Process Modeling Paradigms: An Evaluation

Chunnian Liu, Reidar Conradi\footnote{Detailed address: Div. of Computer Systems and Telematics, N-7034 Trondheim-NTH, Norway. Email: conradi@idt.unit.no, Fax: +47 7 594466, Phone: +47 7 593444.}
Norwegian Institute of Technology (NTH), Trondheim, Norway.


\textbf{EPOS TR 134}


\section*{Abstract}
Software Process Modeling and systems for Software Process Management (PM) have recently drawn increased attention within the Software Engineering community. Yet there are no software process models generally recognized as adequate, nor process management systems put into practical use. Much more work has to be done. This paper identifies 6 themes considered the most interesting for the future research in this area. Around these themes, we summarize and evaluate different software process modeling paradigms and various PM systems, including \textit{EPOS} – the one we have been working on. The paradigm, experience and ongoing research of EPOS are presented and discussed in more detail in the last two sections.

\textbf{Keywords:} Software Process, Software Process Modeling, Software Process Management.

\section{Introduction}
A \textit{software process} is the total set of software engineering activities needed to transform a user’s requirements into functioning software. A \textit{generic software process model} is a framework providing the definitions, structures, standards, and relationships of the various process elements so that common technology, methods and measurements can be applied by any software project. A \textit{project-specific software process model} (e.g. waterfall, spiral, and iterative enhancement models) is a refinement of the generic model to reflect the particular needs of a project. This again can be instantiated to an executable \textit{process} (with subprocesses) to develop a particular piece of software. Thus we have the following chain: underlying semantic model \rightarrow generic process model \rightarrow project-specific process model \rightarrow concrete process(es) \rightarrow process activation(s). A \textit{Software Process Management (PM)} environment should \textit{enact} and \textit{control} development activities semi-automatically and concurrently.

In recent years, there have been quite a lot of PM systems using various process modeling paradigms, yet no software process models generally recognized as adequate, nor PM

\footnote{\textit{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.}
systems put into practical use. Classification, assessment, and combination of different paradigms are both interesting and important. A reference framework is needed for precise description of different paradigms and systems. As no paradigm is adequate for all aspects of process modeling, new or improved paradigms, or combinations of such, are needed.

We consider the following 6 themes the most interesting for future research in process modeling and management:

1. The basic mechanisms for **Modeling and Enactment** of software development activities. Such a mechanism could be either formal or informal.

2. **Coverage** of software life-cycle (requirements specification, design, implementation, verification, installation, operational support, documentation and software maintenance) and process entities (software products, software development activities or tasks, tools, and humans with roles).

3. **Structuring Mechanisms** for the process model (similar to the product model), e.g. by project-level *aggregation* of process types or sub-models. Such structuring mechanisms influence the next theme.

4. **Customization and evolution** of the process model (basic activation rules, guiding policies, tools, roles etc.). This is needed to avoid strait-jacketing effects. Aside from instance-level variability of product and processes, we can identify four type- or model-level customization mechanisms, all with different binding (i.e. evolution) characteristics: *subtyping* (project-level, static), *versioning* (transaction-level, dynamic), *parameterization* (in various granularity, semi-dynamic), and *delegation* (usually to embedding project, dynamic). Their suitability depends on the generic modeling paradigm and its structuring mechanism, the nature of the project and the target software, and what model aspects that need to be customized.

5. **Multi-actor, distributed** environment. E.g. cooperative transactions must be modeled to cope with general change propagation.

6. Integration with a **versioned** DBMS and its CM (Configuration Management) system for managing evolving products, i.e. CM + PM. That is, CM needs PM to express change to evolving products, while the PM information itself is an evolving “product”.

In the next section we will summarize and compare 6 different software process modeling paradigms and associated PM systems. The paradigms are: active databases (DBs), AI rules, task nets, process programming, contracts, and hybrids. The discussion is centered around the central themes given above. EPOS, the PM system we have been working on is described in more detail in the last two sections.

## 2 Process Modeling Paradigms

### 2.1 The Active DB Paradigm

Here the underlying DBMS embeds *Event-Condition-Action (ECA) rules* which model the software development activities. When the *Event* (a DB operation performed by an application task) occurs, the *Condition* (DB queries) is evaluated: if the *Condition* is
satisfied, the Action (a program which may fire other rules) is executed. Such an active DB can be regarded as a rudimentary PM system aimed at short transactions and low-level (programming) tasks. It is actor-oriented towards individual application tasks and busy propagation. There is good locality in expressing and matching rules.

**Problems:** ECA rules cannot easily express long, delayed or privileged operations (via notifiers?), high-level project tasks executed by humans, error handling, general scheduling, or creation of an extra derivation graph. Versioned DB support and CM coupling depend on the capacity of the underlying DB. Project customization depends on the rule structuring, and rule evolution may be difficult to model and manage.

*Adele* [?] and its successor Nomade [?] is a software system where the programmable triggers are associated with normal DB operations, described in object or relationship types. This models the software development activities and controls change propagation within one configuration. The types are clustered in project-specific partitions, which offer dynamic binding of information.

The OIKOS DB interfaces a logic deductive DB, and allows to uniformly store the product and the process structure. A deductive query or rule is different from an ordinary database query, since it is able to deduct new facts from those explicitly stored. The evaluation of the database is done both on request (lazy) by means of query invocations, and automatically when operations bounded to constraints are invoked (busy).

### 2.2 The AI Paradigm

AI techniques found in rule-based systems have been applied to software process modeling to express, plan, enact and control the software development activities. This paradigm exploits elaborate AI techniques (e.g. planning or central blackboard structure), both in process modeling and in interpretation of the models.

There is a rather unstructured set of PRE-CODE-POST rules, that operates on a common blackboard. Both static reasoning/planning and active triggering can be expressed, i.e. implicit sequencing. It is reactor-oriented towards a rule interpreter task. CODE execution is done by the interpreter or by delegated/FORK’ed executor tasks (fine-grained and delayed task creation).

**Problems:** classical AI rules have the same difficulty in expressing long transactions as ECA rules in active DBs. The scheduling and sequencing is also indeterminate: the same fact can trigger several rules or several facts can trigger the same rule (although partial ordering can be enforced by more complex PREs). There is poor locality in expressing and finding the rule, but the rule base is invariant to changes in the product structure. Also, AI rules have no Input/Output specifications which are essential for software development tasks. Project customization can be done by simple rule grouping, often through some subproject environment. AI techniques alone cannot fully support cooperative transactions, DB versioning or CM coupling. Process evolution can be supported by replanning and re-execution.

*MARVEL* [?] is one of the earliest rule-based PM system providing forward and backward chaining of menial software development activities. Project customization is done by replacing the rule set, called a strategy. Originally it had only non-versioned DB support, hence no coupling with any CM systems. But recently the work is being extended to a multi-user environment in MARVEL 3.0.
OIKOS [?] uses a hierarchy of blackboards to model and enact software process. Facts in a blackboard may trigger agents connected to the blackboard. An agent has a theory (a set of ECA triggers), which can be regarded as its type. OIKOS may activate as many agents as dynamically needed. There are problems in modeling a C compiler with an ESP agent because every time a compilation is needed, a new agent has to be activated. The emphasis is put on concurrent and distributed PM. No static planning is yet available, and dynamic enactment is non-deterministic without backtracking.

ALF [?] aims at comprehensive process modeling and exploits a knowledge-based approach. ALF is also a PM system which addresses the project customization issues in a most explicit and detailed manner. The formal, generic process model MASP has 6 components: object model (product model), operator types, logical expressions (used in some other components), orderings, rules, and characteristics. When MASP is customized into a project-specific process IMASP, each of these components is instantiated. For example, the objects belonging to this IMASP are identified, operator types are linked to proper tools, and other components are instantiated by replacing their formal parameters. Here the basic customization mechanism is parameterization, but the parameter categories are fixed. An IMASP may be further instantiated by the ALF system to yield an ASP - a real process for developing a particular software.

2.3 The Graph/Net Paradigm

This approach observes that a software process is similar to a real time system. Techniques for real time systems specification such as Petri Nets have been adapted and extended to model software process, particularly w.r.t. dynamic triggering and concurrency. The main advantages of the paradigm are that it can be given some formal and visual treatment.

The semantic model describes the syntax and semantics of the Graph/Net. There is no generic process model, and a project-specific model is a particular Graph/Net. A concrete process is given by a marked Graph/Net. Process execution is expressed by transition firing. Type-level information is expressed by a Graph/Net (a net type), and a marked Graph/Net will be coupled to actual product instances (a net instance). The Petri net model supports modeling of horizontal (sequential) chaining of activities, but not hierarchical (vertical) decomposition. It is reactor-oriented towards a set of deamon tasks, which can be concurrently executed by a mixture of tools and humans.

Problems: a task network denoting e.g. a derivation graph is pre-made, but at what granularity, created by whom and when? A task network (e.g. a derivation graph) must be updated upon changes in the product structure. The facility for process structuring (hierarchically) is limited, and process customization and evolution is difficult. More flexible binding mechanisms are lacking. As in the AI paradigm, versioning and CM support cannot be expected from pure Petri Net systems. On the other hand, rule-based techniques can be integrated to assist the user in automatically building the task network.

MELMAC/MSP [?] [?] offers two levels for software process description. At the higher level it offers different specialized views, i.e. object type and activity view, process view, project management view etc. At the lowest level, all the process entities are expressed and executed by FUNSOFT nets. A FUNSOFT net consists of a Petri net (with Places or channels, Transitions or agencies, and Edges), a job set, a type set, a predicate set, a modification point set, and an initial marking. Products are expressed as channels, tasks as agencies, with edges for I/O relationships between tasks and products. Agencies can have
attributes in the form of modification point refinements to cope with process breakdown and evolution. Roles and their access rights can be expressed as attributes of agencies. Temporal conditions are simulated by subnets which guard agencies.

**DesignNet** [?] proposes a formal process model combining AND/OR version graph and Petri Nets notation to improve structuring. Using this model, some concepts measuring the quality of the management of a software project can be formally defined and computed efficiently. The result can be used to assess and improve the management of software projects.

Ghezzi et al. present a formal theoretic approach to software process and management [?] that is based on another extension to Petri net, the **E/F Net**.

The **Entity Process Model** (EPM) [?] describes the entities involved in the software process, instead of focusing on the developing activities acting on those entities. Each entity is modeled by its possible states, and each state is either atomic or complex. State transitions occur when given conditions arise. A complex state can be hierarchically expressed as either the XOR or XAND of substates. XAND substates may evolve in parallel. An EPM can be mapped on a State charts [?] and executed. There is no DB coupling.

Note that all the three paradigms above use rules, but in different forms and ways; see comparison in Sec. ??.

### 2.4 The Process Programming Paradigm

*Process programming* means to define the process by a “meta-program” in an abstract process programming language. Software development is thus the execution of this meta-program.

In this paradigm, the semantic model is the process programming language, usually an extended procedural programming language. There is hardly a generic model, and the project-specific model is defined by a program in the language. If each program unit in the language has a separate specification and body (like in Ada), some project policy such as delayed operations can be encoded in the body without affecting the abstract specification.

**Problems:** A general-purpose programming language seems too heavy for process modeling. In addition a traditional procedural language is insufficient for process modeling, since the program has to be statically known before software process enactment. Versioning, DB support and version process knowledge may be hard to express.

**Arcadia** [?] has a prototype process programming language APPL/A [?]. This extends Ada with “active” (operation-triggered), derivative relations between software objects, triggers propagating updates from one relation to another, optional-enforlicable predicates on relations, and composite statements that provide transaction functionality. However, the support for process change (new tools, new policy, new environments) is weak and depends on intricate APPL/A programming. There is no subtyping.

**IPSE 2.5** [?] has a process programming language PML, which is an imperative, concurrent language with very complex, project-specific run-time semantics.
2.5 The Dynamic Contract Paradigm

ISTAR [?] is a project support environment taking a contractual approach to process modeling. No other published PM system of the same approach is known. This approach views every task in a software project as a contract between a contractor and a client. A contract specification is given in terms of a DB view with input and output documents, and includes the specified deliverables, acceptance tests, schedules, reporting requirements and working policies. The whole software process is viewed as a hierarchy of contracts, dynamically created by the developers (being the contractors and/or clients). ISTAR utilizes that the software process is driven by the product structure, a well-known in other fields of engineering.

Problems: There is no formal modeling of the software process. ISTAR knows little of the nature of contracts, and is reduced to the management of contractor/client protocol.

2.6 The Hybrid Paradigm

As we have already hinted, no single paradigm can satisfy all the requirements of process modeling. For example, many PM systems employ rules, but no single paradigm can satisfactorily cope with the policy aspects of rule triggering.

Some of the presented systems are actually multi-paradigm, though we classify them according to their main paradigm. Some PM systems even claim paradigm merging as their purpose. E.g. SPECTMEN [?] tries to merge FUNSOFT nets and a rule-based process modeling language MERLIN to design its own process modeling language PML. In EPOS, we have designed and prototyped a PM system which merges different paradigms and has a tight coupling with the EPOS CM system.

2.7 Evaluation of the Paradigms

Figure ?? sums up the PM paradigms and systems described previously, and compare them with respect to the 6 themes in the introduction. We put our EPOS system as the last item in the table.

The modeling and enactment of software development activities or tasks are the central issue in any PM system. In the previous subsections we have distinguished between active DBs with ECA triggers, implicit AI rules, and explicit task network. Figure ?? gives a pictorial summary of those discussions. All three paradigms operate with a shared DB or blackboard, surrounded by tasks that execute PRE-CODE-POST rules. The main differences lie in rule structuring, which tasks are executing which rules, and how the tasks are created and managed.

Note, that general tasks and derivation graphs cannot be handled by active relationships, serving as function steps. The reasons are: 1) Tasks of a certain duration and complexity cannot be effectively modeled this way. 2) An active derivation graph is fundamentally different from a static dependency graph, since domain-specific closures must be formed, empty derived objects (placeholders) generated etc. That is, we have to augment the primary product structure with generating secondary task structures, which must be maintained upon changes in the product structure.

Lastly, all paradigms have problems with tool modeling: Type/rule-level program spec-
Figure 1: PM System Summary and Comparison
ifications (CODE) often refer to instance-level objects (e.g., a CC compiler) by symbolic names. Such implicit relationships must be bound upon enactment. We must also express tool aggregation and bootstrapping (i.e., versioning).

3 EPOS Approach to PM

3.1 EPOS Background

EPOS [?] is an instrumentable, kernel software engineering environment covering both PM and CM. The core of EPOS is a client-server based EPOSDB which implements Change Oriented Versioning (COV) [?], and offers a version-transparent DBMS interface. A long and possibly nested transaction is connected to a change job, modeled as a project task. Its configuration (a non-versioned sub-DB) is evaluated inside EPOSDB, and checked-out (i.e., converted) to a workspace which contains:

1. Files and their dependencies: the Product Structure, PS.
2. A task network (an instantiated software process) to support PM.
3. Some control information in a Project Knowledge Base, KB, incl. a subset of types.

3.2 Rationale and Status of EPOS PM

The goals of EPOS PM have been five-fold: to couple PM and multi-actor CM via a versioned DBMS, to have a hybrid though compact PM model, to support automatic reasoning and planning, to offer project-level structuring, and to offer flexible mechanisms for PM customization.

EPOS PM has been designed and prototyped in 5000 lines of SWI-Prolog and the PCE graphics package. It comprises an Execution Manager, a Planner, a simple user interface, and a minimum DB interface. It covers the whole life-cycle of software development and various topics in software processes. Several non-trivial examples are running. The execution time is almost instantaneous after workspace initialization. The user- and DB interface takes most of the time, anyway.

We shall present the main features of EPOS PM, and relate these to the 6 themes given in the introduction.

3.3 EPOS PM: A Hybrid Process Paradigm

(Theme 2 and 3 from the introduction.)

An EPOS PM task instance (a task for short) has actual input/output parameters (product instances), defining a partial ordering between tasks. High-level tasks can be decomposed into simpler subtasks. EPOS PM tasks cover four kinds of software processes: short DBMS-triggered operations, simple derivation steps to run OS tools, general negotiation and change propagation involving humans, and long-lived (sub)projects that control DB transactions. EPOS PM also facilitates static planning to generate task (sub)networks, which later can be dynamically triggered.

Thus, EPOS combines the two major software process paradigms – AI (static rule-based reasoning) and Graph/Net (dynamic triggering of task network). For this purpose EPOS has three kinds of rules:

- Meta Rules to express general project and product knowledge, and heuristics to guide the Planner;
- Static Rules as a part of the KB for the Planner;
- Dynamic Rules to guide both low-level triggering and high-level, project-specific scheduling. The low-level rules represent sufficient conditions for activation (temporal rules, such as timestamps to regulate CC compilations), while the high-level ones represent necessary conditions (non-temporal rules, such as compile when or with what switches?).

See Sec. ?? on behavioral properties of task types in EPOS-OOER to express the Static and Dynamic Rules.

We can also identify other modeling paradigms in EPOS PM. For example, an EPOS task type has CODE to express process programming. And task decomposition within a single transaction and between subtransactions (project = transaction) resemble the contractual approach.
The different paradigms are combined into a coherent whole. Process model *interpretation* (planning, decomposition, enactment, and control) on the instance-level is guided by the type-level information, which can be customized to the project. Figure ?? summarizes the architecture of EPOS PM.

Figure 3: EPOS PM Architecture

3.4 The EPOS-OOER Semantic Data Model and its Expressiveness

(Theme 1 from the introduction.)

The EPOS PM formalism relies on the EPOS-OOER (Object-Oriented Entity-Relationship) semantic data model, which handles *Entity* and *Relationship types* in a uniform way. It incorporates Object-Oriented concepts such as subtyping with multiple inheritance, object identity, composite objects, and explicit type objects. EPOS-OOER is used to express product, task, and relationship types, and subtypes of these. It can also express types for other elements of the software process, such as tools and human resources.

*Product* and associated relationship types are used to model the software product. An instantiation of such a product model is called a *Product Structure*. A *task* type models a single software development step or activity, and the available task types constitute a complete process model. All types have ATTRIBUTES declarations, and can inherit
attributes from their supertypes. Task types have extra behavioral properties with special inheritance rules. In section ?? we will describe these properties and show how EPOS PM interprets them – or the other way round, how these properties guide EPOS PM system.

Because of the strong expressiveness and the interpretability of EPOS-OOER data model, EPOS PM covers all software process elements and the whole life cycle of software development.

3.5 The Process Model Interpretation

(Theme 1 from the introduction.)

We will describe the behavioral properties in task types, and show how the integrated EPOS Execution Manager and Planner [?] utilize these to interpret process model at instance-level:

- **Static PRE/POST-conditions** specify the functionality of each task type, and FORMALS specify the input/output constraints to the task type. These properties decide the shape of the instantiated task network which models the software process.

  The Planner reasons on the knowledge expressed by these properties and automatically creates the task network. The Planner is a non-linear AI planner adapted to our PM domain. The domain-specific issues include: reasoning on PRE/POST matching as well as on FORMALS specification; searching the current Product Structure to induce subgoals (tactical planning); and representing the generated plan as a task network.

- **CODE** is the imperative program performing the task functionality, which is triggered by the **dynamic PRE-conditions**. The Execution Manager executes the task network, controlled by the dynamic PREs, encompassing both low- and high-level rules. Some tasks in the network can be executed in parallel, and the execution of each task can be busy, opportunistic, or lazy (goal-directed).

- **DECOMPOSITION** specifies how to decompose a high-level task into a set of subtasks. A repertoire of candidate subtask types is attached to the high-level task type. The Execution Manager and Planner cooperate to decompose a parent task into an executing subplan of children tasks, taking the repertoire as the search space. Thus, AI like hierarchical planning and PM like task structuring is unified.

3.6 PM+CM and Cooperative Transactions

(Theme 5 and 6 from the introduction.)

Unlike most PM systems working on a single-version product, EPOS PM is tightly coupled to EPOS CM. Both intra- and inter-transaction\(^2\) negotiation and change propagation are uniformly modeled. On one hand, CM uses PM to describe and propagate changes in evolving products in the context of cooperating, non-serializable EPOSDB transactions. On the other hand, the PM information itself (task types, task networks, project information like tools, users, change requests and other CM items etc.) can be treated as explicit objects, stored in and uniformly versioned by EPOSDB (??).

\(^2\)Inter-transaction PM is still under implementation, using a message passing facility [?].
3.7 Project Customization and Process Evolution

(Theme 4 from the introduction.)

Referring to the binding chain mentioned in the beginning of this paper, we can summarize the project customization mechanisms of EPOS PM as follows: EPOS-OOER → system-defined EPOSDB schema → project-specific schema → task network.

The EPOS-OOER semantic data model is used to express the system-defined EPOSDB schema (common product types, task types, and connecting relationship types). This is a very generic and incomplete process model.

The gradual binding of a generic to a project-specific model means that the schema is refined in the context of projects and subprojects. Current customization mechanisms are: subtyping (with special inheritance rules), type versioning of task properties (parameterization with respect to versioning attributes), and dynamic look-up of project-level information. Work is under way to expand EPOS-OOER with formal subschemas and dynamic delegation of behavioral properties (typically along the project hierarchy).

Finally, a partially ordered task network (a plan) is generated by EPOS Planner and executed by the Execution Manager. This network represents the concrete software process (cf. section ??).

Process Evolution is achieved by the customization mechanisms mentioned earlier. Upon changes in the product, replanning and re-execution (including error-handling) is used to maintain system consistency. Incremental replanning is implemented to deal with minor and localized change to avoid major replanning.

4 Future Research

EPOS PM addresses the 6 important themes in process modeling mentioned in the introduction. The preliminary experience gained so far is encouraging. However, these themes should be pursued further, much more deeply and broadly. In the context of EPOS, the following issues are to be investigated:

- **Better integration of EPOSDB and EPOS PM** to fully realize CM+PM. E.g., an operating process could be stored ("learned") as a task network template in the database for later reuse. It is easier to experiment at the instance-level, before abstracting to the type-level (also a CAD experience). Task networks may also be versioned to reflect minor variations.

- **Better meta-user support** to specify and evolve project-specific process models. Especially, the meta-user should be able to validate his model.

- **Better understanding and representation of meta-rules**, e.g., to instrument the building of derivation graphs in the PS domain.

- **Refining the task formalism**. For example, the inheritance rules for task properties should be validated and elaborated further.

- **Subschemas and delegation** for project-level customization and evolution.

- **Better support of process evolution**. So far we have only implemented replanning on PS changes.
• Better modeling of multi-actor systems. Our work on cooperative transactions should be extended by AI planning to coordinate subjobs and to evolve the version space.

Acknowledgments: Cordial thanks to the entire EPOS team.

References


