Evolution of Software Processes and of their Models: A Multiple Strategy Approach

Mohamed Ahmed-Nacer
Computer Science Institute, USTHB University
Software Engineering Laboratory
BP 32 El-ALIA-Bab-Ezzouar, ALGERIA
E-mail: anacer@wissal.dz

This paper deals with a new approach to modify software process models and their software processes. Such an approach, in contrast with existing ones, supports the different evolution strategies for the software process models, and allows the adaptation of software process during execution. The integration of these two types of evolution is based on a chaining of phases (generation/evolution/simulation/execution). It allows the change of software process models as and when executed without inconsistencies, to perfect software processes through simulation and to define new ones dynamically.

Keywords: software process, process model, evolution, software engineering, simulation.
Classification: D.2 Software Engineering
K.6. 3 Software management
• Software process

1. INTRODUCTION

Process modelling is a way to describe how a process should be conducted according to a model (process model). A process is generated from the process model that has been attached with enough information to start execution.

Software development is a learning process that is highly dependent on the context and circumstances (Cugola and Ghezzi, 1999). The Software crisis has shown that improvements in production processes are necessary to improve product quality: the process used to develop and to modify the software plays a crucial role in the search for the software quality and the user's satisfaction. However, software processes are incomplete and not determinist. For example, 1) tasks to be executed may be dependent on a product structure that has been fixed at development time, and 2) changes in software development policy may necessitate the dynamic addition or deletions of activities at process time. This means that software processes need to be dynamically adjusted throughout the project.

Most of the works on software processes focus on the evolution of process models (Bandinelli, Fugetta and Ghezzi, 1993; Conradi, Nguyen, Inge and Liu, 1998; Kabba and Derniame, 1995; Kaiser and Ban-Shaul, 1993; Madhavji, 1992; Minh, Wang and Conradi, 1997). Very often, this evolution

Manuscript received: 22 January, 2002
Communicating Editor: Denis Warne

is carried out through ad-hoc procedures or pre-defined policies. The latter is inappropriate because it does not offer a way to adapt rules of evolution to suit a specific software development process. Moreover, as these approaches concentrate only on the process model level, there is no possibility (or limited supports) to operate directly on the software process during its execution.

I shall present, in this paper, a new approach that focuses on the evolution aspects of 1) the software process model level and 2) the software process level. At the first level, an innovative approach, in contrast with existing ones, supports different evolution strategies: process models must not be submitted to the same evolution constraints. At the second level, the possible changes allow adaptation of software processes during their execution.

These two complementary types of evolution are integrated. This integration, based on a chaining of phases (generation/evolution/simulation/execution), offers an evolution environment that takes into account the multiple cases of changes such as faults in the process, missing steps, dynamics of the environment (development policy, users, technologies). This environment allows the modification of software process models at execution time, to perfect software processes through simulation and to define new ones dynamically.

2. SOFTWARE PROCESS MODEL AND SOFTWARE PROCESS TRANSLATION

Many approaches for software process modelling have been proposed (Derniame, Kinkelstein, Kramer and Nuseibeh, 1994; Jaccheri, Conradi and Dyrnes, 2000; Nitto, Patricia, Schaeffer and Hala, 1999; Reimer and Schaeffer, 1997; Wang, Larsen, Conradi and Munch, 1998). These approaches differ on several points according to the process modelling formalism used. For instance, there is a great variety of software process modelling languages (Conradi and Jaccheri, 1999; Fugetta and Wolf, 1996). These formalisms are based on Petri net notations (Fernstrom, 1993; Bandinelli, Braga, Fugetta, and Lavazza, 1994), on rules (Junkerman, Peuschel, Schaefer, and Wolf, 1994), on procedural process programs (Sutton, Heimbigner and Osterweil, 1995; Sutton and Osterweil, 1997) and on events and triggers (Estublier and Dami, 1996). The evolution approach described in this paper uses the concepts proposed in the APEL formalism (Dami, Estublier and Amiour, 1998) for the graphic description of software process models. This graphical notation is very similar to OMT notation (Rumbaugh, 1995).

2.1 Basic Concepts of APEL

APEL is a graphical and executable formalism for process models. Process models are described by using static and dynamic aspects.

Static Aspects

Static aspects encompass all the entities involved in an APEL process definition. They are mainly *activities, products* and *agents*.

An activity is an atomic or composite operation: a process model is composed of a set of activities which can be recursively decomposed, by following different levels of abstraction beginning with the generic process model to the specific one. A product is an entity produced or manipulated in an activity (document, program module, etc). An agent represents a person in charge of one or several activities in which he will be assigned different roles.

Any entity (be it a product, an agent or an activity), which is defined in a process model, is typically defined. The concepts of inheritance (specialisation/generalisation) and composition/decomposition are used during process modelling. A product or an activity type, once defined, can be reused in different processes.

Dynamic Aspects

The dynamic aspects of the process are mainly described through *control flow, data flow* and the *state diagram*. These aspects are based on the concept of event and event capture. Events are automatically generated each time a) a method is called, or b) the state of an entity is changed, or c) a temporal signal (clock) is produced. Events are captured by the entities (activity, product, and agent) that are registered to react to that event.

- The control flow (CF) describes how activities will be launched and synchronised during the process execution. A control flow indicates the expected *event* at its origin and the command to be sent to its destination. Figure 1 shows a control flow between *Development* activity and *Test* activity. This Control flow means that when the event (*Development, end*) occurs (upon the termination of the *Development* activity) the *Start* method is applied on *Test* which is automatically started.
- The data flow (DF) shows how products circulate between two activities that consume, transform or produce them. Thus, a data flow connects the output of an activity to the input of another activity. This connection defines the access mode (*read, write, exclusive*) and the transfer mode of a product (*transfer, share, copy*, etc).
  Figure 1 shows examples of:
  – A data flow that *transfers* the document *spec_doc* from the *Requirement specification* activity to the *Development* activity.
  – Two data flows that indicate two sharing products (*obj_code* and *src_code*) between the *Development* activity and *Test* activity.
- The state diagrams (SD) are determined for the activity, product and agent types. For a given type, the SD describes (1) how an entity of that type evolves in time (i.e. which states it goes through) and (2) the events and conditions that cause the state changes. Activities have a predefined SD with the states: *inactive, active, suspended, terminated* and *aborted*. More details on the static and dynamic aspects of APEL are given in Dami *et al*, 1998.

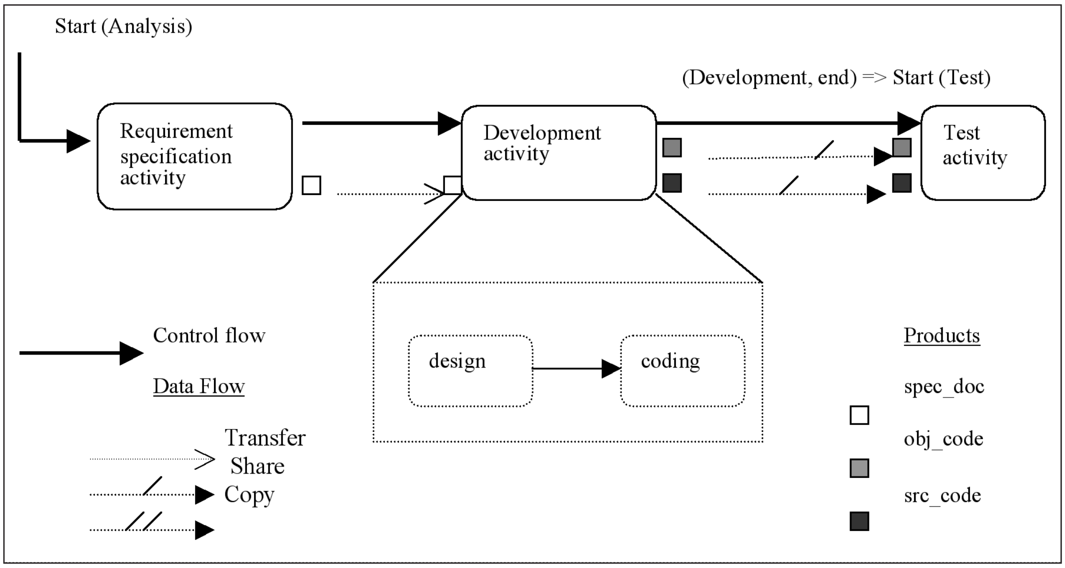


Figure 1: A Process Model

2.2 Translation : Software Process Models and Complex Objects (Aggregates)

A software process model, once defined graphically using APEL formalism, is translated automatically in terms of objects according to the Adele data model (Estublier and Casallas, 1994). This allows the management and modification of software process models and software processes. In the Adele environment, the basic concepts of the data model are:

Object classes and relation classes

Entities (activities, products and agents) are described by objects. The connections between these entities (control flow, data flow) are described by relations. The objects and relations are created from object and relation classes, respectively, that are generated by a compiler at the time of the translation. Object classes describe activity types, product types and agent types. Relation classes describe the different connections of control flow and data flow.

A predefined class called *Activity* describes the default behaviour of an activity as well as the common features that manage it. Thus, an activity is in charge of creating its own Workspace (WS). The workspace of an activity represents the sub-activities, the products, the agent assigned to this activity and the different connections of data flow and control flow. The concepts of inheritance (specialisation/generalisation) and composition/decomposition used for data modelling are also used during the process modelling.

Aggregates

This data model supports complex objects called aggregates. An aggregate is an object linked to its components through relationships. The semantics of an aggregate are defined by relationship behaviour, which is defined by users. Thus, almost any kind of aggregate with any behaviour and consistency constraints may be defined.

Triggers

The dynamic aspects are described in the definition of object classes and relation classes using trigger rule formalism.

Triggers take the following form "*RULE name_rule WHEN event* {*Action*}», where "*event*" is a predicate of the system state, database state and the activities underway (queries, navigation as well as changes). "*Action*" is a method (or a program).

A trigger defined in an object class is executed when the associated event occurs to an object of this class.

A Trigger defined in a relation class is executed whenever the associated event occurs to an object linked by this relation. Thus, we can define for each object a specific semantic depending on the nature of the relation that links this object with the others; that's how the aggregates are managed. So, an action on an "X" object has the effect of executing the triggers defined for the relations where "X" is the origin and also for the relations where "X" is the destination. These triggers are defined respectively by the keywords ORIGIN and DESTINATION.

Section 4 shows how the use of the aggregate notion combined with the use of the triggers on relations allows defining multiple strategies (semantics) to modify software process models.

3. EVOLUTION CONTEXT

Software process models can be adapted to suit different software development contexts. They define parameters and characteristics that determine the way to derive (or to generate) software processes. Moreover, mechanisms must be provided to allow the adaptation of software process models dynamically to the new needs, to correct inconsistencies found during the process execution, to modify some constraints or to act directly on the process execution. For instance, feedback may occur which would require the re-execution of activities, etc.

As the proposed approach integrates two evolution levels (the evolution of software process models and the evolution of the software processes), each change on a process model has an impact on the software processes and vice versa. The evolution approach is based on a chaining of phases (generation/evolution/simulation/execution) to reduce and to control change risks (Figure 2). This chaining of phases, similar to the spiral model (Boehm, 1988), allows the development of the software processes and their models without inconsistencies. For instance, when a problem is detected or a change is needed in a software process model, the system determines the possible changes and effects in the software process. The simulation phase is thus launched to evaluate and validate these changes. During this phase, the software process model is updated and validated through multiple simulations before its re-execution phase. These changes of phase are controlled through triggers (ECA rules). The communication of reports from one phase to another updates the software processes and their models in an incremental manner.

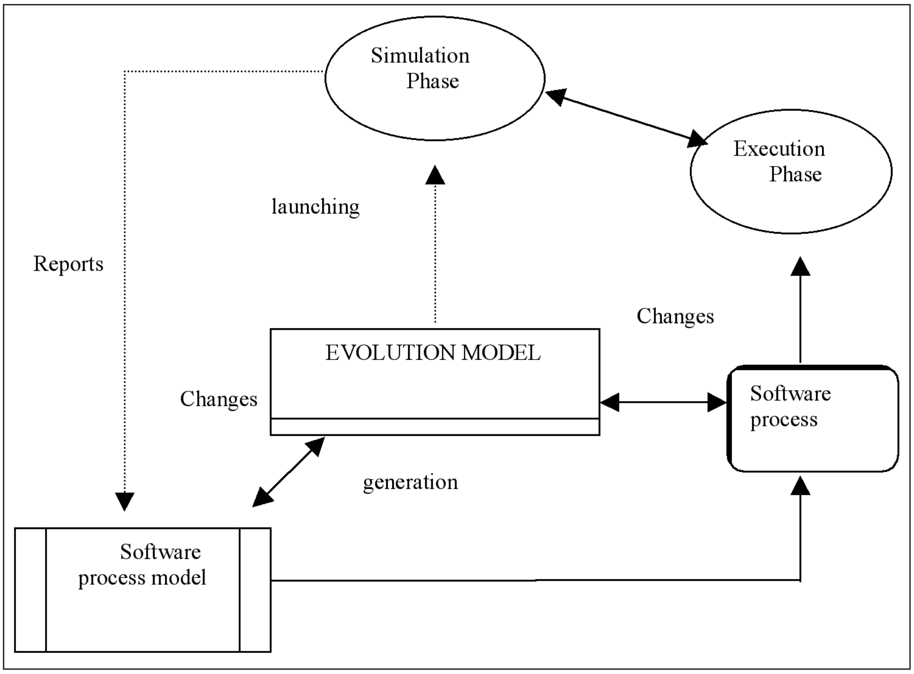


Figure 2: Chaining of Phases

A transaction mechanism is used to avoid multiple feedback whenever a change is needed in the simulation phase: Change operations can be grouped in a unique transaction. This offers great flexibility during the process model evolution. Moreover, because of the order of updating operations (that has the effect on the process model consistency), the transaction mechanism allows intermediate inconsistencies: this leaves the project manager free to define his updating operations in any order. The software process model is thus updated and validated as soon as the transaction is validated in the simulation phase.

4. SOFTWARE PROCESS MODEL EVOLUTION

Very often, process model evolution is achieved either through ad-hoc procedures or pre-defined policies. This approach is not flexible, as it offers no way of adapting evolution rules to suit individual applications. This section presents the evolution mechanism that allows different evolution strategies for the process models. Consequently, the process models should not have to submit to the same evolution constraints. The evolution semantics, during the evolution of the process models may differ according to the desired evolution strategy.

4.1 Taxonomy of Updating Operations

When a required change in the process model concerns entities such as software component types, activity types, tool types or agent types, the evolution mechanism that is activated is the one that we have developed for the schema evolution (Ahmed-Nacer and Estublier, 2000) (indeed activity types, tool types or agent types are defined as object classes in the process model – see Section 2.2).

A semantic is associated with every updating operation, by default. In addition the multiple semantics of evolution are available in a library enabling software project managers to choose their evolution strategy freely. This library is not static. The project manager may modify it at any time by adding new semantics of evolution. The evolution system remains open and managers are able to decide how their process models should evolve.

The taxonomy of updating operations proposed by default is the one discussed in (Ahmed-Nacer and Estublier, 2000) extended with specific updating operations that are inherent to the process model.

1. Inheritance relation modifications:
   - add a class (activity type, product type or agent type) in a process model,
   - delete a class (activity type, product type or agent type),
   - rename a class (activity type, product type or agent type),
   - add a sub-activity, delete a sub-activity, change the order of the sub-activity list.

2. Class definition modifications:
   - add, delete and rename a property (attributes, methods, events and triggers),
   - change the value domain of a property (attributes, method signatures),
   - add, change, rename a default value of an attribute,
   - add, delete, change a constant attribute,
   - change the characteristics of an attribute or a relationship.

4.2 Multiple Evolution Strategies: The Mechanism

As mentioned in Section 2.2, an aggregate is an object linked to its components by relationships. The aggregate semantics is defined by relationship behaviour, which is defined by users. Thus, almost any kind of aggregate with any behaviour and consistency constraints may be defined. This notion of aggregate is used to describe a software process model (that is a complex object) as an aggregate linked to its components (activity classes) by specific relations. These specific relations are named “aggregate_relation”. The "*evolution policy*" of the process model is defined in this aggregate_relation. In this regard, using the Adele trigger mechanism corresponding to the relations (propagation effect on the relation), any change to a process model gives rise to actions that update the process model and maintain its consistency (the aggregate).

For instance, when deleting an activity class from a process model by using the updating operation ‘‘delete_activity (Process Model Name, Activity Class Name)’’, the appropriate triggers will be executed by propagation effects on the relation. These triggers are only those specified in the relations which link this process model to its activity classes (i.e., the aggregate_relation that links the aggregate (process model) to its components (activity classes), and that define the evolution policy).

By default, an aggregate_relation class named **process_def** links a process model to its activity classes (Figure 3). It defines the evolution policy as follows:

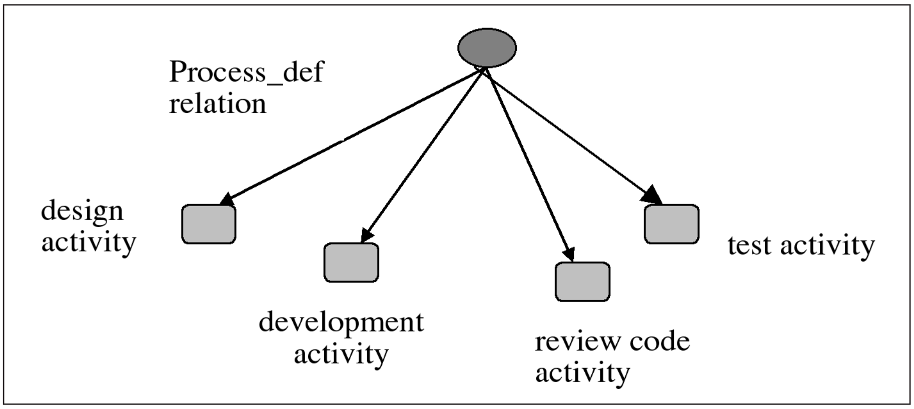


Figure 3: Process Model Aggregate

```
1. relation process_def from process_model to activity extends define action, check_sub-
   activity, check_new_classname…
2. relation define_action from process_model to activity
3. rule delete when delete_activity (ProcessModelName, ActivityClassName) on destination
4. {for t in destination ->process_def
5. {if  (t == ActivityClassName) rmr (this);}}
6. rule rename when rename_activity (ProcessModelName, ActivityClassName,
   NewLocalName) on destination
   ………..
   rule add_sub when add_subactivity (ProcessModelName, ActivityClassName,
   Main_ ActivityClassName) on destination
   ……….
   {...}
   end define_action;
7. relation check_subactivity from process_model to activity
8. rule delete_sub_act when delete_activity (ProcessModelName,
     ActivityClassName) finish on destination

     {for t in destination->process_def
               {if (t == ActivityClassName)
               for  i in ActivityClassName <- isa
     {if (i->isa == ActivityClassName)
               delete_activity (ProcessModelName, i}}}
   end check_subactivity;
```

– *Process_def* (line 1) is defined as an aggregate relation that inherits from multiple relation classes: *defined_action, check_subactivity, check_new_classname*, etc.

– *define_action* (line 2) defines all the possible updating operations. Each updating operation is associated with an event (*delete_activity, rename_activity, add_sub-activity…*) using a trigger rule. For instance, the changing or the updating operation that deletes an activity class is associated to the trigger rule "*delete*" (line 3). This trigger is executed when the event "*delete_activity*" occurs.

The semantics of the updating operations are defined in the relations: *check_sub-activity, check_new_classname, etc*. For instance, the semantics defined in the relation *check_subactivity* (line 7) expresses the fact that when an activity is deleted, its sub-activities are also deleted unless they inherit (i.e. they are also sub-activities) from other activities.

As the aggregate relation *process_def* inherits from these relations and from *define_action (line 1)*, each change on the process model results in the execution of the appropriate trigger rules.

For instance, when deleting an activity class from a process model "*delete_activity (ProcessModelName ActivityClassName)*", the delete action (line 3) is validated when the constraint (rule *delete_sub_act* -line 8) , which is also associated to the same event *delete_activity*, is checked . This rule (post-condition *finish*) is defined in the relation check_subactivity.

– Operators "<-" and "->" make it possible to navigate through relations. "->" means to navigate through the relation, from its origin (named *origin*) to its destination (named *destination*) and "<-" the reverse.

– The expression X->ISA means the set of all the destinations of the ISA relations where X is the origin. Thus, X->ISA means the super classes of X and X<-ISA all the subclasses of X. The operator "==" is the complete set equality.

– In line 4, *destination->process_def* means all the destinations of the process_def relation. As the origin of the process_def relation represents the process model (an update operation is always specified according to a process model), *destination* refers successively to all the activity classes of this process model.

– Lines 4 and 5 mean that when the link between the process model and its activity class (to delete) is detected then this link is deleted ("*rmr this*") . "*this*" means the current object or the current relation.

4.3 Generation of New Evolution Policies.

To generate a new evolution policy, one has only to create a new class of an aggregate relations. This new class will inherit actions described in the *define_action* and will define new evolution semantics. These new evolution semantics are simply defined by the inheritance of existing relations or directly redefined in new relations.

For instance, to move from the default evolution policy to a new evolution policy, the manager should create:

- new relation classes in order to express the new evolution semantics. (for example the relation classes: *no_instance, connect_to* and *no_rename*),
- and a new aggegate relation: *process_new* that inherits these new semantics and that inherits from *define_action* relation.

Relation *process_new* from process model to activity extends *define_action*, *no_instance*, *connect_to*, *no_rename…*

4.4 Process Model Consistency

When a process model is modified, consistency constraints are applied to ensure that the new semantics introduced by the manager does not allow the creation of process inconsistencies.

Respecting these constraints is fundamental to the process model consistency. Examples of these constraints are:

– the deletion of an activity class is not allowed if one or more of its associated activities are under execution (the activity state is ACTIVE).
– the change of a state nature into the activity state diagram (STD) is not allowed if an activity is characterised by this state during process execution, etc…

5. SOFTWARE PROCESS EVOLUTION

This section presents the second aspect of the evolution approach. It concerns the evolution of software processes. This evolution, complementary to the evolution of software process models, allows direct action on the process execution. This will allow efficient correction of inconsistencies found during the process execution and control the process execution with regard to the different possibilities to:

- modify the execution (modification of data flow or control flow, addition, suppression and substitution of activities, etc),
- simulate process execution using consultations and appropriate modifications of the process in order to perfect process models,
- define the process by the dynamic creation of entities (activities, products, control flow, etc).
- consult the state of the different processes during the execution.

5.1 Nature and Semantics of Updating Operations

The taxonomy of the updating operations on software processes is as follows:

- to create, delete and rename an activity,
- to insert and move an activity,
- to create, delete and rename a product,
- to transfer, share and copy a product,
- to create, delete and rename a Control Flow,
- to create, delete and rename a Data Flow.

The semantics of the main updating operations is as follows:

Creating an Activity:
New_activity (ActivityName, ActivityClass, Main_Activity, Agentname, Role)

An activity (*ActivityName*) is always created according to its class (*ActivityClass*). *ActivityClass* is one of the multiple activity classes defined in the software process model. Once created, the activity belongs to the workspace of its main activity (*Main_Activity*). The workspace of an activity represents the sub-activities, the products, the agent associated with this activity as well as its different connections of data flow and control flow. By default, the main activity is the *ROOT* activity.

An activity is created with an *INACTIVE* state. It can be executed:

1. indirectly, through a control flow,
2. directly, therefore dynamically (command *Start ActivityName*).

AgentName refers to the agent associated with this activity. A role is assigned to him for this activity (*Role*). An agent can be in charge of one or more activities.

Deleting an Activity
Delete_activity (ActivityName, Main_Activity)

This operation allows the project manager to delete an activity (*ActivityName*). It implies the deletion of all its workspace (sub-activities, products, etc).

By default, activities that were attached to the deleted activity by a control flow are automatically reconnected by a new control flow which respects the same semantics (Figures 4 and 5). It is in some way the same for the semantics of the data flow.

Example: delete (Unit_test, Test)

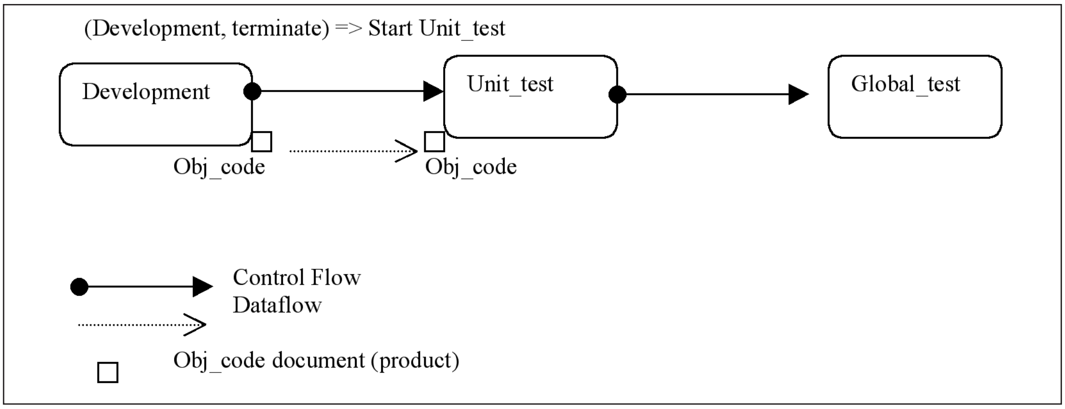

(Development, terminate) => Start Unit_test
Development
Unit_test
Global_test
Obj_code
Obj_code
Control Flow
Dataflow
Obj_code document (product)

Figure 4: Before the deletion of *Unit_test* activity

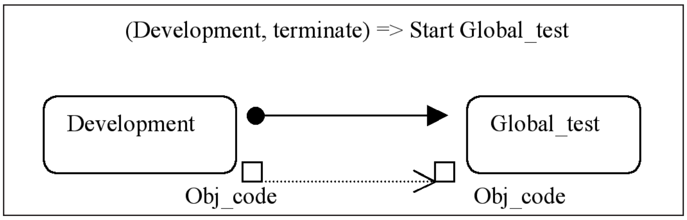


Figure 5: After the deletion of *Unit_test* activity

However, the project manager can add his own control flow or data flow semantics. This results in great flexibility since it allows the manager the freedom to create different semantics for his software process.

Inserting an activity:
Insert (Act1, New_Act, Act2)

When an activity is created, it may aim at doing new tasks among the existing activities (new needs, change of strategy, etc). It is useful to have the ability to insert an activity directly among the existing ones. In this context, according to the semantics by default, two control flow relations are thus created automatically, within the new activity (*New_Act*) and those which are adjacent (*Act1* and *Act2*). The manager can then dynamically create data flow relations according to the desired tasks.

Moving an Activity:
Move (Act1, Act, Act2)

This operation allows moving an existing activity *Act* between the activities *Act1* and *Act2*. Two operations are thus executed.

Firstly, control flow relations and data flow relations that were bound in *Act* are deleted. Secondly, the activity *Act* is inserted between *Act1* and *Act2* using the "*Insert*" operation.

Creating a New Data Flow
NewdataFlow (DF_class, Act1, Outprod, Act2, Inprod)

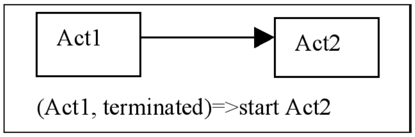
This operation creates a new data flow relation between the product *Outprod* (produced in the output of *Act1*) and the activity *Act2*. This relation defines a *transfer*, a *copy* or a *sharing* of the product according to the *DF_class* parameters. A new local name (*Inprod*) is associated to this product as input in the activity *Act2*.

Creating a New Control Flow
New ControlFlow (Act1, Act2)

This operation creates a new control flow relation between the activity *Act1* (as origin) and the activity *Act2* (as destination). This relation defines the execution sequence between *Act1* and *Act2*. The condition of this execution sequence depends on the nature of these activities:

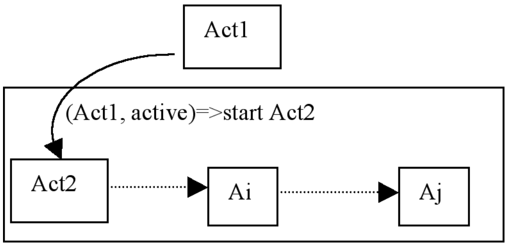
1. When the activities are in a same level, the semantic of the control flow is:

   Start the execution of Act2 as soon as the activity Act1 terminates.



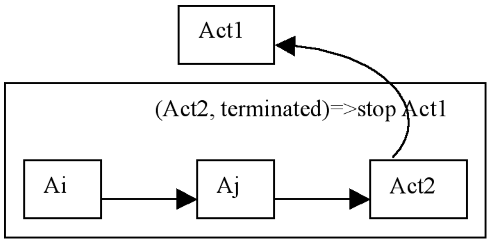
2. If *Act1* is the main activity of *Act2*, with *Act2* as the first or initial sub-activity, the semantics of the control flow is:

   As soon as Act1 becomes active (beginning of execution), then start the execution of its first sub-activity (Act2).



3. If *Act1* is the main activity of *Act2*, with *Act2* as the final sub-activity, the semantics of the control flow is:

   As soon as the final sub-activity (Act2) of the main activity Act1 finishes its execution, stop the execution of Act1 (Act1 terminates).



Deleting a Dataflow or a Control Flow
Delete (Flowclass, Act1, Act2)

This function deletes the control flow or the data flow relation (according to *Flowclass*) between the source activity *Act1* and the target activity *Act2*.

When a data flow is deleted, the product involved at the origin of the data flow remains as the output product of the source activity *Act1*.

Creating a Product
NewProduct (prodName, prodClass, ActivityName, IO_mode)

A product (*prodName*) corresponding to a product class (*prodClass*) is always created according to an activity (*ActivityName*). The product class *prodClass* is necessarily one of the product classes defined in the process model. The product is considered as input product or as output product according to *IO_mode* parameter.

Deleting an Instance Product
Delete_prod (Prodname, ActivityName)

The deletion of a product is always done according to an activity since products may have the same *local names* in different activities.

Transferring, Sharing and Copying a Product

All products can be transferred dynamically, either copied or shared among several activities.

Share (Outprod, Act1, Inprod, Act2, mode)
This operation allows the manager to share the product *Outprod* belonging to the workspace of *Act1* with the workspace of *Act2*. *Inprod* in *Act2* designates this product locally. The access mode of the product in the activity *Act2* is given by the *mode* parameter (*read, exclusive or write*).

Transfer (Outprod, Act1, Inprod, Act2)
This operation transfers dynamically the product *Outprod* of the activity *Act1* towards the activity *Act2*. The new local name is *Inprod*. The transfer is effective: the product belongs only to the workspace of Act2.

Copy (Outprod, Act1, Inprod, Act2, mode)
This operation creates a copy of the product *Outprod* belonging to *Act1* to include it in the workspace of *Act2*. The new local name of this copy is *Inprod*. The *mode* parameter indicates the access mode of this copy in *Act2* (*read, exclusive or write*).

5.2 Consultation operations

Interactive consultations during software process execution are allowed. They greatly assist to perfect a software process. The project manager may orient the consultation by a selection request on entities (activity, product…) or on their characteristics. For instance,

Listsel (Activity, Design, [Activity, state = suspended]) gives the list of sub-activities belonging to the *Design* activity and which execution has been *suspended*.

Listsel (Activity, Design, [Product, CirculationMode = share]) gives the list of the shared products in the *Design* activity.

6. CONCLUSION

This paper has dealt with the crucial problem of evolution in PSEEs (Process Centered Software Engineering Environments). The goal in this work is to define an approach that copes with multiple evolution cases, in particular in 1) finding suitable mechanisms and policies to support process evolution and 2) allowing dynamic process changes.

In this context, the presented approach is innovative since it supports different process model evolution strategies; evolution rules may be adapted (multiple semantics) to suit individual software development projects.

Another key result is the integration of two complementary evolution types, between the process model level and the process level. Thus, based on the chaining of phases, when a change is needed in a process model (first evolution type), a simulation phase is launched to validate these changes by using update operations (second evolution type) which acts directly on the process (instance). A software process model is thus updated and validated incrementally. Moreover, the possibility of acting directly on the process helps to modify and to simulate process execution to perfect software processes dynamically.

These results seem very promising considering the prototype that have been implemented. We have found a great flexibility when defining new policies and a real efficiency to perfect software processes (correct inconsistencies) when acting on the process execution.

REFERENCES

AHMED-NACER, M and ESTUBLIER, J. (2000): Schema management on software engineering databases'', *Journal of Computers and Artificial Intelligence*, 2:183–203.

BANDINELLI, S., BRAGA, M., FUGETTA, A., and LAVAZZA, L (1994): The architecture of SPADE-1- Process Centered SEE. WARBOYS, Brian (ed): *Software process technology, Third European Workshop*, EWSPT'94. LNCS 772:15–30.

BANDINELLI, S. C., FUGETTA, A. and GHEZZI, C. (1993): Software process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering* 19: 1128–1144.

BOEHM, B. (1988): A spiral model for software development and enhancement. *IEEE Computers*: 61–72.

CONRADI, R. and JACCHERI, M. L. (1999): Process modeling languages. *Software Process: Principles, Methodology, Technology*: 27–52.

CONRADI, R., NGUYEN, M., INGE, A. and LIU, W.C. (1998): Planning support to software process evolution. In *Proc Eight International Conference on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco, SU-report 1/98 USA.

CUGOLA, G. and GHEZZI, C. (1999): Software processes: A retrospective and a path to the future. *Software Process Improvement and Practice* 4(3):101–123.
DAMI, S., ESTUBLIER, J. and AMIOUR, M. (1998): The APEL: a graphical yet executable formalism for process modeling. *Automated Software Engineering Journal* 5(1):61–96.
DERNIAME, J. C., FINKELSTEIN, A., KRAMER, J. and NUSEIBEH, B., (1994): Directions in software process modeling and technology. *In Software Process Modeling and Technology*, FINKELSTEIN, A., KRAMER, J. and NUSEIBEH, B. (eds), Research Studies Press & Wiley.
ESTUBLIER, J. and DAMI, S. (1996) : About reuse in multi-paradigm process modeling approach. *In 10th int'l Software process workshop*: 63–65.
ESTUBLIER, J. and CASALLAS, R. (1994): The Adele software configuration manager. *Trends in Software*, John Wiley and Sons, Baffins Lane, Chichester, West Sussex, PO19 1UD, England, 1994, Chapter 4: 99–133.
FERNSTROM, C. (1993): Process weaver: adding process support to Unix. *In Proc. of the 2nd Int'l Conf. On the Software Process*, Berlin :12–26.
FUGETTA, A. and WOLF, A. (1996): Software process. *Trends in Software*. John Wiley & Sons, New York (4):89–100.
JACCHERI, M., CONRADI, R. and DYRNES, B. H. (2000): Software process technology and software organizations. EWSPT 2000:96–108.
JUNKERMAN, G., PEUSCHEL, B., SCHAEFER, W. and WOLF, S. (1994): MERLIN: Supporting cooperation in software development through a knowledge based environment. *In Software process modeling technology*, FINKELSTEIN and KRAMER (eds), John Wiley & Sons Inc. 1:103–129. England.
KABA, A. B. and DERNIAME, J. C. (1995): Transients change processes in process centered environments. *In Proc. of the fourth EWSPT'95*, LNCS 913 :255–259.
KAISER, G. E. and BEN-SHAUL, I. Z. (1993): Process evolution in the marvel environment. *In Proc. of the 8th International Software Process Workshop*, SHAEFER, W (ed), Wadern, Germany. IEEE Computer Society Press:104–106.
MADHAVJI, N. (1992): Environment evolution: The Prism model of changes. *IEEE Transactions on Software Engineering*, 18(5):380–392.
MINH, N. N., WANG, A. I., and CONRADI, R. (1997): Total software process model evolution in EPOS: experience report. *Proc. of the International Conference on Software Engineering*:390–399, Boston, USA.
NITTO, J. H., PATRICIA, L., SCHAEFER, W. and HALA, S. (1999): Cooperation control in PSEE. *Software Process: Principles, Methodology, and Technology*:117–164.
REIMER, W. and SCHAEFER, W. (1997): Towards a dedicated object oriented software process modeling language. *Workshop on Modeling Software Processes and Artifacts, 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland.
RUMBAUGH, J. E. (1995) : OMT: The object model. *Journal of Object Oriented Programming JOOP* 7(8): 21–27.
SUTTON, S. M., HEIMBIGNER, H., D. and OSTERWEIL, L. J. (1995): APPL/A: A language for software process programming. *TOSEM* 4(3):221–286.
SUTTON, S. M. and OSTERWEIL, L. J. (1997): The design of a next-generation process language. ESEC/SIGSOFT FSE: 142–158.
WANG, A. I., LARSEN, J. O., CONRADI, R. and MUNCH, B. P. (1998): Improving cooperative support in the EPOS CM system. *In Proceedings of the Sixth European Workshop in Software Process Technology*, Springer-Verlag:75–91, Weybridge, UK.

BIOGRAPHICAL NOTES

Mohamed Ahmed-Nacer received the PhD degree in computer science from the Polytechnic National Institute (INPG), Grenoble, France. He is a research director and is responsible for the software engineering team at the Computer Engineering Laboratory of USTHB (Algiers) University. His current research interests include process modelling, software databases and schema evolution.

Mohamed Ahmed-Nacer