A Process-Oriented Approach to Configuration Management

Yves Bernard and Pierre Lavency
Philips Research Laboratory, Brussels, Belgium

Abstract

We present a framework integrating concepts from configuration management and process management. A language, interpretable by the environment, is proposed in order to specify the development process of versions and configurations. Two structuring mechanisms are provided: a class-subclass hierarchy and a task-subtask hierarchy. The resulting expressive power is illustrated on some examples. We describe then how the environment supports the processes specified in this language as well as how it supports the dynamic refinement of process specifications.

Keywords: development process modeling, configuration management, generic environments.

1 Introduction

It is now becoming widely accepted that one of the major difficulties of maintenance problems stems from the fact that some knowledge, available during development phases, is not available any more during maintenance phases [1]. Typically this includes knowledge about the development process itself or about the different steps, the order in which these steps have been applied, the decisions taken at each step, etc.

Process modeling has thus become an active research area and efforts have been made to explicitly represent and make use of knowledge about the process in the environment, ranging from explicit representation of the life cycle ([26], IPSE2.5 [17]) or top level steps to rules describing tools (ODIN [3]) and elementary activities ([19], [14], MARVEL [8]).

At the same time configuration management techniques have been developed to help maintaining large software systems. Software configuration management has been defined in [25] as the discipline of controlling the evolution of complex software systems. Although there is an obvious relationship between the evolution of the software and the software development process, configuration management (CM) and process management (PM) facilities are usually loosely coupled. PM oriented environments provide system modeling facilities (MAKE [6]) as low level process description [23] but most of the time rely on tools such as RCS [24] for version control while CM oriented environments such as DSEE [11] do not really support emerging PM concepts such as activities or contracts of ISTAR [22].

In this paper, we describe a system where CM and PM concepts are integrated. In section 2, we develop our process model approach. In section 3, we present our configuration management model stressing the similarities between CM and PM problems. In section 4, we present our integrating framework. The next sections are dedicated to examples. We conclude by a brief section positioning our approach with respect to related research.

2 Process Model

In the literature two main approaches to process modeling can be distinguished.

On the one hand, in the process programming approach [16], the development process is described as a program before the development is started. This program is then executed by the environment and the developer's role is to provide the information required during the process execution. From the environment point of view, the process used by the developer to obtain this information does not matter and is buried in documents.

On the other hand, an environment such as ISTAR [4] focuses on capturing how this information has been acquired. The definition of the process is seen as an integral part of the development itself and is one of the developer's task. To support this view, ISTAR keeps track of the contracts dynamically issued by the developers in the contract hierarchy. However, the acquired knowledge is closer to a process trace (or program trace) than to a process description (or program). For instance, the terms of the contract (characteristics of the objects to be produced) and the relationships between subcontracts (sequence, parallel, etc.), are not known by the environment. Consequently, the functionality provided by the environment is reduced to the management of the client/contractor protocol.

In this paper, the process is a priori described in terms of classes (similar to process programs). This description can then be refined dynamically during the development (similar to issuing contracts dynamically).

From an architectural point of view, supporting this process model means that we have to go from generic environments instanciated with a process description to environments able to deal with dynamically acquired process descriptions. This raises, among others, the problem of maintaining the consistency of the knowledge about the process.

From the user's (developer's) point of view, this model requires that he formalizes his development process. This should be contrasted with the user's role in environments tracing the user's actions. This extra burden is put on the developers in order to have higher level descriptions that can be re-used later on during maintenance phases.

3 Configuration Management Model

In this section we review the different elements of our CM system emphasizing the similarities between CM and PM problems.

To control the evolution of the different components or modules, CM systems rely on version control systems (RCS [24], SCCS [21]). A version control system keeps track of the modifications by keeping different versions of a component. Versions are immutable and changes are made by introducing new versions (obtained by modification of a previous version copy). The set of versions associated with a component or the version family is usually structured into a history tree, the sons of a version A being the versions derived from A.

The functionality of version control systems described above is quite similar to the functionality of PM systems tracing user's actions. The system knows what modifications have been done but does not know their semantics or what assumptions have been made, what design decisions have been taken, why they have been taken, etc. Furthermore, there is usually no way to a priori specify sequences of steps that must be applied in order to develop a component although these development procedures ("life cycles") exist in all organizations.

As pointed out in [25], very little attention has been paid to facilities allowing a top down approach or change request driven CM. MRCS [10] is an early attempt to support the notion of change request. The difficulty to impose such a system to the developers reported in [10] must be related to the discussion, in the previous section, about the user's role. The DSEE task concept ("high level" tasks split into subtasks) can be seen as another attempt.

Large software products are not monolithic but rather they are composed of often many other components. Since these components exist now in different versions, we have to deal with a new kind of objects called generic configurations (system model in [11]). A generic configuration is a structured object containing references to version families. The configuration structure represents the relationships existing between the different components (*is-used-by*, *is-included-in*, etc.). A product is then an instanciated configuration (bound configuration thread in [11]) or a structured object obtained by substituting, in the generic configuration, each component name with the name of a specific component version. In our framework, an instanciated configuration is specified by a query denoting intensionaly its components versions. For this purpose, we use a relational query language extended with preferences described in [12,15].

From a PM point of view the notion of generic configuration is important to support top down software developments. As a matter of fact, a generic configuration is an abstract description of a software, independent of the low level design decisions that can be taken while developing its components. Furthermore, product building rules (or elementary process programs [23]) describing how to construct derived objects (i.e. object modules, executables, etc.) a la Make, can now be factored out of the specific instanciated configuration and associated with the generic configuration [11]. A previous paper [2] describes how it is possible to further factor out the product building rules by typing the configuration structures and associating the rules with the configuration types.

In the next section, we present a framework where

CM and PM facilities are tightly coupled. The system described here (called the task manager or TM) is only a component of our experimental environment. As far as this component is concerned, there are basically three types of versions: the generic configurations (viewed as a list of version families), the instanciated configurations (viewed as a list of versions) and the atomic versions. One should however keep in mind that this is only the predefined Task Manager view. In the environment we are elaborating, complex objects are typed [13]. On top of such a platform, the TM can be customized to take into account the knowledge about the different types of atomic and other predefined kinds of versions.

4 Overview

This section surveys and motivates the main features of our system. Our goal is here to be able to specify the development process of version trees (i.e. version family objects).

First, version *attributes* can be declared with their range of values. The attribute declarations specify the general version characteristics such as date, author, etc. as well as the relevant design decisions. An attribute models then a decision, its possible values represent the different alternatives and its value on a given version indicates what decision has been taken during the development of this version. Note that although the content of a version is immutable, its attribute values can be changed.

The development process or how the different versions must be developed is specified as a set of *tasks.* A task is described in terms of a *precondition* and a *postcondition* on the version attribute values. The precondition specifies on which kind of versions the task can be performed and the postcondition what kind of versions the task can produce (i.e. creation of a new version or change of the attributes of an existing version). They make explicit in the environment when a task can be performed and what are its effects (possible decisions that can be taken, etc.).

Constraints allow us to factor out of the task descriptions some knowledge about the development process or to specify knowledge that is independent of the decomposition in tasks. Constraints are relationships between attributes or conditions on attributes that must be true at any time or when some specific events occur (i.e. the creation of a new version and the change of attributes of an existing version) independently of the task triggering the event. These constraints are expressed as "directed" closed well-formed formulas written in a Prolog style, with references to the attributes of a version and of the versions derived from this version, before and after the events.

Preconditions and postconditions can be seen as task declarations. The task *body* further specifies how the task must be performed. This is specified with the same elements as the overall process (i.e. attributes, constraints and subtasks). These elements specify how intermediate versions and goal versions (i.e. satisfying the task postcondition) are developed from the initial versions (i.e. satisfying the task precondition). Since the attributes and constraints model knowledge independent of task decomposition, they are inherited through the task-subtask hierarchy. However, in order to allow some *local* inconsistency with respect to this knowledge, the inheritance can be blocked at a given level by redefinitions in the corresponding task body.

The notion of *class* is introduced in this framework as an encapsulation mechanism for attributes, constraints and task declarations describing a given development process. The same process "program" (class) is now shared by all the version family objects of a same class. Furthermore, a class can be re-used in different task bodies.

Finally, as in object-oriented languages, the classes are structured in a class-subclass hierarchy with inheritance of all the definitions from the class to the subclass. It allows us to incrementally describe the process by successive refinements and specializations. This is also a key feature as far as the dynamic knowledge acquisition is concerned. As a matter of fact, this acquisition can now be seen as a combination of two steps. In the first step, the knowledge, acquired during the development of an object, is modeled in a new subclass refining the class of this object. In the second step, the object is updated in order to become an instance of the new subclass.

5 A Priori Knowledge Specification

In this section, we develop a simple example illustrating how the a priori knowledge about the development process of components or version families is modeled in terms of class specifications.

The root class is called *basic-class.* It has three predefined attributes: *Id* the version identifier, *G-configuration* and *I-configuration*, indicating whether or not a version is a generic (resp. instanciated) configuration version. These attributes are automatically maintained by the system.

We first model two general version properties, the creation date and the author by defining a new class *dated-class* as a sublass of *basic-class*:

```
dated-class SUBCLASS-OF basic-class
    ATTRIBUTES
(A1) Date: String.
(A2) Author: Name.
    CONSTRAINTS
(C1) get-date(Date:initial-value).
(C2) get-user(Author:initial-value).
(C3) Date:old-value = Date:new-value.
(C4) Author:old-value = Author:new-value.
```

The attributes *Date* (A1) and *Author* (A2) model the corresponding general properties. The argument of the predicate *get-date* (resp. *get-user*) unifies to the current date (resp. user). The constraints (C1) and (C2) must be satisfied when a new version is created. They specify that no matter what tasks produce a new version, its *Date* (resp. *Author*) will be initialized to the current date (resp. user). The constraints (C3) and (C4) must be satisfied when attributes of an existing version are changed. They specify that those attributes may not be changed, no matter what tasks change the version attributes.

We now want to model the following scenario.

> The "successful" development of a version from another is decomposed into two basic steps, a version development step and a review step. A version is "successfully" developed when it has been positively reviewed. A negative review means that further developments cannot be based on the reviewed version or on any versions developed from this version[1]. Furthermore, the development of successive intermediate versions which must not be reviewed is allowed.

To support the above scenario, a new class is introduced:

```
step-class SUBCLASS-OF dated-class
    ATTRIBUTES
(A3) Status:[exp,rejected,to-review,released].
    CONSTRAINTS
(C5) Status=rejected⇒next-version:Status=rejected.
(C6) Id=scratch⇒Status=exp.
```

[1]The decision of a negative review on a version can be taken well after other versions have been derived from it. For simplicity sake, we have not introduced in the scenario, the notion of "soft" negative review allowing further development or enhancement from the reviewed version.

It is specified as a subclass of *dated-class* and it thus inherits the attributes and constraints defined at this level. The attribute *Status* (A3) is introduced to model the decisions involved in the scenario (*released*: a positive review, *rejected*: a negative review, *to-review*: not yet reviewed but must be reviewed, *exp*: not yet reviewed and must not be reviewed). The constraint (C5) specifies that all the versions derived from a rejected version are themselves rejected. This constraint ensures that the rejection decision on a version will be propagated on the versions derived from this version, no matter who takes this decision. Currently only one propagation "direction" is supported: from a version to its derived version. The constraint (C6) specifies the status of an empty version created when the module is initialized. These constraints must always be satisfied.

The *step-class* must now be completed with task specifications as follows:

```
(T1)  TASK step
(Pe1)  ON version.
(Po1)  TO version having Status=released.
(B1)   BODY {
(T2)     TASK develop
(Pe2)      ON version having  Status≠rejected
             preferring version having max Date.
(Po2)      TO version having  Status≠released.
(T3)     TASK review
(Pe3)      ON version having Status=to-review.
(Po3)      TO version having  Status=rejected
                              or Status=released.
(M3)       MODE attribute-update.  }
```

The global "successful" development is called *step* and its overall specification in terms of precondition (Pe1) and postcondition (Po1) is straightforward. The *step* task is itself composed of the two subtasks *develop* and *review* defined similarly by a precondition and a postcondition. These subtasks are basic (no associated body). Note how the preference is used in (Pe2) to model the notion of successive development. The mode (M3) is used to further describe the effect of a task. It specifies if the task updates the attributes of an existing version, if it builds a new version or if both are allowed (the default case).

The definition of a complex task such as *step* divides a version tree of class *step-class* into different subtrees. The *external tree* contains the versions produced by the task and on which the task is applied. An *internal* tree is associated to each version on which the task has been applied. It contains the versions on which the subtasks, specified in the body (B1), are applied and the versions produced by these subtasks. With respect to the *internal trees*, the definitions of a task body play a role similar to a class. It is possible to specify here all the elements of a class: attributes, constraints as well as tasks. The attributes and constraints defined above in the task-subtask hierarchy are inherited unless they are redefined internally. Here all the attributes (A1-3) and their constraints (C1-6) are inherited in (B1).

The Task Manager interprets the class definitions and shows what tasks can be started, ensures that only these tasks are started and detects conflicts between tasks. In the current system, there is no attempt to automatically start the tasks. The users must thus explicitly start the tasks as well as complete them. A started task is interpreted by the TM as an indication that the task *must* be performed to solve the problem at hand.

Starting at the top level external tree, the TM determines the "state" of each task. To do this, it evaluates the task precondition on the external task tree as follows:

1. If there is a unique version satisfying the precondition, the task state is *executable* and can be started on this version.
2. If there is no version satisfying the precondition, the task is *not-executable* and cannot be started.
3. If there are more than one version satisfying the precondition, the task is *to-refine*. The task can be started provided that first a unique version among the versions satisfying the precondition is selected.

Moreover, the subtasks of a task cannot be started as long as the task itself is not started. When a complex task (i.e. with a body) is started on a version, the TM determines the subtask "states" by evaluating their preconditions on the corresponding internal task tree. Initially, this tree consists of the version on which the complex task is applied. It is then enriched by the versions produced by the subtasks. It will be reestablished whenever the same task (i.e. with the same name) is reexecuted on the same version. When a basic task is started, the system merely creates a working copy of the version on which the task is applied (with only read permission in *update-attribute* mode). The working copy can then be viewed or modified by tools.

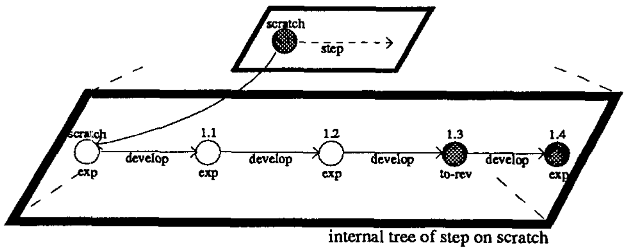

Figure 1: An object of *step-class*

For example, Figure 1 represents a snapshot of an object of *step-class* taken at a time where *step* is started from the scratch version and after some executions of *develop*. The two levels correspond to the two level task-subtask decomposition. Versions on which a task is *executable* are represented in gray. At this point, *develop* and *review* are *executable*: the former can be started on 1.4 while the latter can be started on 1.3.

The decision to complete a task is also user initiated. In the task external tree, this can result either in the change of the attributes of an existing version or in the production of a new version. The "production" means the creation of a new version when the task is basic (i.e. "check in") while it means the "export" of an internal version into the external tree when the task is not basic.

Besides ensuring that the postcondition is satisfied, the role of the TM is here to check if there are no conflicts between tasks. A conflict is detected if :

1. the version on which a task is started does not satisfy the task precondition any more;
2. a task is completed when some of its subtasks are started.

These are conflict situations since the decision to start a task on a version is interpreted as user indication to the TM that this task *must* be executed.

Let's now assume that *develop* is started on 1.4 from the situation described in Figure 1. This indicates that further development from this version must be done and is illustrated in Figure 2 where the following events (t0-5) are also depicted. The *review* task is started on 1.3 and end up with a negative result (t0). In order to take (C5) into account, the system must set the *Status* of 1.4 to *rejected* . It detects a first type of conflict since 1.4 does not satisfy (Pe2) any more. In this case the developer and the reviewer involved have to negotiate who should take the precedence. The next events in Figure 2 then describes a situation resulting from a precedence of the reviewer: 1.4 is set to rejected (t1) and the system then backtracks *develop* on 1.2 (t2). This is followed by a developer's decision to further backtrack by a rejection of 1.2 (t3), then the development of a new intermediate version (1.1.1.1) followed by the development and positive review of version 1.1.1.2 (t4). Then the *step* task is completed on 1.1.1.2 (t5). If at that time *develop* has been started on 1.1.1.2, a second type of conflict would be detected.

We now refine our scenario as follows:

> We distinguish four kinds of versions: the specification, the design, the prototype and the implementation versions. They can be developed in two different ways. The first way is to split the development in three successive phases that must ("successfully") develop respectively a specification, a prototype and an implementation version from the version produced by the previous phase. The second way is to split the development in two phases. In the first phase, the problem is decomposed and a generic configuration version or a design version is ("successfully") produced. In the second phase, the specification (resp. prototype and implementation) versions of the components identified in the generic configuration, are integrated to produce the final specification (resp. prototype and implementation) versions.

We first model the three phase development in a new class called *life-cycle-class.*

```
life-cycle-class SUBCLASS-OF step-class
         ATTRIBUTES
(A4)       Language :  [spec,design,impl,prot].
         CONSTRAINTS
(C7)       Language:old-value=Language:new-value.
(C8)       Id=scratch ⇒ Language=spec.
(C9)       Language=design ⇔ G-configuration=true.
(T4) TASK spec REFINE step
(Pe4)      ON version having Language=spec.
(Po4)      TO version having Language=spec.
           BODY {
             CONSTRAINTS
(C10)          Language:initial-value=spec.  }
(T5) TASK prot REFINE step
(Pe5)      ON version having Language=spec.
(Po5)      TO version having Language=prot.
           BODY {
             CONSTRAINTS
(C11)          Language:initial-value=prot.  }
(T6) TASK impl REFINE step
(Pe6)      ON version having Language=prot.
(Po6)      TO version having Language=impl.
           BODY {
             CONSTRAINTS
(C12)          Language:initial-value=impl.  }
```

The attribute *Language* is introduced to model the different kinds of versions. The constraint (C7) specifies that it may not be changed and (C8) the attribute value on the initial version. The constraint (C9) specifies that *design* versions are the only generic configuration versions.

The three phases are specified by three different refinements (T4-6) of the inherited *step* task, the new elements specified at this level being taken into account in conjunctions with the inherited definitions. Note how the constraints (C10-12) specified in task bodies force the intermediate versions to be of a certain kind independently of the decomposition of the task into subtask (here *develop* and *review*). Since (C9) is inherited through the task-subtask hierarchy, these intermediate versions may not be generic configuration versions.

To model the two phase development (i.e. a decomposition followed by an integration phase), we must add the following definitions to *life-cycle-class*:

```
        CONSTRAINTS
(C13)   G-configuration=true⇒
              component:Class=life-cycle-class.
(C14)   I-configuration=true⇒
              Language=component-version:Language.
(C15)   the versions of all components must have
              same Language.
(T7)  TASK decompose REFINE step
(Pe7)   ON version having   Language=spec
                          or Language=design.
(Po7)   TO version having   Language=design.
        BODY {
          CONSTRAINTS
(C16)       Language:initial-value=design.  }
(T8)  TASK integrate
(Pe8)   ON version having   Language=design.
(Pe8')  I-ON version of all components having
          Language≠design and Status=released.
(Po8)   TO version having  Status=released
                          or Status=rejected.
```

The constraint (C13) specifies that the class of the components of generic configuration versions must be *life-cycle-class* (or one of its subclass). The constraint (C14) specifies that the *Language* attribute of an instanciated configuration version is the *Language* of its constituting versions. The constraint (C15) is what we call a compatibility constraints [15]. It prevents the mixing of different kinds of versions in all the tasks building instanciated configurations.

The specification of the decomposition phase (T7) is similar to (T4-6) except that the produced and intermediate versions must be *design* versions (C16) and thus from (C9) generic configuration versions.

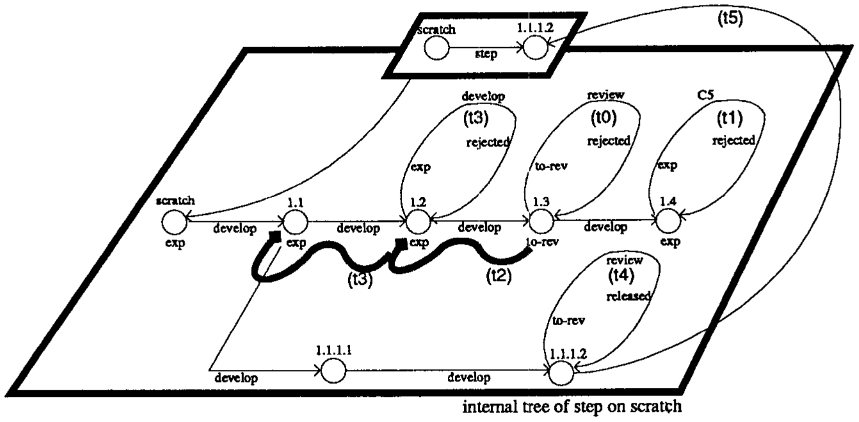

Figure 2: An object of *step-class*

The integration phase (T8) is specified as a task building instanciated configurations. These tasks are characterized by preconditions such as (Pe8,Pe8'). The condition (Pe8') is evaluated on the version sets of the different components of a generic configuration (satisfying Pe8), the instanciated configurations already built from this generic configuration or violating the compatibility constraints being filtered out of the result. As long as the combined evaluations of (Pe8) and (Pe8') yields an empty answer, the task (T8) cannot be started. When this task is started a working instanciated configuration is created. It will become a built instanciated configuration version when the developer decides to complete the task.

Our scenario is incomplete. For simplicity sake, we have not talked about the synchronization between the different integration phases (i.e. the integration of the specification versions must be performed before the integration of the prototype versions, etc.). The overall goal of the development, which is of course to produce an implementation version, can however easily be modeled . We merely have to specify this goal as a postcondition of a task with a body specified by *life-cycle-class*. We have

```
project-class SUBCLASS-OF basic-class
TASK project
  ON version.
  TO version having    Status=released
                  and Language=impl.
  BODY { life-cycle-class. }
```

6 Dynamic Knowledge Acquisition

In this section, we develop an example illustrating how the system supports dynamic knowledge acquisition. This is done in two steps. In the first one, the knowledge, acquired during the development of some objects, is modeled in a new subclass refining the class of these objects. In the second step, these objects are updated in order to become instances of the new subclass. The current system supports only update to a target class subclass of the initial object class. Other conditions must be satisfied and are checked by the system. We will discuss them later on.

For example, let's consider a project to develop a query interpreter. An object named QEV of class *project-class* is thus created and the task *project* is started on the scratch version of QEV. Let's now assume that during the internal development process of the task *project*, it is decided to decompose the problem (i.e. start the task *decompose*). Let's further assume that this task produces a generic configuration version with three components of class *life-cycle-class*: SYN a syntax checker, SEM a static semantic checker and EVL a query evaluator. The development of versions for these components can now proceed concurrently.

We now consider the development of versions of EVL. Let's assume that it is decided not to further decompose the problem (start *spec* rather than *decompose*). We then have the following scenario.

> After the development of a specification (i.e. the subtask *develop* of *spec* denoted by *spec/develop* is completed) and before the specification review, its developer wants to document an important decision that he has taken about the form of the answer that must be returned by the evaluator.

This is first modeled by introducing the *Answer* attribute in a new subclass:

```
EVLdoc1 SUBCLASS-OF life-cycle-class
     ATTRIBUTES
(A5) Answer:[undecided,intensional,extensional,both].
(T9) REFINE spec
(C17)  BODY { CONSTRAINTS
                         Date≤date1⇒Answer=undecided.
      (Date>date1,Date≤date2)⇒Answer=intensional.  }
```

The attribute (A5) models the *Answer* form alternatives while the constraint (C17) specifies when and what decision has been taken. Note that a task (T9) defined as the refinement of an inherited task has the same name (here *spec*) unless a new name is explicitly specified (as in T4). The object EVL must now be updated to become an instance of this new class. When a new attribute is defined in the target class, the user performing the update is prompted for the attribute values of each existing versions. Only the values satisfying the constraints such as (C17) linking the new attribute with previous attributes, are proposed. In this example, the update will thus be performed automatically if there is no version developed after *date2*.

The next step in our scenario is the following one.

> The specification has now been successfully reviewed. The reviewer wants to motivate this decision and relate it to the development alternatives. Since he thinks that this motivation is relevant for the other reviewers, he wants to share this knowledge with them.

To model this, the reviewer first defines the following class.

```
EVLdoc2 SUBCLASS-OF EVLdoc1
     REFINE spec,impl,prot,decompose
       BODY {
(C18)    CONSTRAINTS Status=released⇒
             (Answer=intensional;Answer=both).  }
```

The constraint (C18) specifies the decisions that must have been taken about the *Answer* form during the development of a version in order to allow its positive review. Since (C18) specialilizes the body of the four tasks *spec, decompose, prot* and *impl*, it affects the review subtask in each of them. For instance, *prot/review* will not be allowed to release a prototype version if the value of *Answer* is not *intensional* or *both* on this version. The object EVL must now be updated to become an instance of this new class. This update will be aborted by the system if there are versions that do not satisfy the new constraint (C18).

We now consider an example of task plan with the following scenario.

> After the completion of the specification phase, the prototype phase is started. The developer of this phase wants to plan its work and to decompose it in two phases: (1) the "successful" development of a prototype version returning an *extensional* answer and (2) from this version, the "successful" development of a prototype returning an *intensional* answer.

The developer first defines the following class which refines the *step* task into *step-int* and *step-ext.*

```
EVLplan1 SUBCLASS-OF step-class
(A6)   ATTRIBUTES
         Answer:[intensional,extensional].
(T10) TASK step-ext REFINE step
(Po10)  TO version having Answer=extensional.
        BODY {
(C19)     CONSTRAINTS Answer=extensional.  }
(T11) TASK step-int REFINE step
(Pe11)  ON version having Answer=extensional.
(Po11)  TO version having Answer=intensional.
        BODY {
(C20)     CONSTRAINTS Answer=intensional.  }
```

This new subclass is then re-used for defining the body of the *develop/prototype* task as follows:

```
EVLdoc3 SUBCLASS-OF EVLdoc2
(T12) REFINE prot/develop
(Pe12)  TO version having local:Answer=intentional
                          and Answer=intensional.
(Po12)  BODY { EVLplan1.  }
```

This definition transforms the basic task, *develop* of *prot* into a complex task. Except *Language*, all the attributes are redefined locally in the class specifying the task body, their associated constraints are thus not inherited. Note however that some of these constraints (C1,C2, etc.) are inherited by the defining class and thus introduced anyway. Practically, this means here that (C18) is not inherited allowing internal positive review of prototype returning an *extensional* answer (i.e. a local inconsistency with respect to C18) but (C7,C9,C11) are inherited constraining the intermediate version to be *prototype* versions. Note also how the key word *local:* is used in (Pe12) to distinguish the internal attribute (A6) from the external one (A5). Once again the object EVL must now be updated. This update will be aborted by the system if versions have already been produced by *prot/develop.*

Since the plan, knowledge acquired during the prototyping phase, has been encapsulated into a class, it can be easily re-used by the developer of the implementation phase as follows:

```
EVLdoc4 SUBCLASS-OF EVLdoc3
REFINE impl/develop
  TO version having local:Answer=intensional
                    and Answer=intensional.
  BODY { EVLplan1. }
```

We now assume that all the phases have been completed and we discuss a maintenance problem. We assume that the top level external trees of EVL contains a specification (1.1), a prototype (1.2), an implementation and a scratch version (see thin arrows and nodes in Figure 3).

We consider the following maintenance scenario.

> A check that has not been implemented is detected. It has not been implemented because it has not been specified.

The identification of the new alternative is modeled by

```
EVLdoc5 SUBCLASS-OF EVLdoc4
ATTRIBUTES
  Check1:[undecided,performed,not-performed].
```

The object EVL must now be updated to an instance of this new class.

The "state" of the *spec* task is *to-refine* (i.e. precondition satisfied by scratch and the specification version 1.1). In order to develop a new specification including the decision to perform the check (i.e. *Check1 = performed*), one must first decide if it should be developed (1) from scratch or (2) from the previously developed specification (1.1). If (1) or backtracking to the scratch version is chosen, the internal tree of *spec* on scratch is reestablished. This solution should be chosen if some decisions taken during the development of 1.1 must be changed. If on the other hand the introduction of the check has no (or few) effects on these decisions, the solution (2) should be chosen. The completion of these two possible choices are illustrated in Figure 3. The situation existing before the choice is depicted in thin lines and white circle versions, while the two possible resulting situations are in thick lines and gray circle versions. The relationship between the development tree of EVL and the overall project is also represented.

It is possible to specify that the same type of choice must be made if a similar situation occurs again by refining the task definitions. For instance, the second solution will be modeled as

```
EVLdoc6 SUBCLASS-OF EVLdoc5
REFINE spec
  ON version having Check1=undecided.
  TO version having Check1=undecided.
TASK spec' REFINE spec
  ON version having Check1=undecided
                    and Id≠scratch.
  TO version having Check1=performed.
```

This will allow the system to ensure that whenever a new specification including the decision to perform the check will be needed, it will be developed from a specification where the decision to perform the check is not taken. Note that the inheritance mechanism makes sure that the knowledge previously acquired (among others C18) during the development will be re-used during this maintenance phase (i.e. execution of *spec'*). The update of EVL to an instance of *EVLdoc6* will be aborted by the system if there are versions produced by *spec* with $Check1 \neq$ *undecided* or if those versions have been produced from versions having $Check1 \neq$ *undecided*.

We now assume that all the phases have been completed on all the components (i.e. EVL, SYN and SEM) and we discuss an integration problem. We have the following scenario.

> During the integration phase of prototype versions, a check (called *Check2* in the sequel) that has not been implemented is identified. This check can be performed by either SYN or SEM.

It is first modeled on SYN and SEM in a similar way *Check1* is modeled on EVL in a previous example (class *EVLdoc5*) followed by an update of SYN and SEM to instances of the new class. At the integration level it is modeled by the definition of a subclass where a new compatibility constraint is introduced as follows:

```
QEVdoc1 SUBCLASS-OF project-class
CONSTRAINTS
  the version of SYN must have Check2=performed or
  the version of SEM must have Check2=performed.
```

This constraint prevents the building of any instances having the same problem and thus ensures that no one will have to go through the same bug analysis again (for example during the integration of implementation versions).

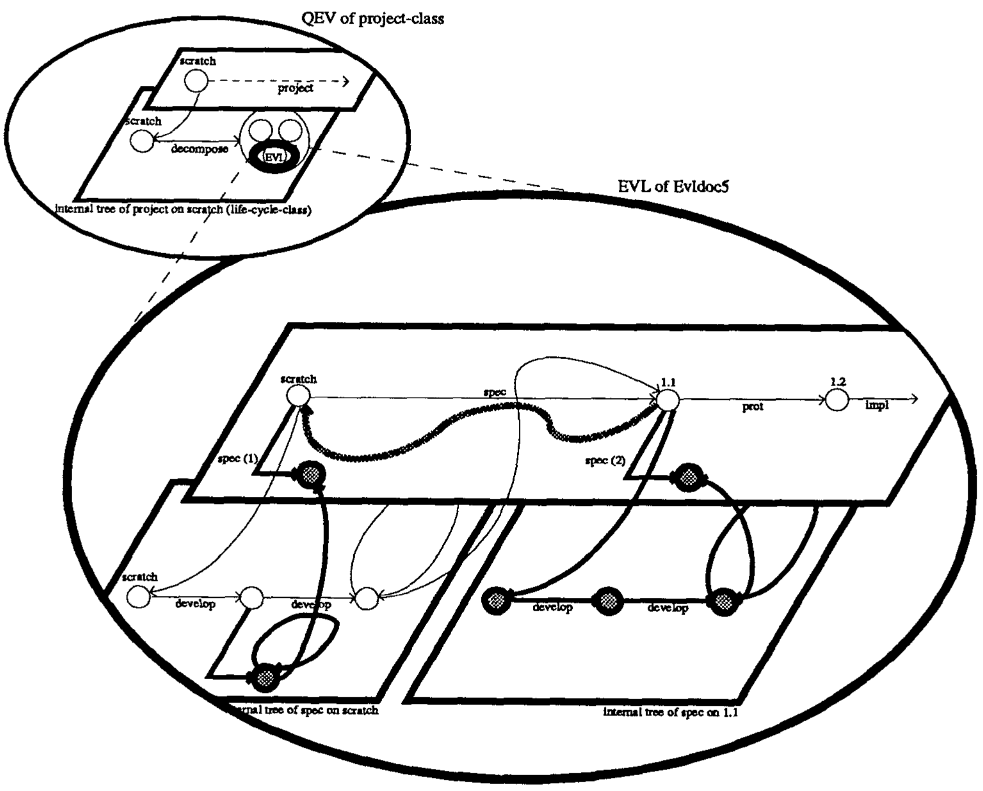

Figure 3: EVL and QVL

7 Conclusion

In this section, we compare our work with other research and summarize our main contributions.

Our decision model is similar to the issued based model of [20]. Our approach can also be seen as a complementary and intermediate approach as compared with language specific approaches such as in process management, the transformational approach [1] and in configuration management, the approach proposed by [27].

Tasks can be seen as contracts [4] whose terms have been formalized in a system interpretable manner. Task formalization in preconditions and postconditions has been influenced by work done in process management [26] and more specifically in tool integration such as [14], ODIN [3], MARVEL [8]. We are now currently studying the integration of task descriptions with tool descriptions as proposed in [14]. This will allow us to specify what tools can be used within a task. We are also studying the feasibility of inferring task plans from task descriptions with techniques similar to those used in MARVEL.

As far as we know, there is no equivalent approach proposed in configuration management. This is the first contribution of this paper. Note however that the conflict detection mechanism can be seen as an application, in version control, of predicate locking as proposed in [5] and should be contrasted with the explicit object locking of RCS [24]. The above mentioned inference mechanism should allow us to support conflict prevention rather than detection.

Our language allows us to specify process programs or life cycle as in the process programming approach [16]. However, as MELD [9], our language does not have any explicit control structures. This property has allowed us to integrate two structuring mechanisms, the class-subclass and the task-subtask hierarchies. It is this combination that gives to our approach the power necessary to support not only a priori life cycles but also dynamic refinements of this knowledge (e.g. dynamic task splitting as subcontracts issued in ISTAR [22]) in an integrated manner. This is the other main contribution of this paper.

The support of the dynamic knowledge acquisition makes that our system can also at least partially support "faking the process" [18] or less constructive approaches. With respect to these approaches, the consequences of the system limited support must still be assessed. Similarly the problems of (logical) consistency inherent to any dynamic acquisition have only been partially solved [7].

We are currently studying features allowing one to make explicit in the environments other aspects of the development process such as resource allocation and other project management issues. Finally, since an environment will not be fully practical without a powerful user interface, we are now developing graphical browsers and other visualization tools[2].

Acknowledgement: We would like to thank M. Lacroix, M. Vanhoedenaghe, R. Conradi and J. Mueller for their useful comments and suggestions.

References

[1] R. Balzer, "A 15 Year Perspective on Automatic Programming", IEEE Transaction on Software Engineering, SE-11(11), November 1985.

[2] Y. Bernard, M. Lacroix, P. Lavency, M. Vanhoedenaghe, "Configuration Management in an Open Environment", 1st European Software Engineering Conference, September 1987, Strasbourg, France.

[3] G.M. Clemm, "The ODIN System: An Object Manager for Extensible Software Environments", Phd Thesis, University of Colorado-Boulder, Department of Computer Science, 1986.

[4] M. Dowson, "An Integrated Project Support Environment", 2nd ACM/SIGSOFT/SIGPLAN software Engineering Symposium on Practical Development Support Environment, ACM SIGPLAN Notices, vol. 2, no. 1, January 1987.

[5] K.P. Eswaran, J.N. Gray, R.A. Lories, I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Data Base System", CACM 19, 11, November 1976.

[6] S. I. Feldman, "Make — A Program for Maintaining Computer Programs", Software — Practice and Experience 9, 4, April 1979.

[7] E. Gribomont, M. Lacroix, P. Lavency, "Consistency of Compatibility Constraints in Configuration Management", COMPEURO 88, Brussels, Belgium, April 1988.

[8] G.E Kaiser, P.H. Feiler, "An Architecture for an Intelligent Assistance in Software Development", 9th International Conference on Software Engineering, Monterey, California, USA, March 1987, 180-188.

[9] G. Kaiser, D. Garlan,"Melding Software System from Reusable building blocks", IEEE Software, July 1987, 17-24.

[10] D.B. Knudsen, A. Barofsky and L.R. Satz. "A modification request control system". Proceeding of the 2nd International Conference on Software Engineering, 1977.

[11] D.B Leblanc, and R. Chase, "Computer-aided software engineering in distributed environment ", ACM SIGPLAN notices, 19(5):104-112, May 1984. Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

[12] M. Lacroix, P. Lavency, "Preferences: Putting More Knowledge into Queries", 13th International Conference on Very Large Data Bases, Brighton, England, September 1987, 217-225.

[13] M. Lacroix and M. Vanhoedenaghe, "Manipulating Complex Objects" in Proceedings of the Workshop on Database Programming Languages, Roscoff, France, September 1987, F. Bancilhon and P. Buneman, editors, ACM Addison Wesley, to appear.

[14] M. Lacroix, M. Vanhoedenaghe, "Tool integration in an Open Environment", submitted for publication.

[15] P. Lavency, M. Vanhoedenaghe. "Knowledge Based Configuration Management" 21st Hawaii International Conference on System Sciences Kona-Kailua, Hawai , USA, January 1988.

[16] L. Osterweil, "Software Processes Are Software too", 9th International Conference on Software Engineering, Monterey, California, USA, March 1987, 2-13.

[17] M.A. Ould, C. Robert, "Modeling Iteration in the Software Process" , Iteration in the Software Process, Proceeding of the 3rd International Software Process Workshop. Breckenridge, Colorado, USA November 1986 , Mark Dawson, Editor.

[18] D. Parnas and P.C. Clements, "A rational Design Process: how and why to fake it", IEEE Transaction on Software Engineering, SE-12: 251-257,1987.

[19] W. Polak, "Framework for Knowledge-Based Programming Environments", Advanced Programming Environments, Proceeding of an International Workshop, Norway, June 1986. Lecture Notes in Computer Science, Springer-Verlag, edited by G. Goos and J Hartmanis.

[20] C. Potts and G. Bruns, "Recording the Reasons for Design Decisions" 10th International Conference on Software Engineering, Singapore, April 1988, 418-427.

[21] M.J. Rochkind, "The Source Code Control System", IEEE Transactions on Software Engineering, December 1975.

[22] V. Stenning, "An Introduction to ISTAR", P. Perigrinus Ltd.,London, U.K., 1986.

[23] R. Taylor at al., "Next Generation Software Environments: Principles, Problems, and Research directions. Information and Computer Science, University of California Irvine, Technical report, July 1987.

[24] W.F. Tichy, "Design, Implementation and evaluation of a Revision Control System", in Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, Japan, September 1982.

[25] W. Tichy. "Tools for Software Configuration Management", International Workshop on Software Versions and Configuration Control, Grassau, FRG 27/29 January 1988.

[26] L.G. Williams. "Software Process Modeling: A Behavioral Approach", 10th International Conference on Software Engineering, Singapore, April 1988, 174-186.

[27] J. Winkler, "Version Control in Families of Large Program", 9th International Conference on Software Engineering, Monterey, California, USA, March 1987, 150-161.

[2] All figures have been made with NeWSillustrator, a graphically extensible drawing editor developed by Y. Bernard.