Evolving Software Processes by Tailoring the Behavior of Software Objects

Noureddine Belkhatir
LGI
BP 53
38041 Grenoble France
e-mail: belkhatir@imag.fr

Walcélio L. Melo
University of Maryland
UMIACS,
College Park, MD, 20742 USA
e-mail: melo@umiacs.umd.edu

Published in the *Proc. of the IEEE Int'l Conf. on Software Maintenance.*
Victoria, Canada, September 1994.

Abstract

Software process evolution corresponds to the act of improving the existing prescriptive software process models in a controlled and supported way. As software processes change constantly, it is therefore necessary to support one or more methods for assisting environment administrators in improving models. Changes are made in order to adapt software process models to new requirements, correct inconsistencies encountered in the course of execution, and modify, add or remove certain constraints.

This article shows how software process evolution is supported in the Tempo. Tempo is a process-oriented software engineering environment where software processes are formally described in an object-oriented process schema. In Tempo, a process schema is comprised of descriptions of software agents, software products and software processes. A new approach is presented which supports the dynamic evolution of software process descriptions. In this approach, software process change is the result of tailoring the behavior of software objects manipulated during software process enaction.

Keywords: Software process evolution, process-centered software engineering environments, object with roles, object-oriented approach, event-condition-action rules, active database.

1 Introduction

A process-oriented software engineering environment (POSE) is a special kind of software tool which monitors and/or controls software processes according to the software policies explicitly described in a software process program. The descriptions, expressed as process program tell which software activities are computer-supported by the environment, who can carry out such activities and under which conditions, how software activities must be coordinated and synchronized, which software resources are used to maintain and/or develop software products and what the environment policies are for utilizing such resources, how software tools can be applied and in which circumstances such tools must be called, etc. The POSE is the element responsible for enforcing the software policies described in a process program in a software environment. In this context, it is of paramount importance to provide support for the evolution of process programs. For the rest of this paper, we will take evolution of process programs to mean tailoring or customizing the behavior of a POSE in order to adapt its functioning to new needs or new situations. Unlike previous generations of software engineering systems, such as Pact [15], where the way the software processes must be performed is hard-wired in the system by the tool constructor, evolving software process programs are the core of a POSE. Thus, by changing such program, the way the POSE interacts with software performers, controls the application of software tools, guides the enaction of software activities, allocates software resources, etc. will change, too.

Since Tempo is a POSE, software process evolution require changing the process schemas and then enforcing the changed schema. Changing process schema dynamically is the hard part and is the focus of the rest of the article. A process schema is composed of description of software agents, software products and software processes. Section 2 discusses some related work. Section 3 presents the conceptual architecutre of the Tempo system. Section 4 discusses the main characteristics of the Tempo software process mod-

eling language. Section 5 presents the mechanisms which make it possible to dealt with software process evolution in the Tempo system. Conclusions are given in section 6, with indications of further work.

2 Some related works

Versioning mechanism, such as those used by Rcs, have been used by software engineering environments (SEE's) in order to support the evolution of software artifacts. Recently data schema evolution mechanisms, such as those found in object-oriented database management systems, have been adapted to undertake object description evolution. However, so far, little software engineering research work has concentrated on supporting software process evolution.

In Prism [11] a process is managed using a change life cycle model, comprising three phases: simulation, initialization and operation. Process evolution is supported by providing a kind of spiral model. That is, if a problem is detected or a requirement change is required during process performance, Prism's change model allows the process to return from the operation phase to the simulation phase, where the process model is modified in order to stay up to date with the change. After the change, the process is again enacted and performed. Our goal is not to develop a new process change method, but to support software process engineering in a large-scale context. Unlike Prism, we have not attacked methodological aspects of software process management. However, we have implicitly provided some novel and interesting facilities for the management of change in processes and software systems by designing a new software process paradigm, the **role** approach, and building an automated environment supporting various facets of process enaction, performance and evolution. In fact, we believe that Prism's change model could be implemented using Tempo, once the two main environmental facilities provided by Prism, i.e. dependency structure and change structure [10], could be realized using the Tempo's resource manager of (i.e., the Adele database [4]) and the Tempo's activity manager (i.e. the Nomade trigger mechanism [3]).

In AP5 [13] software process model evolution is provided using trigger modification commands, the addition of new rules and the removal of existing ones. As AP5 was built using LISP, an environment administrator can change the process model without stopping software process execution. Management of consistency between modifications and the currently executing processes is delegated to those responsible for the modifications. As described in [12], Tempo also provides built-in commands making it possible to add incrementally new trigger rules to data and process schema. We discuss how Tempo provides this kind of evolution in the next section.

In Marvel [9] the process description can evolve due to changes to the pre- and post-conditions which encapsulate rules. Marvel ensures the consistency of rules in relation to the execution mechanism and in relation to the descriptions contained in its data schema [2]. In contrast to AP5, process description evolution is static; processes must be stopped in order to permit changes to their descriptions. Once it has been modified, the new description is compiled and validated. After these stages, the processes can continue execution, in a new context which takes account of the modification.

Melmac [8] provides *modification points*. If a modification point is associated with a software process step, for instance PS, during process enactment, but before PS execution, Melmac could use the procedure described in such a modification point to accomplish the required process change.

Peace [1] provides *meta-plans* which are responsible for describing procedures to be followed for modifying a software process model when an exception arises during process enaction. This mechanism allows Peace to dynamically change plan definition, e.g., replacing one plan by another, modifying plan hierarchy, etc., using the knowledge described in such meta-plans.

3 The architecture of the Tempo system

As figure 1 reveals, the Tempo environment consists of following components:

- A resource manager. The Tempo's resource manages uses Adele database as a persistent object base for storing objects and activities, and for tracing the project's progress [4]. It supports an entity-relationship data model which is extended with object-oriented concepts like inheritance, methods and encapsulation. Simple and composite objects with attributes and relationships can be described and managed.

- An activity manager which is responsible for the control integration in our platform. This activity manager is driven by temporal-event-condition-action rules (TECA) and supported by the Adele's trigger mechanism. We enhanced Adele's trigger mechanism with the ability to manipulate temporal expressions [7, 12].

- A process manager which offers the concepts of process and role. Process execution is supported by work environments (WE) wherein software activities are performed. The process manager, based on the activity manager, manages communication and synchronization between teams, and between agents involved in the same project. It also controls the consistency of complex objects used simultaneously in different work environments by different agents [7]. This component represents the conceptual component responsible for process integration in the Tempo architecture.

Adele 2 plays the role of resource manager and activity manager in the current version of Tempo [7]. Adele 2 [4] is a commercial product which is the result of the union of two long term projects in the framework of the *Laboratoire de Génie Informatique de Grenoble*. Adele 2 integrates the results produced by the Adele 1 and Nomade projects [3]. Adele 1 [3] was a version management system hard-coded with a configuration builder. Nomade was a prototype of an active software engineering database. This database was driven by an object-oriented data model. The active part of this database was supported by a trigger mechanism, which was driven by event-condition-action rules. Nomade incorporated the version management system of Adele 1 for dealing with the evolution of software artifacts in versions. Adele 1's configuration manager was also included in the nucleus of Nomade. Tempo [12] is the successor of Nomade. Tempo is able to deal with user defined software process models, multi-points of view of software artifacts, temporal events, long-time duration activities, and it provides support for communication of software activities. The concepts and mechanisms proposed by Tempo are going to be incorporated into Adele 3, which is will be the new commercial version of Adele.

4 The Tempo product and process definition language

This section describes the main characteristics of the software process modeling language used by Tempo for describing software products and software processes. Readers familiar with the Tempo [6, 7, 12] can skip this section.

4.1 Product modeling

Software products are described using the Adele data model and supported by the Adele database. The Adele data model is based on the entity-association model and integrates object-oriented concepts. The basic entities of the model are object type and relationship type.

The data model supports complex objects referred to as aggregates. An aggregate is an object linked to its components by relationships. For example, a Pascal module can consist of an interface and an implementation. The Pascal module object can consequently be represented as an object linked to two other objects by two types of relationship, possesses-interface and possesses-implementation. Aggregate semantics are defined by the dynamic properties of the relationship linking the aggregate to its components. The semantics are defined by the user; any aggregate can thus be defined, using its own semantics and consistency constraints.

4.2 Process modeling

A software process model of considerable size can be written by grouping of various software process types. A software process type has a recursive definition. It is a mixture of several software process types. For example, an activity to check a module design document consists of two sub-processes:

1. A sub-process which models the modification activity allowing modifications to the design document.

2. A sub-process which models the revision activity allowing approval of any design document modifications which have been made.

```
MonitorDesign ISA PROCESS;
   CONTROL md;
      sub = ModifyDesign;
      card = 1;
   CONTROL rd;
      sub = ReviewDesign;
      card =1;
END_OF MonitorDesign;
ModifyDesign ISA PROCESS;
   ATTRIBUTES
      begin_date = DATE := now();
      end_date = DATE;
      deadline = DATE;
   METHODS . . .
   RULES  . . .
END_OF ModifyDesign;

ReviewDesign ISA PROCESS; ...
```

The example above shows the software process type "`MonitorDesign`", composed of the sub-processes

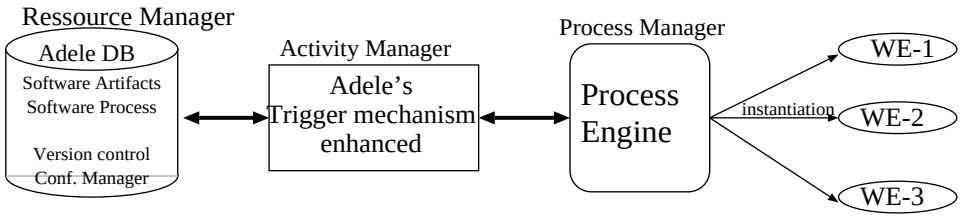


Figure 1: Conceptual architecture of the Tempo POSE.

"`ModifyDesign`" and "`ReviewDesign`". The activity coordinating the module design document modification is represented in the Tempo formalism by the "`MonitorDesign`" type. This is composed of two sub-processes: "`ModifyDesign`" and "`ReviewDesign`". "`ModifyDesign`" is the type which describes the design document modification process, and "`ReviewDesign`" is for revising this modification.

It is possible, for every process type, to define attributes, methods and temporal constraints by using the temporal event-condition-action rules.

4.3 Temporal constraints

The flow of the software production process is controlled by temporal constraints. For this, we need, on the one hand, to provide a conceptual framework allowing the tracing and persistency of anterior (past) states and on the other hand, to describe and verify temporal constraints during the execution of software processes.

4.3.1 The "temporal-event-condition-action" rules

The temporal constraints are described in Tempo software process language by temporal-event-condition-action rules (TECA). TECA rules are defined both in the data model and in the activity model. They are inherited in the hierarchy of types. In the data model, the TECA rules describe integrity constraints independently of the context of utilization. In the activity model, these rules are used to express software development policies: the execution order of activities, their synchronization, and constraints above the use of software resources.

A TECA rule that goes like this:
"WHEN *event* Do *Method*"
where:

event is a predicate expressing an event about the present or past state of the system or about the object base.

method is a method.

Example:

```
EVENT
 delete_sensible = (!cmd == remove AND
       (!object\comp/state == released OR
       !object@(status == validated));
```

This line expresses that event "`delete_sensible`" will be true whenever there is an attempt to delete a component (`!cmd == remove`), which is either a component of a released configuration (`!object/comp/state == released`) or which has been in the past the status validated (`!object@(status == validated)`). The expression "`!object`" represent the name of the object receiving method "`!cmd`". Similarly all parameters of the called method can be checked, as well as previous values of attributes and object when changed by the methods.

We added the operator "@" in the expression defining the event in order to be able to lay conditions on the past. This operator is interpreted in relation to the log of object evolution. All updates performed on an object is stored in this log (changing of attributes and events). Temporal constraints are checked following a reverse scanning of the history from the triggering of the event to the satisfaction of the temporal constraint. These constraints are expressed in relation to object properties (attributes and events stored in the objects log). If temporal constraints are not checked at any time at all, then no operation will be executed.

4.3.2 The methods

A method is program written in a simple imperative language similar to Unix's

```
METHOD delete ;
   IF [%state == stable] THEN ABORT
   ELSE "rmobj %name ";
END delete;
```

This method enables to suppress objects with unstable states. The late-binding mechanism is used during the execution of methods. In the above example, when the method "delete" is executed, the variable *"%name"* takes the identifier of the object being deleted as value.

4.3.3 Examples of utilisation of TECA rules

Definition of TECA rules into data model

Figure 2 presents an example of use of TECA rules in the data model. TECA rules describe constraints about the manipulation of software objects. Such rules are independent of the context where and when software objects are handled. In order words, TECA rules when defined in the data model are useful for description of (1) integrity constraints about object relationships, and (2) software policies which are context independent or invariants.

In this *body* type description we find in lines 1 the definition of attribute *lines* which represents the number of lines in the body. *Lines* is declared *COMP* which means the value provided at instantiation is not the attribute value but the program that, when executed, will return the real attribute value. In line 9 the value of line is the result of the execution by Unix shell of *wc -l !filename* i.e the number of line in file *!filename*.

Line 2 is a pre-condition which specifies that if the event `delete_official` occurs, the command which triggered this event must be aborted. Event *delete_official* in defined line 12 occurs when the command `delete` is applied to an official body (i.e. an object body with attribute state equal to official). Line 3 expresses a post-condition on event `replace_body_c` defined in line 13. When the command `replace` is applied to a c program body (an object with the attribute language equal to c) this program must be compiled. If compilation is successful (line 9) the binary object is recorded with its source code (line 10) and the line numbers of the source object is computed and recorded (line 11).

The relation `comp` relates a configuration with its components. Before replacing a component of a configuration (line 5), the number of lines of the configuration (`!O` refers to the origin of the relation i.e. the configuration), is reduced by the number of line of the component (`!D` refers to the relation destination i.e. the replaced component, `!DD%lines` is the value of attribute lines of the component); after the replace command (line 7), the actual number of line of the component is added to the number of lines of the configuration (line 8). That way, the number of line of all configurations is always up to date and recursively.

Description of TECA rules into process model

On the other hand, TECA rules when defined in the process model they (1) describe fragments of software activities, (2) specify software policies which are context dependent, (3) define ordering of software activities, and (4) pre- and post-condition about the actions of user performers. For instance, figure 3 gives a fragment of software process definition, where:

1. The rule described in line 1 specifies that the design document modification activity must stop when the date foreseen has been reached.

2. The rule in line 2 states that resumption of the activity if has not been completed yet first requires that the termination date be changed.

```
ModifyDesign ISA PROCESS;
  ATTRIBUTES
    begin_date = DATE := now();
    end_date = DATE;
    deadline = DATE;
  METHODS
    continue_execution;
    . . .
  RULES
(1) AFTER WHEN deadline_arrived
      DO stop_execution;
(2) PRE WHEN (continue_execution AND
              @(not deadline_changed))
         DO ABORT;
END_OF ModifyDesign;
```

Figure 3: An example of the utilisation of TECA rules in the process model

4.4 Object with roles

4.4.1 Motivation

The problem of multiple perspectives or multiple viewpoints often occurs in the lifetime of a software. In this case, users handle objects simultaneously, use different viewpoints of these objects, and carry out actions limited and directed by the constraints of their own activities. These users, directed by multiple development strategies, handle different models of the same product.

```
  TYPEOBJECT body ;
     DEFATTRIBUTE
1       lines COMP = INTEGER;
2    PRE WHEN delete_official DO ABORT;
3    POST WHEN replace_body_c DO
4         "store_binary %name" ;
  END body;

  TYPERELATION comp ;
5 PRE  WHEN DEST replace_body_c DO
6         "modify_attr !O -a line-conf = %line-conf - ~!D%lines" ;
7    POST WHEN DEST replace_body_c DO
8         "modify_attr !O -a line-conf = %line-conf + ~!D%lines" ;
  END comp ;

  DEFACTION store_binary;
9    IF "cc -c !filename" THEN
10     {"replace %name -do" ;
11      "modify_attr %name -a lines = \"wc -l !filename\"" } ;
  END store_binary;

  DEFEVENT
12   delete_official = [ !command=delete, state=official ];
13   replace_body_c =  [ !command=replace, language = c ];
  END
```

Figure 2: An example of the utilisation of TECA rules in the data model

A SEE must therefore provide a framework permitting the description and control of these aspects in the environment. Tempo offers concepts allowing the description and structuring of multiple viewpoints. Basing on the rule concept and for each object handled, every occurrence of software process can have constraints (TECA rules), local operations (methods) and local properties (attributes). For example, a module belonging to the Pascal object type, M1, has properties and constraints inherited from this type. Via the role concept, a module M1 can have new properties, new methods and new temporal constraints based on its role in an activity. For example:

```
TYPEOBJECT C_body ISA body;
  METHOD
    compilation; # With debug option
    link;
END C_body;

test ISA PROCESS;
 ROLE under_test;
   derived_from := C_body;
   METHOD
     compilation;
       # without debug option
END_OF test;

integration ISA PROCESS;
  ROLE under_integration;
    derived_from := C_body;
      METHOD
        compilation;
          #  without debug option, but
          #  with optimization option
END_OF integration;
```

The above example shows that the objects of the `C_body` type can be handled differently depending on the role they play. Objects of `C_body` type acting as under-integration in an integration process will be compiled differently from the one described in the `C_body` type. Likewise , when these objects act as under-test in a test process, they will be compiled differently.

Roles are typified. A role type can refer to different types of objects. This allows to integrate many behaviors and properties, coming from different types of objects, in a unique view. By using this concept, Tempo allows enables us to unify the processing of a hetero-

geneous set of objects. The advantage of this strategy is that, using the role concept, a set of objects having different static and dynamic characteristics can be perceived in a homogeneous manner, during the execution of a particular software process phase. This homogeneity is maintained by multiple inheritance rules used in the object-oriented models.

For example:

```
test ISA PROCESS;
  ROLE under_test;
    derived_from := C_body;
      METHOD
        compilation; # without debug option
  ROLE interfaces;
    derived_from := C_interface,
                    CPP_interface;
      METHOD
        list;
END_OF test;
```

The `C_interface` types (C programming interfaces) and `CPP_interface` (C++ programming interfaces) are specialized in the test process via the "*interfaces*" role. Objects of the "`C_interface`" or "`CPP_interface`" type playing this role will be handled by the methods described in the role. Therefore, the list method can be applied both to the C interfaces and to the C++ interfaces. Many roles can be described by a software process which then becomes a list of roles where every type of object can play different types of roles. As a result, two objects of the same type can be managed in different ways in a software process. Parallel to this, the same object can play different roles in different software processes.

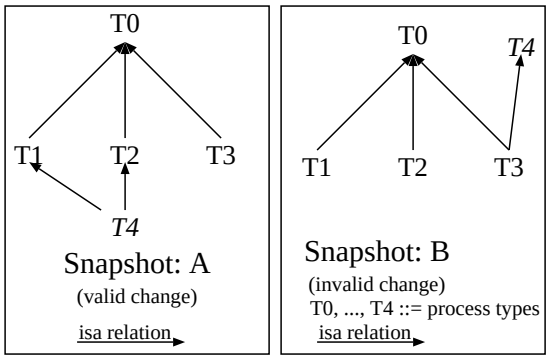
5 Process evolution support in Adele/Tempo

5.1 Adding process types to a process schema

The Tempo process language is based on the object-oriented approach where process types, data types and connection types are defined in a process schema and "instantiated" as objects in the Adele database [12]. The consistency of the process schema is assured by standard multiple inheritance rules. Furthermore, Tempo imposes the two following consistency constraints:

1. types can be added to a process schema if and only if the added type will not change the inheritance graph of the already defined types.
2. Process types with enacting software process occurrences cannot be removed from the process schema.

For example, the following figure shows two process schema changes:



The Snapshot A shows an example of a valid process schema change. In this snapshot the T4 process type has been added to the current process schema. The already instantiated types have not changed with the T4 inclusion. The Snapshot B shows an example of an *invalid* process schema change, because adding process T4 would change the inheritance graph of the already instantiated process types. (Tempo provides pre-defined commands making it possible to build up a process schema incrementally [12]. These commands can be only used by privileged users [5]).

5.2 Updating Temporal-event-condition-action rules

Adele/Tempo provides pre-defined commands to incrementally change rules, events and methods defined in the process schema [12]. These commands can be only used by authorized users [5].

The body of methods already defined in the process schema can be modified without triggering modification of the enacting software processes occurrences.

Constraints can be relaxed by changing the event clause of the Temporal-event-condition-action rules (TECA rules). As well, rules can be added to or removed from a process schema, independent of the existence of already enacting processes belonging to types under change. Given the following process schema:

```
DEFEVENT e1 = {condition1};

TPROCESSTYPE P1; ...
rule1:     WHEN e1 DO m1;
END P1;
```

The rule1 associated with the P1 process type can be relaxed by adding another event to the process schema and composing the condition part of that rule, for instance:

```
DEFEVENT e1 = {condition1};
 e2 = {condition2};

PROCESSTYPE P1; ...
rule1:     WHEN e1 OR e2 DO a1;
END P1;
```

As well, new rules can be added to or removed from a process schema, independent of the existence of already enacting processes belonging to types under change. For instance:

```
DEFEVENT e1 = {condition1};
 e2 = {condition2};

PROCESSTYPE P1; ...
rule1:     WHEN e1 OR  e2 DO m1;
rule2:     WHEN e1 AND e2 DO m2;
END P1;
```

The rule number 2 has been added to the process schema. This rule says that when the e1 event and the e2 event occur, the m2 method will be triggered.

TECA rules are always interpreted, never semantically compiled in Adele/Tempo. A late-binding mechanism is heavily used during rule interpretation. Backward chaining is not implicitly supported by Adele/Tempo, i.e, rules are not interpreted in order to achieve a user-defined goal. Due to these two characteristics, TECA rules are, unfortunately, added to and removed from the process schema without consistency verification, unlike other rule-based system, such as Marvel [2], where the pool of rules is supposed consistent.

5.3 The role concept and process evolution

Tempo process programming language is heavily based on the role concept. A role, as well as a type, is a template applied to a set of instances sharing the same definition (static and behavioral). A given object instance can be simultaneously a member of different roles (classes). Both roles and types can be seen as a viewing mechanism since a given object instance has a different description depending on the role from which it is managed. One would need to create a sub-type for all the possible combinations of roles for a single type, and to change instance type dynamically each time a new role is applied to it. However, there is a fundamental difference:

> The association between an instance and its type is statically defined at instantiation time, while an instance can be dynamically bound to an arbitrary role at any time.

Furthermore, since the instance may shared and play different roles simultaneously, dynamic typing cannot be used. We introduce the possibility to changing type dynamically. In an usual O.O. DBMS the type definition is created first, and then the instances of the types. In Tempo on the other hand, the instances are usually created first, and are dynamically associated, for a while, to a (set of) role(s).

Since a given object can be simultaneously a member of different roles there are compatibility rules between the roles allowed for shared objects. In Tempo, objects can change behavior depending on the context without changing identity.

The role concept makes it possible to evolve the characteristics of software objects handled during the software processes. The role concept naturally integrates a type evolution facility, since role types are similar to object types in O.O. languages. The role concept offers two kinds of evolution:

1. role definition can change generating role version;
2. objects can change its roles dynamically.

Adding roles to enacting process

Role types are used to model the life cycle of a software object. This raises the problem of how to integrate new role types which correspond to unforeseen situations (exceptional situations). This approach may also be used to allow for incremental development of the life cycle of objects, gradually defining roles. An object participating in an activity has a structure and behavior which complies with its role. Roles provide a way of tracking the various facets of object evolution.

For example:

> A software object representing a software configuration evolves in conformance with its life cycle: specification, construction, test and qualification. For each of these stages, we can define "*on the fly*" a corresponding role:
>
> **in the specification role,** configuration properties are defined;

in the construction role, there is a description of how the configuration should be generated;

in the test role, procedures for adjusting the configuration are defined (compilation, linking and testing);

in the qualification role, the configuration is approved prior to being made available to users.

Using this approach, the role type casting of an object represents stages in its life cycle. Transitions from one stage to the next are governed by Temporal-event-condition-action rules which are explicitly defined by the environment administrator. For example:

> The administrator will only add the qualification role type to the configuration installation process once the configuration has been tested and approved.

Let W be an occurrence of the P software process type currently being executed (see figure 4). The objects manipulated in this occurrence are visible and behave in compliance with the roles defined in P.

Adding the R1 role type to P triggers a change in the organization of the objects manipulated in W, as certain objects which were playing R0 role in W will now be playing R1.

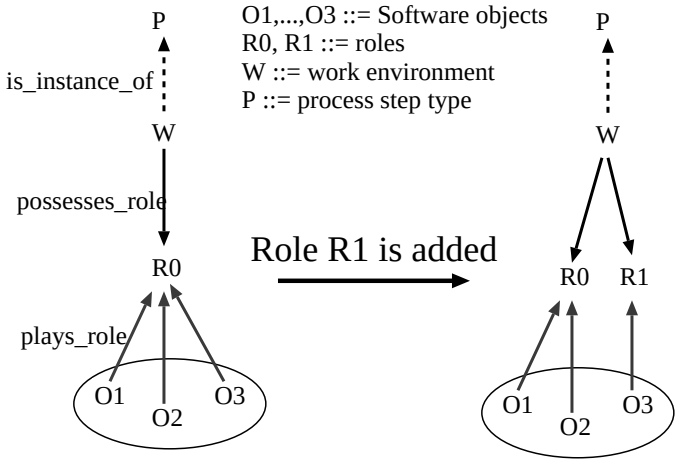


Figure 4: Evolving software process by role adding.

The role casting has two effects on process evolution:

1. An existing object, O3, playing a given role, R0, in a work environment, W, will have a new point of view, once it has changed its role to R1.
2. The behavior of the O3 object in the new R1 role changes, because new and different methods and rules can have been described in R1.

For an object, the switch from one role to another is dynamic; the switch causes the software process to evolve. A new type of role is dynamically added to the software process schema. At same time, the object changes its role thus changing its behavior. In order to control these changes, Temporal event-condition-action rules can be defined in the software process types.

6 Conclusion and perspectives

This article has presented evolutionary features in a Software Engineering Environment. Evolution is taken into account in both the software product model and software process model.

Changes at the software product model include changes to the schema where software object and relationships between such objects are defined. At present, we are using the same evolution mechanism proposed by Orion, which triggers database reorganization when the object schema is modified.

Software process changes make it possible to develop incrementally software process models. This evolution is achieved by dynamic addition of roles types. A role can either be an object view or a sub-process (role aggregate). An object view is a definition of a set of encapsulated rules in a consistent unit, or in other words a sub-process. In this way the evolution consistency of the whole depends on adding roles with transitions implemented by temporal rules, controlling the flow of role instances. Flow can be intra-process (evolution of a role inside a sub-process) or inter-process by adding sub-processes.

By developing a software process language based on an object-oriented approach, extended by the role concept, makes allowance for evolution an inherent part of software process models. The major asset resides in the fact that role concepts and temporal rules allow evolution to be expressed naturally, including the transfer of the affected instances.

Work is moving forward in these areas, both in terms of research and development:

- Implementation and integration of schema evolution in the commercial version of Adele is under way.
- Research work is currently striving to bring the ways product and process evolution is viewed

closer together. It seems likely that models based on object roles may be an answer to support both product and process modeling and evolution. This would make it possible to unify process and product visions. The process level is one level of abstraction above the product level, providing support for activities by organizing them around the role concept.

We therefore consider that this constitutes a reasonable basis for specifying evolution in our software development environment without claiming to cover all possible forms of software process evolution. Our principal concern is to avoid chaotic situations caused by uncontrolled evolution, resulting in the long term de-structuring of software products and uncontrolled deviation by the processes they produce.

In our opinion, integration by the data and the software process represent a reasonable basis for integrating and supporting an evolution process.

Acknowledgements

We would like to express our thanks to Barbara Swain for suggesting substantial and helpful revisions to the original text.

During this work, Walcélio L. Melo was supported by the Technological and Scientific Development National Council of Brazil (CNPq) under grant No. 204404/89-4.

References

[1] S. Arbaoui and F. Oquendo. Peace : goal-oriented approach and nonmonotonic logic-based formalism for supporting process modeling enaction and evolution. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press, 1994.

[2] N. S. Barghouti and G. E. Kaiser. Scaling-up rule-based development environments. *Int'l Journal on Software Engineering & Knowledge Engineering*, 2(1):59–78, March 1992.

[3] N. Belkhatir. *Nomade : un noyau d'environnement pour la programmation globale*. Thèse de doctorat, INPG, Grenoble, France, 1988.

[4] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In M. Dowson, editor, *Proc. of the First Int'l Conf. on the Software Process*, pages 159–170, Redondo Beach, CA, October 21–22 1991. IEEE CS Press.

[5] N. Belkhatir, J. Estublier, and W. L. Melo. User modeling and control in the Adele system. In *Proc. of the 4th Int'l Conf. on Computing and Information*, pages 334–337, Toronto, Ontario, Canada, May 28-30 1992. IEEE CS Press.

[6] N. Belkhatir, J. Estublier, and W. L. Melo. Software process model and work space control in the Adele/Tempo system. In L. Osterweil, editor, *Proc. of the 2nd Int'l Conf. on the Software Process*, pages 2–11, Berlin, Germany, February 1993. IEEE CS Press.

[7] N. Belkhatir and W. L. Melo. Supporting software maintenace processes in Tempo. In *Proc. of the Conf. on Software Maintenance*, pages 21–30, Montreal, Canada, September 1993. IEEE CS Press.

[8] W. Deiters and V. Gruhn. Systematic development of formal software process models. In *Proc. of the 2nd European Software Engineering Conf.*, Univ. of Warwick, Conventry, UK, September 1989.

[9] G. E. Kaiser and I. Z. Ben-Shaul. Process evolution in the Marvel environment. In Schafer [14].

[10] N. H. Madhavji. Environment evolution: The Prism model of changes. *IEEE Transactions on Software Engineering*, 18(5):380–392, May 1992.

[11] N. H. Madhavji and W. Schafer. Prism — methodology and process-oriented environment. *IEEE Transactions on Software Engineering*, 17(12):1270–1283, december 1991.

[12] W. L. Melo. *Tempo: Un environnement de développement Logiciel Centré Procédés de Fabrication*. Thèse de Doctorat, Université Joseph Fourier (Grenoble I), Laboratoire de Génie Informatique, Grenoble, France, 22 de Octobre 1993.

[13] K. Narayanaswamy. Enactment in a process-centered softwre engineering environment. In Schafer [14].

[14] W. Schafer, editor. *Proc. of the 8th Int'l Software Process Workshop*, Germany, 1993. IEEE CS Press.

[15] I. Thomas. Version and configuration management on a software enginnering database. *ACM*

Software Engineering Notes, 14(7):23–25, November 1989.