Software Process Planning and Execution: Coupling vs. Integration

Chunnian Liu*
Norwegian Institute of Technology (NTH), Trondheim, Norway

Abstract

EPOS[1] is a kernel Software Engineering Environment integrating Software Configuration Management and Software Process Management. EPOS has a software process planner, based on the EPOS-OOER data model, and working on a versioned software engineering database – *EPOSDB* with an original *Change-Oriented Versioning (COV)* paradigm. The EPOS Planner applies non-linear and hierarchical planning techniques to Software Process Management. This paper describes the design, implementation and preliminary experience of the EPOS Planner. Specially, we present and discuss two different methods to combine planning and execution of software processes: coupling and integration. The former makes a good compromise between static reasoning and dynamic execution/triggering; while the latter provides a new solution to modeling and automation of software development iteration and replanning.

Keywords: AI for Software, Planning, Software Process Management.

1 Introduction

EPOS [C*89] [COWL90] is an instrumentable kernel Software Engineering Environment (SEE), supporting both Software Configuration Management (CM) and Software Process Management (PM). EPOS CM and EPOS PM use the common EPOS-OOER semantic data model, describing both software products and software processes in an OO style. The core of EPOS is a central EPOSDB, coupled with checked-out, project-specific workspaces.

*Detailed address: Div. of IDT, N-7034 Trondheim-NTH, Norway. Email: liu@idt.unit.no, Fax: +47 7 594466, Phone: +47 7 594483. The author is on leave from Beijing Polytechnic University, P. R. China, supported by an NTNF Postdoctoral Fellowship Program of Norway (ST.60.61.221277).

[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 EPOS [Lem86].

The EPOSDB is a *versioned* software engineering database to store objects and relationships produced in software evolution. It applies *Change-Oriented Versioning (COV)*, which resembles and generalizes conditional compilation. COV keeps the versioning orthogonal to the data model and the product (module) structure, and gives better support of cooperative work than the traditional Version-Oriented Versioning (VOV).

A detailed description of COV stands in [LCD*89]. Here we present only some basic (and simplified) concepts. In COV, each *functional change* in the product is represented by a global boolean variable, called an *option*. Each *fragment* of the database (a line of a source file, or a relational tuple) is tagged by a logical expression, a *visibility*, of options. A version of the DB is described by an interpretation of these options, and consists of all fragments whose visibilities have the value `true` under the current interpretation. An interpretation is called a *version-choice*, expressed as a set of option bindings: `[option, true/false]`. (see section 5 for more).

An EPOSDB transaction to update a configuration is described by a *config-descr:* <*version-descr, product-descr*>. A version of the DB is first derived from the *version-descr*, and the *product-descr* is then evaluated to a *Product Structure (PS)*, a configuration, within this version. Configurations are checked-out from the central EPOSDB into local workspaces, so that EPOS PM can work on and coordinate them. (In section 3.3, we will give a scenario of serialized transactions). Figure 1 shows the overall structure of EPOS.

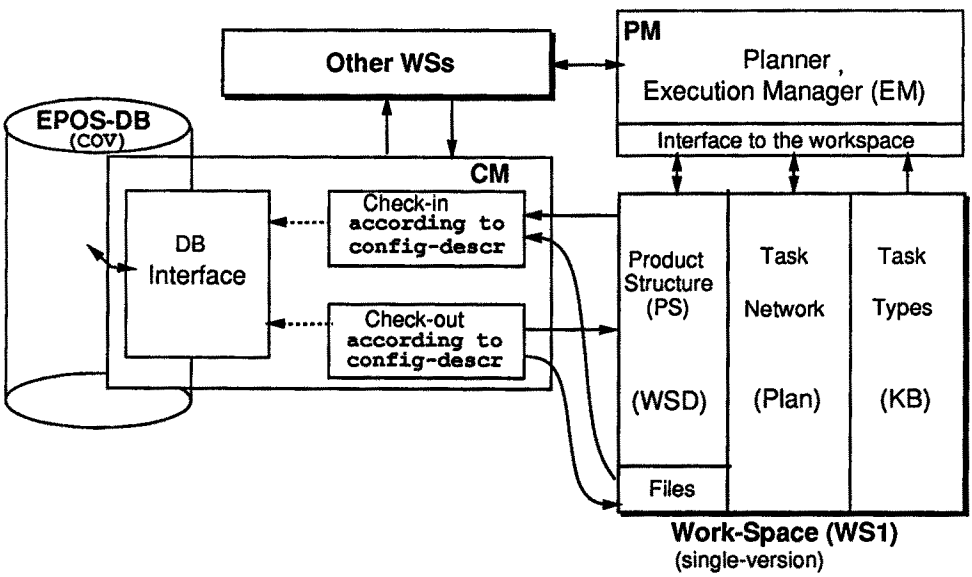

Figure 1: The Overall Structure of EPOS

EPOS PM intends to cover the following items of software processes: deriver tools as well as human actors; life cycle phases in software development as well as low-level derivation graphs; work decomposition (subtasking); project-specific policies for change propagation, user coordination and communication, and other work procedures. EPOS PM has a Planner to fetch, reason about, modify and store project-specific knowledge about the whole life cycle of software development.

The EPOS-OOER describes the software product model (by *CM types* or *product types*). A Product Structure (PS) is an instantiation of the product model and serves as the *World State Description (WSD)* for the EPOS Planner. The EPOS-OOER also describes the software process model (by *PM types* or *task types*), which serve as the *Knowledge Base (KB)* for the Planner. A generated plan is an instantiated *task-network* within a workspace (Figure 1).

The EPOS Planner applies several up-to-date AI techniques to the Software PM area: **Non-linear planning** to support parallel processing; **Hierarchical planning** to deal with domain complexity; **Integration of planning and execution** to automate iteration and replanning.

In this paper we describe and discuss two different methods used in the EPOS Planner to combine software process planning and execution, sketched in Figure 2.

In the **coupling method** (Figure 2(a)), the EPOS Planner has three layers. The inner layer is a domain-independent non-linear planner (section 3.1) which has been tested on both robot and PM domains. The middle layer deals with PM-specific issues described mainly in section 2.2. The outer layer interacts with another PM module – the Execution Manager (EM) to achieve hierarchical planning (*i.e.* task decomposition, see section 3.2). This method represents a good compromise between static reasoning and dynamic execution/triggering. But exception handling (*e.g.* iterations) and replanning could be difficult or messy.

In the **integration method** (Figure 2(b)), the inner layer is extended to cover activities such as task decomposition, task execution and monitoring, besides basic planning. As a result, the outer layer is removed together with the EM. All these activities are described and activated in a uniform mechanism, so they can be interleaved at a very fine granularity. This gives us the opportunity to automate iterations and replanning, being the key point to introduce the integration method (section 4).

The ensuing sections of this paper are as follows. In section 2 we summarize the EPOS-OOER, and show how it is used to describe the WSD and KB for the Planner. Section 3 presents the Planner and the EM as two separate mechanisms and shows how they cooperate, illustrated by an implemented scenario. The integration of planning and execution is described in section 4. The preliminary experience and some ideas on future work (*case-based planning* and *multi-workspace cooperation*) are briefly discussed in the concluding section 5.

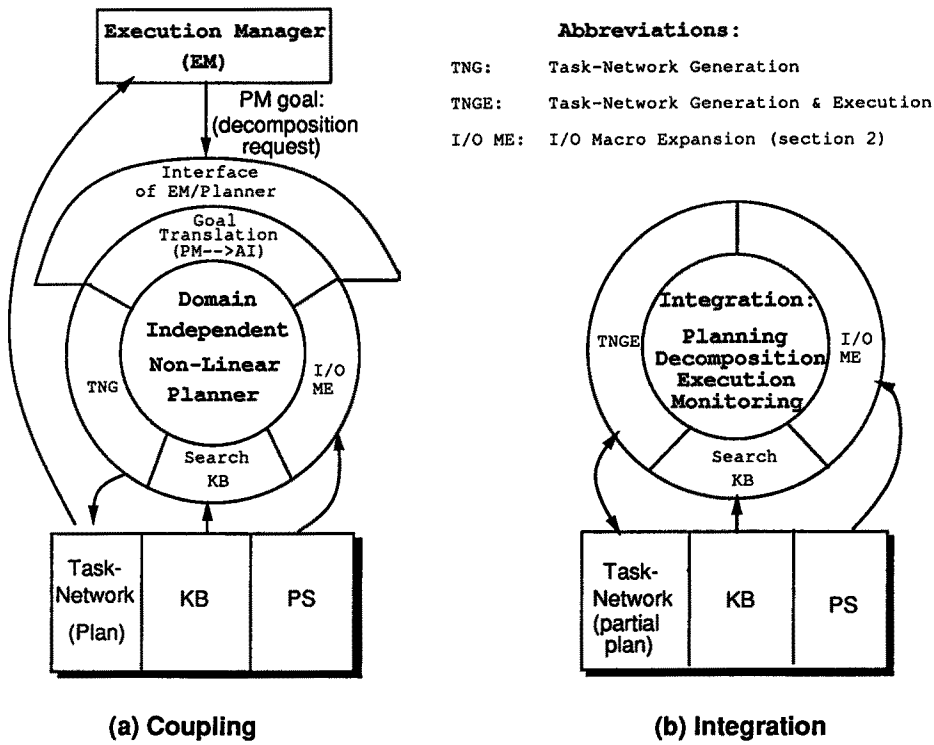

Figure 2: Two Methods to Combine Planning and Execution

2 The WSD and KB Specified in EPOS-OOER

An AI planner needs a World State Description (WSD) and a Knowledge Base (KB). In this section we explain how the EPOS-OOER data model describes these fundamental and domain-dependent constituents of the EPOS Planner.

The EPOS-OOER data model handles *entity* types and *relation* types in a uniform way. It incorporates Object-Oriented concepts such as subtyping, multiple inheritance, and type descriptors (representing types as objects in the instance level, see section 2.2). There are two kinds of types – *CM types* and *PM types*, for passive and active objects, respectively. Figure 3 shows the (simplified) type hierarchy (the *schema* of the EPOSDB) used in our current implementation of EPOS. All these types (and associated annotations in the figure) will be used and explained later.

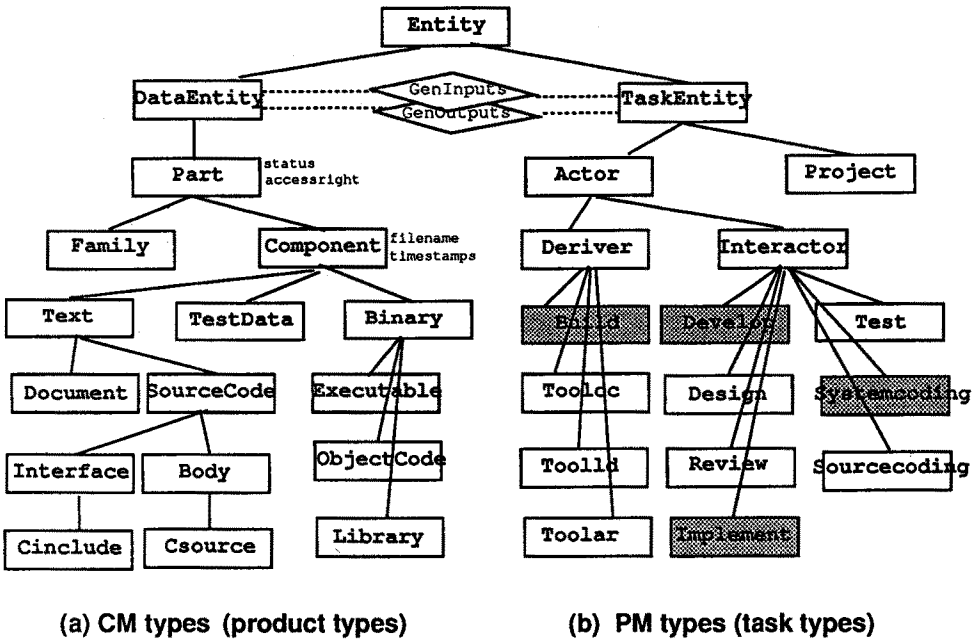

Figure 3: The Type Hierarchy of EPOS schema

2.1 CM types and WSD

The CM types are subtypes of `DataEntity`, and describe software products by families (subsystems) and components (files). These, together with connecting relation types, describe the software product model. The following relation types are used:

```
Composition(Family,Part)              %% Part belongs to Family
  FamilyOf(Family,Family)             %% SubFamily breakdown
  InterfaceOf(Family,Interface)       %% Self-explanatory
ImplementedBy(Interface,Body)         %% Interface is implemented by Body
SourceImports(SourceCode,Interface)   %% SourceCode imports Interface
```

An instantiation of the product model is called a *Product Structure (PS)*. It is a product closure, defined by the hierarchical product model, within a particular version of the EPOSDB. A PS is checked-out into a workspace as ordinary directories and files, so programming tools such as compilers can work on them. Our current implementation of EPOS PM is in Prolog. In order to speed up PM work, instances of `DataEntity` subtypes and above relation types, which defines the PS, are also "checked-out" into the workspace as a set of Prolog facts. This constitutes the *World State Description*

(WSD) for the EPOS Planner. Whenever the Planner checks if a (sub)goal is satisfied by the WSD, it searches this cache of Prolog facts. (Of course, the Planner may search the EPOSDB directly, through the DB interface – a set of Prolog predicates implemented in C). Figure 4 (scenario) in section 3.3 shows a graphic representation of a sample PS.

All types can have ATTRIBUTE declarations, and inherit attributes from their supertypes. In Figure 3 attributes are indicated by text annotations in small font for the `Part` and `Component` types. For example, attribute `status` describes the current processing status of a software part. All subtypes of `Part` (`Family`, `Component` and their subtypes) inherit this attribute. The value range of this attribute is the enumerated domain `[created, designed, reviewed, coded, tested, ...]`.

2.2 PM types and KB

The PM types (or *task types*) are subtypes of `TaskEntity`. Their instances (tasks) represent software processes, *e.g.* design activity, editing or compilation, which are the *actions* in terms of AI planning. Each task type specifies an action rule, and the sum of declared task types constitute the *Knowledge Base (KB)* for the EPOS Planner. A generated plan is a partially ordered set of task instances, called a **task-network**.

In addition to ATTRIBUTE declarations, task types have the following **type properties** to support planning and execution:

1. **FORMALS** to express the *formal parameters* of the task type. FORMALS specifies constraints (type, number, status) on the actual parameters, expressed by the `GenInputs` and `GenOutputs` relationships (see Figure 1). For example, FORMALS for task type `Toolcc` (C compiler) is:

   ```
   FORMALS Csource(coded) * $Cinclude(coded) -> ObjectCode(created)
   ```

   This specifies that a `Toolcc` task takes one "coded" `Csource` instance and zero or more "coded" `Cinclude` instances as Input, and one "created" `ObjectCode` instance as Output. We will see later in this subsection how the Planner uses the FORMALS property in reasoning and building of the actual parameters.

2. **DECOMPOSITION** to describe legal *subtasking*. Instances of "high-level" task types (expressed as shadowed boxes in Figure 1) should be decomposed into a network of *subtasks*. DECOMPOSITION specifies the candidate types for the subtasks. For example:

   ```
   DECOM repertoire([toolcc,toolar,toolld])        %% for Build type
   DECOM repertoire([design,review,implement,test])%% for Develop type
   ```

   The first line specifies that a `Build` task should be decomposed into a (sub)plan (a task-network) built from `Toolcc`, `Tollar` and `Toolld` tasks (C compiler, library-maker and linker). The exact shape of this (sub)plan is the result of the planning

process when the Planner is called by the Execution Manager to decompose a particular Build task. The point here is that, when the Planner makes the (sub)plan to achieve the effect of Build, it searches the pool of the task types in the above repertoire, rather than the entire KB. For the second line above we have a similar situation for the Develop task type. Section 3.2 and 3.3 will detail how the DECOMPOSITION property is used.

Note: FORMALS and DECOMPOSITION specify relationships between *types*, in contrast to relationships between *instances* of types. The EPOS-OOER introduces a special TypeDescr type (not shown in Figure 3). For each type T, we create an instance of TypeDescr, called the *Type Descriptor (TD)* of T to represent T at the instance level. Type relationships can then be implemented by relationships between *TD*s.

3. **PRE-** and **POST**-conditions. They are declarative specification of the task, around the CODE part of the task. The EPOS Planner reasons on PRE/POST chains of tasks as traditional AI planners do. PRE/POST are $wffs$ (well-formed formulas) in First Order Predicate Logic, with similar restrictions as in the classical STRIPS. The PRE/POST of task A will be written as $PRE(A)/POST(A)$.

   A large part of the PRE/POST, concerning constraints on Input/Output of the task type, has been specified implicitly by FORMALS. On the other hand, the detailed form (as conjunctions of literals) for this part of the PRE/POST cannot be written down at the type level. For example, zero or more Cinclude files may be a part of the Input to a Toolcc task. However, we do not know the exact number until the Planner tries to select Toolcc to achieve a particular (sub)goal.

   Our strategy to deal with this PM-specific problem is to use the predicates *assert_input* and *assert_output* in PRE and POST, respectively, to represent constraints on actual I/O parameters. These two predicates can be considered as macros, at planning time they are expanded into conjunctions of literals, based on the current (sub)goal and the concrete Product Structure (PS). Therefore, we can reason on them as any other PRE/POST-conditions.

   For example, suppose that the Planner is trying to introduce a Toolcc task to satisfy the goal *thereis(m.o, objectcode, created)*. It first expands the macro *assert_output* in the POST-condition into a literal *thereis(AnyObj, objectcode, created)*, based on the Output constraints specified in FORMALS of Toolcc type (see above). The matching between the goal and this literal confirms the selection of Toolcc, with *AnyObj* bound to *m.o*.

   Next, the Planner expands *assert_input* in the PRE-condition to deduce the subgoals. This expansion is essentially to establish the actual Input to this Toolcc task, given the established actual Output *m.o*. It is more complicated, based on the Input constraints specified in FORMALS and the current Product Structure (PS). First, the "main" Input *m.c* is deduced from the Output *m.o*. Then the current PS is searched, starting from *m.c* and following the instances of ImplementedBy and SourceImports relations, to find all relevant " *.h" files which

are also a part of the Input to this Toolcc task. In our scenario (section 3.3), this expansion would find out that the total actual Input is *m.c*, *m.h* and *b.h* (cf. Figure 4). Then the subgoals for the Planner to work on is the conjunction:

$$thereis(m.c, c\_s, coded) \bigwedge thereis(m.h, c\_i, coded) \bigwedge thereis(b.h, c\_i, coded)$$

This planning-time expansion is task- or language-specific. Each task type has its own way of doing it. *E.g.* Deriver tasks for programming languages like C, Pascal and Fortran needs the *transitive closure* of the relevant Input, as we have shown above for Toolcc. This closure is not necessary for languages like Ada and Modula with separately compiled interfaces.

With the coupling method (section 3), planning and execution are treated as two different mechanisms, so a PRE-condition is split into two parts: **PRE_STATIC** used by the Planner, and **PRE_DYNAMIC** used by the Execution Manager (EM) to trigger execution of planned tasks. The dynamic part involves comparison of timestamps (*e.g*, a task may start only if its Input is newer than its Output), or temporal conditions (such as "something can be done only at night"). With the integration method (section 4), however, there is no need for such a split.

4. **CODE**. It is a *sequential* program performing real task actions upon task execution (*e.g.* to call the underlying programming tool). Essentially, CODE contains the imperative description of the task, while PRE/POST provide the declarative specification.

To sum up, we sketch the specification for two sample task types as follows:

```
ENTITY Toolcc IS Deriver{
ATTRIBUTES ...;
FORMALS    Csource(coded)    *
           $Cinclude(coded) ->
           ObjectCode(created);
DECOM none; %% an "atom" task
PRE   assert_input AND
      newer(In,Out);
CODE  call("cc -c In -o Out");
POST  assert_output      }
```

```
ENTITY Build IS Deriver{
ATTRIBUTES ...;
FORMALS    Family(coded) ->
           Executable(created)

DECOM repertoire(Toolcc,Toolar,Toolld);
PRE   assert_input AND
      at_night;
CODE  none; %% performed by subtasks!
POST  assert_output     }
```

Some final comments on *type inheritance rules*: ATTRIBUTES obey normal OO inheritance rules (bottom-up, left-right), while CODE inheritance is by concatenation using SIMULA's INNER mechanism. Therefore the inheritance rules for PRE/POST *etc.* are guided by Hoare's mechanism of program assertions.

3 Coupling of Planning and Execution

In the last section we have elaborated some important (PM) domain-specific issues which are dealt with in the middle layer of the EPOS Planner (cf. Figure 2 (a)). In this section, we describe the inner layer (a domain-independent non-linear planner) and the outer layer (coupling the Planner with the Execution Manager to achieve hierarchical planning). An implemented scenario follows to illustrate most of the relevant concepts and techniques.

3.1 Non-Linear Planning

Non-linear planning means that the produced plan is not sequential. Rather, it is a network of partially ordered task instances, so it can potentially be executed in parallel. The core of the EPOS Planner is a non-linear planner modeled as a formal **Production System**, similar to IPEM [AS88], [Amb87] and TWEAK[Cha87]. This formalism represents the state of the art in the AI planning area, and can quite easily be extended to cover the activities of plan execution and monitoring (see section 4).

According to this formalism, at any time during the planning, we have an *incomplete-plan*. The *production rules* are incomplete-plan transformers, modeling the planning activities. The germinal plan P_0 consists of two dummy tasks: **BEGIN** with the initial WSD as its POST, and **END** with the goal G as its PRE. P_0 is incomplete, containing many "flaws". At any time during the planning, a proper production rule may be applied to the current incomplete-plan P_i, fixing up some flaw, and transforming P_i to a new incomplete-plan P_{i+1}. This process will continue until we get a *flaw-free* plan P_n, which is the plan achieving the goal G.

The key concept in non-linear planning is the so-called *protection range* $q \longrightarrow p$, which links a conjunct q in $POST(A)$ and a conjunct p in $PRE(B)$ (assuming A is before B). The two conjuncts linked by a range are required to unify, and the unification is protected during planning (see below).

We need only two kinds of production rules (*Task Selection Rules* and *Task Ordering Rules*) to specify the planning activities. These rules are quite easy to understand and express in Prolog. The informal descriptions of these rules are as follows:

```
1. Task Ordering Rules:
   IF   there is a range q-->p (q in POST(A), p in PRE(B))
   THEN IF   there is C parallel to B and POST(C) undoes p
        THEN set B before C
        IF   there is C parallel to A and POST(C) undoes p
        THEN set C before A
        IF   C is between A and B and POST(C) undoes p
        THEN introduce a new task W between C and B to re-establish p
```

```
2. Task Selection Rules:
   IF   no range supports p in PRE(A)
   THEN IF   there is B before A and q in POST(B) supports p
        THEN set a new range q-->p
        IF   there is B parallel to A and q in POST(B) supports p
        THEN set a new range q-->p; set B before A
        IF   q in POST(B) supports p but B is not in the plan
        THEN introduce B between BEGIN and A; set a new range q-->p
```

The search space for this production system is the set of all possible incomplete-plans generated by the production rules starting from the the germinal plan. The goal of the search is a flaw-free plan. The search strategy is *best-first plus backtracking.* The *best-first* search is guided by a set of **heuristics** to decide which flaw should be fixed first; and for a flaw, which fix should be tried first.

It is well-known that *goal interactions* are the source of complexity in non-linear planning. The EPOS Planner has been tested on both robot and PM domains. In the former case, this planner is faster than *e.g.* IPEM (a factor of 5–10). In the latter case, because there are much fewer goal interactions in the PM domain, the EPOS Planner works quite efficiently with a set of simple heuristics. The scenario given in section 3.3 needs only several minutes to run on a Sun station, including planning, execution, calling OS tools, and graphics. Planning takes less than one third of the time, even though it is an AI process involving intensive searching and computation.

The heuristics used in the Planner is as follows. At each planning cycle, we first check possible range violations. If there are violations, use a proper Task Ordering Rule to fix one of them. Otherwise one of the Task Selection Rules is used to solve one of the unsupported PREs. Whenever an unsupported PRE can be fixed by existing tasks, we always use them to keep the number of task instances as small as possible. The order in which the unsupported PREs are considered and the order in which the candidate task types are tried are determined by more domain-dependent heuristics. Many of these heuristics can be expressed in Prolog simply by the order in which we write the production rules and the task types.

To complete the production system, we need a **Controller** to integrate heuristics and apply them effectively. It maintains an **agenda** (a priority queue) of items specifying all of the necessary planning activities for the current incomplete-plan. Items are in the form of pairs [flaw, fix] and ordered using heuristics. The *controlling algorithm* is informally described as:

```
1. Build germinal plan PO based on initial WSD SO and goal G;
2. Initiate the agenda AG by inspecting the current plan PO;
3. If   AG is empty          %% The current plan is flaw-free
   Then STOP;
4. If   AG(TOP) has no fix   %% A dead-end
   Then backtrack;           %% Automatically done by Prolog
```

```
5. Fix the flaw in AG(TOP); %% Current plan pi transformed to Pi+1
6. remove AG(TOP), and adjust AG by inspecting the new plan Pi+1;
7. GOTO 3.
```

Whenever a new task is introduced in the plan, both the task and its actual Input/Output parameters (of `DataEntity` subtypes) are instantiated, together with instances of connecting relation types `GenInputs` and `GenOutputs`. This instantiation process is also (PM) domain-specific and carried out in the middle layer of the Planner (see Figure 2 (a)). The resulting plan is a network of partially ordered task instances which may potentially be executed in parallel.

3.2 Hierarchical Planning

As the EPOS Planner deals with real world PM problems, a hierarchy of abstractions is essential. The process of alternatively adding detailed steps to the plan and actually executing some steps should continue until the goal is achieved. This is the *Hierarchical planning* technique. In our coupling method, hierarchical planning is accomplished by the cooperation of two different mechanisms – The Planner and the Execution Manager (EM). The interface between them is the outer layer of Figure 2 (a).

Of course the primary functionality of the EM is to execute the tasks of the plan generated by the Planner. The execution is guided by the partial order among the tasks set by the Planner (so some tasks can be carried out in parallel), and triggered by the PRE_DYNAMIC condition of each task. We extend this basic connection between the Planner and the EM to achieve hierarchical planning.

The EM and the Planner interact as follows. At the beginning, the EM instantiates a very "high-level" task, based on the user's request. The task could be of `Develop` type, for example, to develop an initial version of the software product. This single task can be regarded as the first coarse plan. When the EM tries to execute this "plan", it calls the Planner to decompose the "high-level" task into a more detailed (sub)plan. The process goes like this:

1. The EM sends a PM goal to the Planner. The PM goal consists of the the "high-level" task (the parent task) and its actual Input/Output, representing a decomposition request.

2. The Planner translates this PM goal into a set of AI goals – a conjunction of literals. This translation is (PM) domain-specific, and dealt with also in the middle layer of the Planner.

3. Then the Planner works to make a (sub)plan to achieve this set of AI goals. In doing so, it takes the current world state as its initial world state, and the repertoire specified in the DECOMPOSITION property of the parent task type as the (sub)KB to select useful tasks.

4. The generated (sub)plan is added to the original plan, with each task in the (sub)plan (called a *subtask*, which may be a "high-level" task again) linked to the parent task by an instance of the SubTask relation type.

5. Then the EM executes this (sub)plan to achieve the effects of the parent task.

Obviously, the above process can work in multi-levels: if the EM meets another "high-level" task when it executes the (sub)plan, it will call the Planner again. This strategy fully utilizes the desired hierarchical planning techniques. In the next subsection, a scenario in software PM will illustrate how this strategy works in the real world, and also how the EPOS Planner works on the real EPOSDB.

3.3 A Scenario

Assume that an EPOS user wants to develop an initial version of a software product. This is modeled as a `Family` f, with a tested `Executable` program *m.exe* as the (main)result. The COV option `Init` is introduced, representing this initial function of f. The *config-descr* for this transaction is <{[Init,true]}, f>, and the empty `Family` f is checked-out from EPOSDB as the start point of this transaction.

The first coarse plan consists of only one `Develop` task. The EM first makes sure that its PRE_DYNAMIC is O.K., and then tries to execute it. As `Develop` is a "high-level" task, the EM calls the the Planner to decompose it. The resulting (sub)plan is a sequence of `Design`, `Review`, `Implement` and `Test` tasks.

Suppose that the `Design` and `Review` tasks are carried out ("executed") by the user (a human actor), and that as the result of the executions of these two tasks, the PS for f is settled down as shown in Figure 4 (cf. section 2.1, also see Figure 5).

Now the EM tries to execute the `Implement` task, which also is "high-level". The Planner is called again, and the produced (sub)plan is the sequence of `Systemcoding` and `Build` tasks. Based on the Product Structure of f, `Systemcoding` is decomposed (recursively) into seven parallel `Sourcecoding` tasks for *m.c*, *m.h*, *a.h*, *a1.c*, *a2.c*, *b.h* and *b.c*. Each of these `Sourcecoding` tasks consists of an editing session which can be "executed" by the user with some interactive editor tool.

Then the EM tries to execute the `Build` task. It calls the Planner once again, and the produced (sub)plan consists of four `Toolcc` tasks to compile *m.c*, *b.c*, *a1.c*, *a2.c*, one `Toolar` task to make the `Library` *a.a*, and one `Toolld` task to link everything together. Figure 5 is automatically generated by the Planner so far, showing a fully developed plan.

Now the EM executes the (sub)plan for the `Build` task to produce *m.exe*, and then executes the `Test` task to complete the first version of the product f.

Note: before the `Design` and `Review` tasks have been executed, neither the `Systemcoding` nor the `Build` tasks can be decomposed properly, because they both depend on the

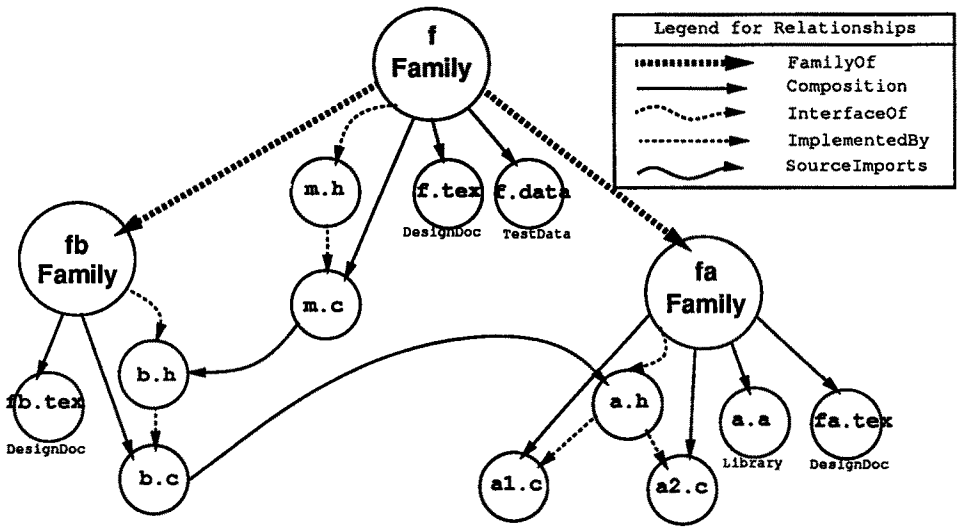

Figure 4: A Sample Product Structure (PS)

Product Structure (PS) which is the result of `Design` and `Review`. This example shows clearly that hierarchical planning is essential in the PM domain.

To conclude this transaction, the PS (including the real files) and the plan for `Develop` are checked-in to EPOSDB (the plan is stored for reuse, see below). This is the first version of f, specified by the version-choice {`[Init,true]`}.

Suppose that the end-user of this software product f requests some enhancement, then the EPOS user should start a new transaction. A new COV option `New1` is introduced to represent the additional functionality requested by the end-user. The *config-descr* should be <{`[Init,true]`, `[New1,true]`}, f>. Then, the PS for f and the plan for `Develop`, which were checked-in previously, are checked-out again into a new workspace. This time the "top" task is of the type `Change_Processing` (not shown in Figure 3), which is also a "high-level" one and therefore decomposed into a (sub)plan. A task of `Develop` type turns out as one of the subtask within this (sub)plan. Suppose there is no structural change in the PS, then the old plan for `Develop` will be **reused** to make a new version of the product f. (If there are structural changes, we need replanning for `Develop`, see next section). Finally, the new version of the product f, specified by the version-choice {`[Init,true]`, `[New1,true]`}, is checked-in for future evolution.

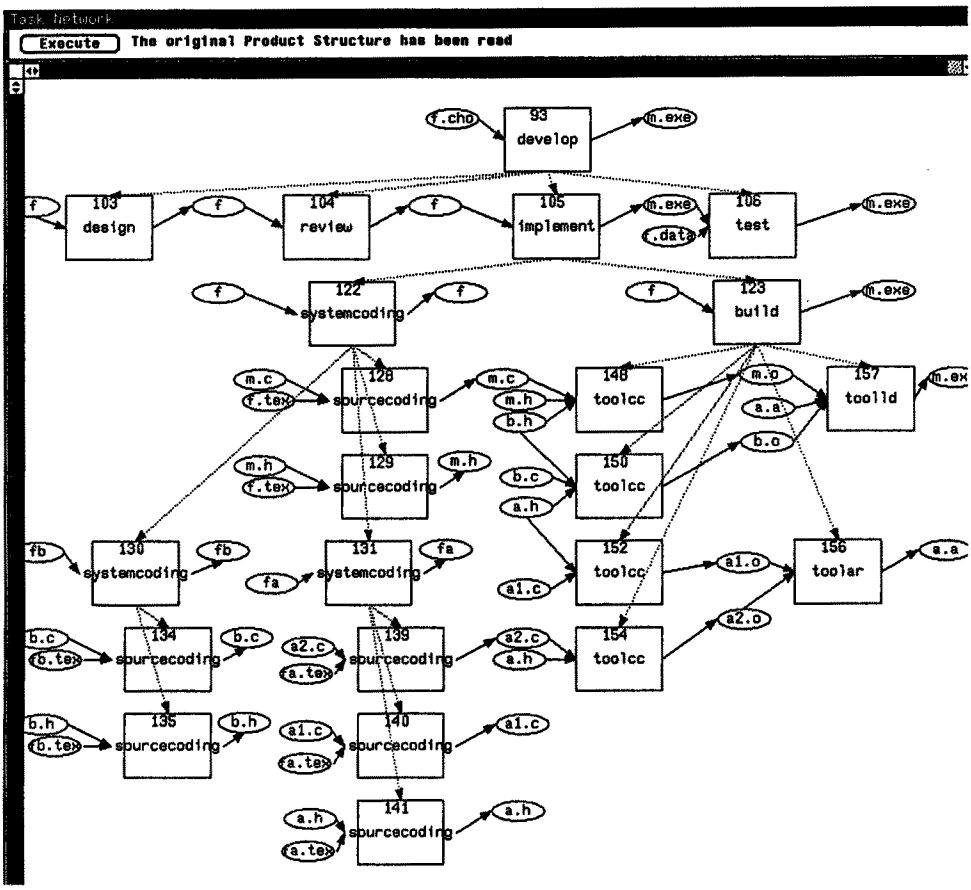

Figure 5: A Fully Developed Plan

4 Integration of Planning and Execution

As seen in the scenario, with the coupling method, the Planner and the EM (as two separate mechanisms) work cooperatively in hierarchical planning and plan execution. This is also a nice compromise between static reasoning (at planning time) and dynamic scheduling (at execution time – remember that the EM always checks PRE_DYNAMICs). However, there are several points to consider further:

- *expected exceptions.* A `Toolcc` task could achieve its effect (producing an object file) or fail (due to linguistic errors); a `Test` task could confirm the product or demand bug-fixing; etc. These "expected" exceptions cause *iterations* in software development, and are considered crucial in PM. In the coupling method, we have some ad hoc solution to this problem (*e.g.* CODE reports the execution failure to

the user, and the user points out which task in the network should be re-executed). The automation of iterations will give the user more intelligent assistance.

- *unexpected exceptions.* Hardware failures, file system overflow or other unexpected events can be only handled by execution monitoring and dynamic adjustment of the plan.

- *replanning.* Whenever the Product Structure (PS) or KB has some change, replanning is demanded. *E.g.* if an editing session changes some *#include* statement in *m.c*, the plan in Figure 5 may have to be changed accordingly. Here some kind of *incremental replanning* is very much desirable.

Inspired by IPEM [AS88], we think that integration of planning, execution, and monitoring would be a promising solution to the above problems. This means a uniform mechanism for all those activities, so we can interleave them at a very fine granularity. In the following we show how the set of production rules for non-linear planning (Task Ordering Rules and Task Selection Rules described in section 3.1) can be extended to cover plan execution and monitoring, as well as task decomposition (section 3.2). We will also see how this integration solves the *automatic iteration* and *incremental replanning* problems.

```
%% 1. 2. are Planning Rules (see section 3.1)

3. Execution Rule:
   IF   A is not "high-level" and PRE(A) is satisfied by current WSD
        and A is not involved in any range violation
   THEN execute A, and remove A when it times out (succeeds or not)

4. Decomposition Rule:
   IF   A is "high-level" and ready for execution
   THEN decompose A into a subplan

%% 5. 6. 7. are Monitoring Rules:

5. IF   A is being executed
   THEN a). for effect q that really appears, change the producer of
           ranges q-->p from A to BEGIN (i.e. put q into WSD);
        b). for failed effects, simply excise the relevant ranges
6. IF   there is some change in the Product Structure (as an exception)
   THEN redo I/O macro expansions (sec. 2.2) for all tasks in the plan
7. IF   some other exception (for example, ~q) occurs
   THEN delete q from WSD (excise all ranges q-->p starting from BEGIN)
```

With the new rules, exceptions (failed effects in Rule 5, PS changes in Rule 6, and others in Rule 7) always cause some new unsupported PREs, which sooner or later will

trigger some planning activities (Rule 1 and 2). That is, replanning is automatically triggered and based on the existing plan (rather than from scratch). This is the key point to introduce the integrated model.

For example, the Toolcc task #150 in the scenario of section 3.3 may fail to achieve its effect (*i.e.* producing *b.o*), due to some syntax errors in the source files. According to Rule 3, it is removed nevertheless when it times out. On the other hand, due to Rule 5 b), the subsequent Toolld task now has an unsupported PRE (*thereis(b.o, objectcode, created)*). This in turn triggers some planning activities to re-introduce to the plan the previously removed Toolcc task (compiling *b.c*) to re-establish this PRE. But the re-introduced Toolcc task has its own PRE unsupported, therefore some Sourcecoding tasks are also re-introduced to the plan. All these tasks will be re-executed, to modify *b.c* and relevant "*.*h*" files, and then to re-compile *b.c*. Here, the *iteration* is obtained through execution failure.

Another example concerns the changes in the Product Structure. In the same scenario, *b.h* was originally implemented by *b.c*. Suppose that *b.h* is now implemented by both *b.c* and *b1.c*, *i.e.* a change in the PS. According to Rule 6, I/O macros are re-expanded, causing a new unsupported PRE of the Toolld task (*i.e. thereis(b1.o, objectcode, created)*). This in turn triggers some planning activities to add a new Toolcc task and a new Sourcecoding task (to process the new *b1.c* file) to the existing plan. Here, replanning is *incrementally* carried out, not from scratch.

The software development strategy (a project- and/or company-specific policy) can be implemented by proper heuristics determining the priorities among the rules 1 – 7 (an extension of the heuristics determining the priorities of rules 1 – 2, described in section 3.1). The *controlling algorithm* given in section 3.1 largely remains unchanged. Here, the terminating condition for the algorithm – "the Agenda AG is empty" – means that all tasks in the plan have been successfully executed and removed.

Note that in this integrated model, PRE-conditions need not be split into static and dynamic parts (Rule 3 checks the entire PRE(A)). Also note that now the WSD is changing continuously.

5 Conclusion and Future Work

The first prototype of the EPOS Planner and EM uses the coupling method, implemented by 5000 lines of SWI-Prolog, working on the real EPOSDB implemented in C. SWI-Prolog has a symmetric interface with C, and an Object-Oriented extension PCE for graphical UI.

Together with the EM, the EPOS Planner provides the user with intelligent assistance on the *product level*. On the *project level*, it models the life cycle stages in software development, including task decomposition, human interaction, and develop-review iteration.

The preliminary experience with our first prototype proves that our EPOS Planner is capable of covering a large area of software development activities, and can be integrated with CM systems. It has successfully applied advanced AI planning techniques to an important Software Engineering area – Software Process Management.

Usually a PM system is based on some particular paradigm: process programming as in ARCADIA [TBC*88], static reasoning as in ALF [B*89] and partly MARVEL [KF87], dynamic triggering as in PCMS [HM88] and NOMADE [BE87], or subtype refinement as in Process-Oriented CM [BL89]. These PM systems use none or very little AI techniques (such as simple-minded forward/backward chaining). In contrast, EPOS PM combines all the above approaches, applies more advanced AI techniques, and also covers larger area of software development processes.

We are now implementing the integration of planning and execution (section 4) to support automatic iteration and incremental replanning. Another important direction for future work is **Case-Based Planning (CBP)**. The central idea of CBP [Kol88] is *learning*: (re)planning from memory and parameterization of previously generated plans to avoid the problem of constantly rederiving the same plan. We have already realized a simple plan-reuse strategy on the instance level (section 3.3). In future, we want to be able to generalize frequently used task networks as new, *composite task types* and put them back into the KB.

Yet another crucial point for future work is *cooperating transactions*. A transaction T_1 is not only associated with a version-choice C_1 (section 1), but also with an *ambition* A_1 – an incomplete version-choice, with some options `unset`. A_1 represents a set of version-choices (including C_1). When the changes made in T_1 is committed, they will be visible in all these versions. Now suppose there is another transaction T_2 with *overlapping* ambition and/or Product Structure with those of T_1. Then the policies for *change propagation* between T_1 and T_2 must be defined and implemented by PM. The subsequent challenge to the EPOS Planner is how to deal with more dynamic WSDs and KBs. Here, we will step forward from a single workspace to cooperating workspaces, and from a single-actor system to a multi-actor system.

Acknowledgments

Many thanks are due to Prof. Reidar Conradi for encouragement, inspirations and valuable comments on earlier drafts of this paper. I also thank P. H. Westby and E. Osjord for many discussions.

References

[Amb87] José A. Ambros-Ingerson. *IPEM: Integrated Planning, Execution and Monitoring.* Technical Report, University of Essex, Colchester CO4 3SQ, U.K., 1987. M. Phil. Dissertation.

[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 *[MSW90]*, May 1989. 23 p.

[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 *Proc. of the 11th Int'l ACM-SIGSOFT/IEEE-CS Conference on Software Engineering, Pittsburgh, PA*, 1989. 14 p.

[C*89] Reidar Conradi et al. Design of the Kernel EPOS Software Engineering Environment. In *[MSW90]*, May 1989. 17 p.

[Cha87] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[COWL90] Reidar Conradi, Espen Osjord, Per H. Westby, and Chunnian Liu. *Software Process Management in EPOS: Design and Initial Implementation.* Technical Report 15/90, EPOS report 100, 12 p., DCST, NTH, Trondheim, Norway, April 1990. Accepted at 3rd Int'l Workshop on SW Engineering and its Applications, Toulouse, France, 3-7 Dec. 1990.

[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.

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

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

[Kol88] Janet Kolodner, editor. *Proc. of a Workshop on Case-Based Reasoning*, Defence Advanced Research Projects Agency, Information Science and Technology Office (DARPA/ISTO), 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 *[Tic89]*, pages 56–65, 1989.

[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.

[MSW90] N. Madhavji, W. Schaefer, and H. Weber, editors. *Proc. of the First International Conference on System Development Environments and Factories*, Pitman Publishing, London, March 1990. SDEF'89, 9-11 May 1989, Berlin.

[NS87] Howard K. Nichols and Dan Simpson, editors. *Proc. of 1st European Software Engineering Conference* (Strasbourg, Sept. 1987), *Springer Verlag* LNCS 289, *404 p.*, September 1987.

[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.

[Tic89] Walter F. Tichy, editor. *Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, USA, 25-27 Oct. 1989, 178 p.*, ACM SIGSOFT Software Engineering Notes, November 1989.

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