` attribute identifies the derived output, its inputs, and tool version and tool switches used. There is an accompanying, versioned `DateTime` attribute.

The set of non-empty versions of the `DerivResult` attribute can be treated as a global *cache* of derived objects, and is subject to user policies for deletion; see ODIN [Cle88]. It is important to *share* attribute versions between configurations – i.e. smart recompilation! – by "increasing" attribute *visibilities* [Lie89].

5.1.3 The Builder

The *Builder* operates in the current or LOCAL workspace (2.2). It is really a part of the Activity Manager, which also has created or planned the derivation graph of task objects. An example of a derivation graph is:

The Builder will generate a complete, derived configuration – upon explicit re-

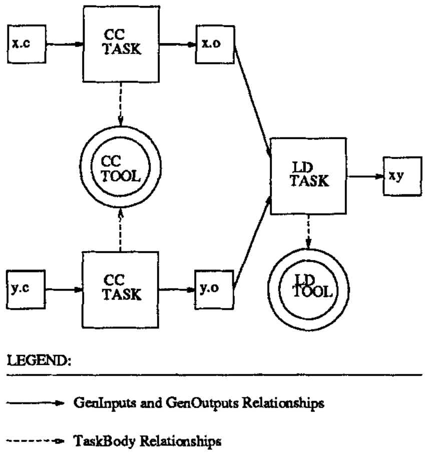


Figure 5: A derivation graph

quest (*lazy*, backward chaining), or when triggered by project-specific rules (*busy*, forward chaining). Error handling is difficult, e.g. to interpret compilation errors sensibly. Suppose that we want to nightly rederive the outdated objects, caused by DB-changes in the LOCAL workspace:

```
Deriver TYPE LocalDeriver =
PRE                              % Note PRE-concatenation by subtyping.
  NOTIFY(Inputs) AND (Inputs IN LOCAL) AND
  ClockTime() WITHIN (.24:00 .. 06:00.)
---
END-TYPE LocalDeriver;
```

Such a policy could also have been defined in `CurrProject.Rederiv_Policy` (4.5.5).

5.2 Subtasking and Task Sequencing

The task-subtask hierarchy, as illustrated in figure 6, covers many different purposes.

Projects fit nicely into the task hierarchy due to the definition of `Projects` as a `TaskEntity` subtype. This implies that projects may be decomposed into subprojects, and that they have a limited lifespan.

Task transition chains can be used to describe phases or revisions of software components at a more detailed level:

- Horizontal life-cycles, such as (`Requirements, Specification, Design, Implementation, Testing, Release, Delivery`).

- Revision lines, such as edit, review and test operations on individual objects within a lifecycle phase. `Status` attributes with values such as (`Initial`, `Experimental`, ..., `Finished`, `Integrated`, `Approved`) are suitable for composites like a configuration.

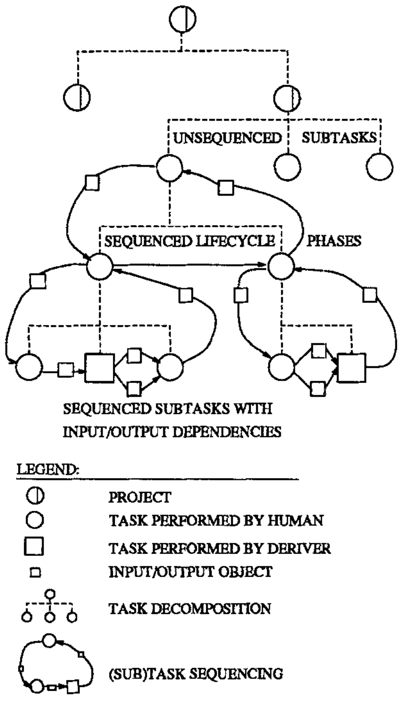


Figure 6: General Subtasking

Below is an example of task modelling of a development task, with 5 subtasks of the requirements specification phase:

```
Project TYPE DevelopJob =
PRE                                                    % Assumes existence
  Parent <> NIL ANDIF                                  % of InformalReqSpec.
  Parent.InformalReqSpec.Status >= Experimental        % Note '>='.
CODE
  ---
POST
    ALL(     rt:Children ! rt.TypeId SUBTYPE_OF ReqTask
        ANDIF rt.Status >= Approved)
  AND
    ALL(     tt:Children ! tt.TypeId SUBTYPE_OF TestTask
        ANDIF tt.TestLog.Status >=Approved)      % TestLog=produced doc.
  AND
    ALL(     dt:Children ! dt.TypeId SUBTYPE_OF DeliverTask
        ANDIF dt.Status = Finished)
```

```
DECOMPOSITION
  SEQ(ReqTask, ImplTask, TestTask, ReleaseTask, DeliverTask)
END-TYPE DevelopJob;
```

6 Conclusion and Future Work

The EPOS PM model provides both a dynamic and a static view of description, planning and execution of software processes. The model covers deriver tools, human actors, high-level projects and low-level tool activations, task transition networks, and project and task decomposition. Both type and instance hierarchies can be used to express task knowledge.

The CM and PM areas are connected through a common data model, EPOS-OOER. CM is coupled to PM through change jobs associated with config-descriptions, and through more detailed revision tasks. Likewise, PM-relevant control information is contained in a project KB, which is versioned (i.e. controlled by CM) to allow easy project customization and evolution.

All in all, we think that the proposed PM support is a fruitful basis for continued work in the area. Still, there are many issues to be pursued:

- A more powerful, imperative CODE language.
- The well-suitedness of PRE/POST-conditions for for general task synchronization. Consider a task that can be activated in three different ways, identified by PRE-conditions B1, B2, and B3. This has to be written as:

  ```
  PRE
    B1 OR B2 OR B3 OR ---
  CODE
    IF      B1 THEN Code1
    ELSEIF B2 THEN Code2
    ELSEIF B3 THEN Code3
    ELSE ---;
  ```
- More high-level type templates for task/tool patterns. See e.g. [ENE87] on graph grammars.
- A more generic and possibly dynamic data model, to avoid proliferation of trivial subtypes each time e.g. a new programming language is added. This resembles versioning of task types according to the current project.
- Better formalization of projects, and their workspaces for long transactions.
- Overall methodologies for project and process modelling.
- Better modelling of CASE-like meta-tools with internal tool policies.
- Planning: heuristics, intertwined planning and execution, knowledge representation, and KB support.
- Industrial scenarios and trial use.

Only a prototype EPOS implementation will be built in Trondheim, and only of the basic CM and PM system. On the other hand, CM has high priority within our industrial partners, Sysdeco and Veritas Research, so that future industrialization seems assured.

Acknowledgements

Thanks to P. Lavency et al. from Philips Research in Brussels on PM, and to J. Müller from the Techn. Univ. of Karlsruhe on the Planner design.

References

[AS88] José A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proc. of AAAI'88*, pages 83-88, 1988.

[B*89] K. Benali et al. Presentation of the ALF project. In *Prelim. Proceedings from Int'l Conf. on SDEF, Berlin, 9-11 May 1989*, page 23 p., May 1989.

[BE87] Noureddine Belkhatir and Jacky Estublier. Software management constraints and action triggering in the ADELE program database. In *[NS87]*, pages 44-54, 1987.

[BL89] Yves Bernard and Pierre Lavency. A Process-Oriented Approach to Configuration Management. In *[IEE89]*, 1989. 14 p.

[BLA88] Roberto Bisiani, F. Lecouat, and Vinzenco Ambriola. A Planner for the automation of programming environment tasks. In *Proc. of the 21st Annual Hawaii International Conference on System Sciences*, pages 64–72, Hawaii, USA, January 1988.

[Bul87] *PACT: The initial PACT Environment.* Bull, Louveciennes, France, September 1987.

[C*89] Reidar Conradi et al. *EPOS Day Compendium – 1 Nov. 1989.* Technical Report, DCST, NTH, Trondheim, Norway, October 1989. NTH, 23 Oct 1989, 174 p.

[CDrG*89] Reidar Conradi, Tor Martin Didriksen, Bjørn Gulla, Håvard Eidnes, Even-André Karlsson, Anund Lie, Per Harald Westby, Svein Olav Hallsteinsen, Per Holager, and Ole Solberg. Design of the kernel EPOS software engineering environment. In *Proc. from Int'l Conf. on System Development Environments & Factories, Berlin*, May 1989. 16 p., Rev. Oct. 1989. Forthcoming as a Springer LNCS.

[Che76] P. P.-S. Chen. The entity-relationship model — towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cle88] Geoffrey M. Clemm. The Odin specification language. In *[Win88]*, pages 144-158, 1988.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471-521, 1985.

[DGS89] Wolfgang Deiters, Volker Gruhn, and Wilhelm Schäfer. Systematic development of software process models. In *Carlo Ghezzi, John A. McDermid (Eds.): Proc. of ESEC'89 – the 2nd European Software Engineering Conference '89, Warwick, UK*, September 1989. Springer Verlag LNCS 387, p. 100-117.

[Dow87] Mark Dowson. ISTAR and the contractual approach. In *Proc. of the 9th ACM-SIGSOFT/IEEE-CS Int'l Conference on Software Engineering, Monterey, CA, USA*, pages 287-288, April 1987.

[ENE87] *Proc. 3rd Int'l Workshop on Graph Grammars and their Application to Computer Science*, Warrenton, VA, USA, 1987.

[GEC87] *GENOS, GEC Software's IPSE.* GEC Software, London, UK, May 1987.

[HC88] Dennis Heimbigner and Steven Crane. A graph transform model for configuration management environments. In *[Hen88]*, pages 216–225, 1988.

[Hen88] Peter B. Henderson, editor. *Proc. of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Boston, 28-30 Nov 1988), *257 p.*, November 1988. In ACM SIGPLAN Notices 24(2), Feb 1989.

[HM88] Tani Haque and Juan Montes. A Configuration Management System and more (on Alcatel's PCMS). In *[Win88]*, pages 217-227, 1988.

[HN86] A. Nico Habermann and David Notkin. GANDALF: software development environments. *IEEE Transactions of Software Engineering*, SE-12(12):1117–1127, December 1986. (Special issue on GANDALF).

[Hol88] Per Holager. *Elements of the Design of a Change Oriented Configuration Management Tool.* Technical Report STF44-A88023, 95 p., ELAB, SINTEF, Trondheim, Norway, February 1988.

[IEE89] IEEE/ACM, editor. *Proc. of the 11th International Conference on Software Engineering*, Pittsburgh, USA, May 1989.

[KF87] Gail E. Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *Proc. of the 9th ACM-SIGSOFT/IEEE-CS Int'l Conference on Software Engineering, Monterey, CA, USA*, pages 180--188, April 1987. (on MARVEL).

[LC89] Chunnian Liu and Reidar Conradi. Planning Software Development Processes in EPOS. October 1989. In [C*89].

[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 *Proc. of 2nd Int'l Workshop on Software Configuration Management, Princeton, USA*, October 1989. ACM SIGSOFT Engineering Notes, Vol. 17, Number 7 (Nov. 1989), pp. 56-65.

[LDC*89] Anund Lie, Tor M. Didriksen, Reidar Conradi, Even André Karlsson, Svein O. Hallsteinsen, and Per Holager. Change Oriented Versioning. In Carlo Ghezzi and John A. McDermid, editors, *Proc. of ESEC'89 – the 2nd European Software Engineering Conference '89, Warwick, UK*, pages 191–202, September 1989. Springer Verlag LNCS 387.

[Leh87] M. M. Lehman. Process models, process programming, programming support. In *Proc. of the 9th ACM-SIGSOFT/IEEE-CS Int'l Conference on Software Engineering, Monterey, CA*, pages 14–16, March 1987. (Response to an ICSE'9 Keynote Address by Leon Osterweil).

[Lem86] P. Lempp. Integrated computer support in the software engineering environment EPOS – possibilities of support in system development projects. In *Proc. 12th Symposium on Microprocessing and Microprogramming, Venice*, pages 223–232, North-Holland, Amsterdam, September 1986.

[Lie89] Anund Lie. *Outline Design of the EPOS Database.* Draft, DCST, NTH, Trondheim, Norway, April 1989.

[Mey88] Bertrand Meyer. Eiffel: a language and environment for software engineering. *The Journal of Systems and Software*, 199–246, 1988.

[Mul89] Jürgen Müller. *Process Management Using AI Planning Techniques.* Technical Report 29/89, EPOS report 86, 117 p., DCST, NTH, Trondheim, Norway, June 1989. (MSc Thesis).

[NS87] Howard K. Nichols and Dan Simpson, editors. *Proc. of the First European Software Engineering Conference* (Strasbourg, Sep 1987), LNCS *289* Springer Verlag, 404 p., September 1987.

[OR86] Martyn A. Ould and Clive Roberts. Modelling iteration in the software process. In Mark Dowson, editor, *Proc. of the 3rd International Software Process Workshop*, Breckenridge, Colorado, USA, November 1986.

[Rei85] Wolfgang Reisig. *Petri Nets – An Introduction.* Springer-Verlag, 161 p., 1985.

[Rum87] James Rumbaugh. Relations as semantics constructs in an object-oriented language. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 466–481, Kissimmee, Florida, October 1987. In ACM SIGPLAN Notices 22(12), Dec 1987.

[Sun88] *Network Software Environment: Reference Manual.* Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, USA, part no: 800-2095 (draft) edition, March 1988.

[SWKH76] Michael Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. on Database Systems*, 1:189-222, 1976.

[SZ86] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 483–491, Portland, Oregon, 1986. In ACM SIGPLAN Notices 21(11), Nov 1986.

[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, 1988.

[TL82] Dionysios C. Tsichritzis and Frederick H. Lochovsky. *Data Models.* Prentice Hall, 343 p., 1982.

[Win88] Jürgen F. H. Winkler, editor. *Proc. of the ACM Workshop on Software Version and Configuration Control* (Grassau, FRG, 27-29 Jan 1988), *Berichte des German Chapter of the ACM, Band 30, 466 p.*, B. G. Teubner Verlag, Stuttgart, 1988.