TEMPO:
a Software Process Model Based on Object Context Behavior

Noureddine Belkhatir Walcelio L. Melo Jean-Michel Adam

Laboratoire de Genie Informatique
BP 53X, 38041 Grenoble Cedex France
{belkhatir, wmelo, adam}@imag.imag.fr

Published in the *Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*, Toulouse, France, December 7–11 1992. pp. 733–742.

Abstract

Recent work on software processes has produced a considerable amount of detailed information which renders the software life cycle more explicit by describing it as an enactable software process model. This article presents TEMPO: a software process modeling strategy based on Adele: a software configuration management kernel. The facilities for describing and enacting software process models are highlighted. TEMPO is an object oriented process model. Each software process is modeled as an object which encapsulates (role concept) the operations around a set of resources required to carry out a specific software development activity. Each software activity provides workspace in which the developers work by calling operations on processes. On this way a software development environment may consist of a set of workspace working together by coordinating their activities. This paper concludes with an overview of the TEMPO implementation on top of Adele.

Keywords

Process model, Configuration management, Object Orientation, Roles, Ressources, Workspaces, Tempo, Adele

1 Introduction

Recent work on software processes has produced a considerable amount of detailed information which renders the software life cycle more explicit by describing it as an enactable software process model. We claim that software configuration management (CM) functionalities are highly relevant in this context since CM is evolving to provide support for many software development activities, i.e. software processes [6]. In this paper, Software Configuration Management is outlined, not only as a tool that provides services, such as version and access constraints, for controlling software system changes, but as a software engineering environment which supports several aspects concerned with software system evolution activities. In this paper, we shall show that a CM system can be considered as process-centered software engineering, wherein software configuration policies can be described and enforced automatically.

As CM is a discipline for managing changes, it involves change control and software process management (PM). The integration of these shared services is implemented generally by a two-level database architecture. A primary database supports software product evolution and provides a permanent repository of objects. This is the CM WorkSpace (WS). In turn, many secondary databases act as PM workspaces, which define and support the context where CM activities are carried out by different software engineers making up the CM team.

This architecture raises the general problem of integration of CM functionalities with PM needs.

- Objects are exchanged between the CM-WS and PM-WSs. How should CM-WS be extended to PM-WS to facilitate definition of WS policies without cumbersome communication protocol exchanges.
- the problem of consistency maintenance when multiple copies of objects are distributed across different work spaces.

In this article, we show how these important problems are resolved in the Adele system. We propose a solution where the combination of a software process model with a workspace mechanism provides the necessary flexibility to support CM and PM functionalities. This approach is based on the **role concept**, which defines a software process step as a list of objects playing a role. An object's behavior depends on the role it plays in the software process step, and may be part of different simultaneous work spaces. We show how communication and synchronization policies can be captured by our formalism and enforced by the mechanisms provided by Adele system.

2 Modeling and enacting software processes

When a software product is introduced as a managed Adele project, the follow abstraction levels can be used in order to model it more sucessfully.

1. Product level. In this level the software entities, the characteristics of these entities and the relationships between them are captured and modeled using the Adele's data model. A product model specially designed to support modular programs is also provided. In this level an object type is characterized by: (1) a set of standard attributes, (2) a special long field attribute (used to record software objects), (3) a set of integrity constraints describing those invariants that must be respected during software product life, and (4) a set of methods which can be applied on object instances.

2. Process level. From the product model, the software types are tuned to satisfy the activity requirements in the process level. As we shall see later, to satisfy the specific request of each process step, original database types could have its characteristic and behavior changed. New methods and rules can be respectively overloaded and incorporated, as well the software characteristics (entity and relationship attributes). On this way, process model provides a software development environement in which software enginers are working concurrently in their workspace to carry out specific goals. Each workspace consists of different set of ressources depending on the task to be performed. This adaptation of ressources to the performed tasks is done using **role concept**.

The first level has already been described [1, 2]. In the remainder of this section we shall present the second level which is the process model named **TEMPO** .

3 Design issues

Although, a large portion of software process cannot be mechanized, because software process is before all a creative activity, there is a lot of actions performed by the software engineers when carrying out their activities that need automation, e.g. project planning enforcing, resource allocation monitoring, progress control, change control, user guidance, activity synchronization, historic management etc.

From our point of view, a software process can be modeled as a combination of sub-processes or process steps. A process step characterizes a well accurate set of controlled activities, identifiable in the software process model. Roughly speaking, a process step is carried out by one or more users. It is always linked to set objects, which compose one sub-database (workspace). It has rules giving the manipulation policies that govern the use of these objects inside the software process step execution context. These rules are called *private rules*.

We see software processes as a set of activities executing concurrently and asynchronously. In our case, the communication and synchronization protocol between software process steps is described by event-condition-action rules, because our approach is event-processing based. This means, that each manipulation made on an object during the software process life generates events. Rules defined in a process step make it possible to take account of these events. Thus, synchronization between process steps is accomplished by capturing and processing the events provoked by manipulation of shared objects. These rules are called *react rules*.

In order to satisfy the process step requirements, we need to change the characteristics and the behavior of the manipulated objects, i.e. depending on the context where an object is used, its role is different. We introduce the "role" concept to describe this kind of object customization. A role make it possible to change the definition of attributes, methods, and constraints of an object type for a process step. That is, the role is a new definition of the properties and the behavior of an object type in a process step.

Each software process step is described in the data model by a process type. Below, we give the syntax of a typical process. In the remainder of this section, we shall explain each clause in detail.

```
TYPEPROCESS process-name IS ListSuperTypes;

 { ROLE role-name  = list-of-types  ;
```

```
     [ ATTRIBUTE list-of-attributes]
     [ METHOD    list-of-methods ]
     [ PRIVATE-RULES   list-of-rules ]
     [ SHARED-RULES    list-of-rules ]}+
END process-name;
```

A software process step is recorded in the Adele-DB as a standard object type. Thus a process entity can be instantiated, characterized by attributes, revised, removed and connected with other process or software entities. Thanks to multiple inheritance mechanisms, a process type can be refined and specialized. New attributes, roles, methods and rules can be overloaded and modified. Therefore, process customization will be achieved by process type specialization.

3.1 The role concept

To define a role, we give it: a name, a reference type, characteristics or local attributes, a list of methods, as well as a list of both private and react rules.

The role name is used as the internal identification of either the original reference type or other role name (in section 3.2 we shall show as a **role** references another role). The reference the type to which the role is related. For example, figure 1 shows how the module type (from the product model) is referenced inside two process steps: in the `WS-change` by the `view role` and `WS-valid` by the `to_valid` role. During `WS-change` process step enactment, the `view role` will be bound to the module instances manipulated by this process step. This example also shows how attributes can be added to satisfy the needs of a specific software process step. The attribute `status` is defined to satisfy the requirements of the `WS-change` process step, and the up-dates made inside the execution context of this step to `status` attributed will be limited to `view role`, in this way other software processes that manipulated the same module instances will be not affected by these up-dates.

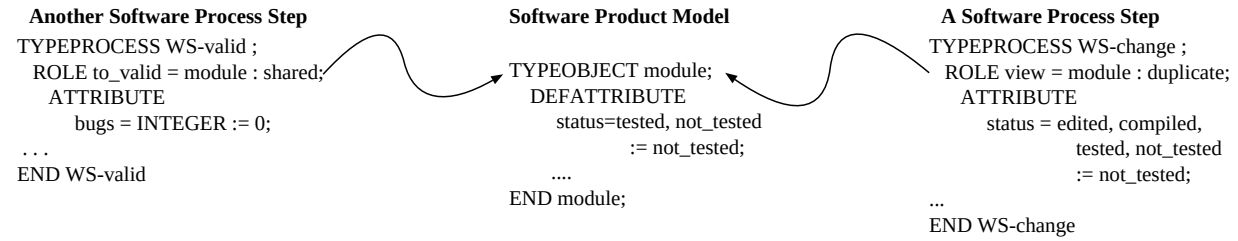
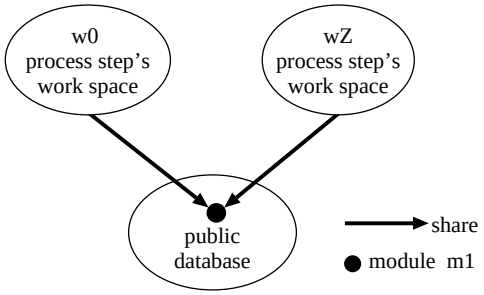


Figure 1: Software product model versus software process model.

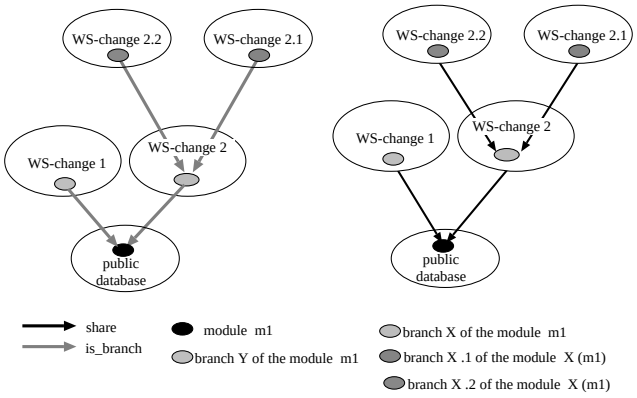
3.1.1 Roles vs. work spaces: coupling modes

As software process step is always linked to a work space, in which the software objects manipulated during the process step life are available, we have defined two kinds of relationship between a software process step and the objects manipulated by it: (1) shared, (2) duplicate and (3) new.

The *shared* key word is used to classify those software objects manipulated by a software process that are not *modified* in isolation. That is, a *shared* object is available in different process steps at the same time. instance, the `to_valid` role is specified as *shared* in the `WS-valid` process step. Thus if a module, e.g. `m1`, plays this role in a `WS-valid` process step occurrence and has either its contents or characteristic modified, these modifications are automatically propagated to all other software process steps that use `m1` in shared mode.



The *duplicate* key word is used to identify those software objects manipulated in *isolation* during the life of the software process step. When the software objects are allocated as a duplicate role to a software process step, a branch is created automatically and this new object is placed inside the work space, in which the process step is carried out. Thanks to this mechanism, all modifications (either content or characteristic) are limited to the process occurrence's work space, and only when the user responsible for the process step decides to propagate these modifications (`promote`), will the other software process steps be **alerted**.



The *new* key work is used to identify those objects that will be created during the process execution. Only when those objects are made persistent in the public database, other process occurrences could reference them.

3.2 Structuring software processes

In the our formalism, a complex software processes step can also be broken down in other sub-process until the desired level of detail is achieved, thus a complex activity can be broken down into a hierarchy of other less complex activities. However, no special semantic is provided to express this policy. The role facility, make it possible to define a sub-process like a standard role. As a software process step roles can view all characteristics of its incorporated roles, in a software process step can be defined that will be played by other software processes. For example, let us imagine a `manager` process, which is responsible for controlling various software engineers working in parallel, where each one carried his software change activity in his respective work space. This management

activity can be described as a software process step, `manager`, composed of other sub-process, `WS-change`, as shown in the figure 2.

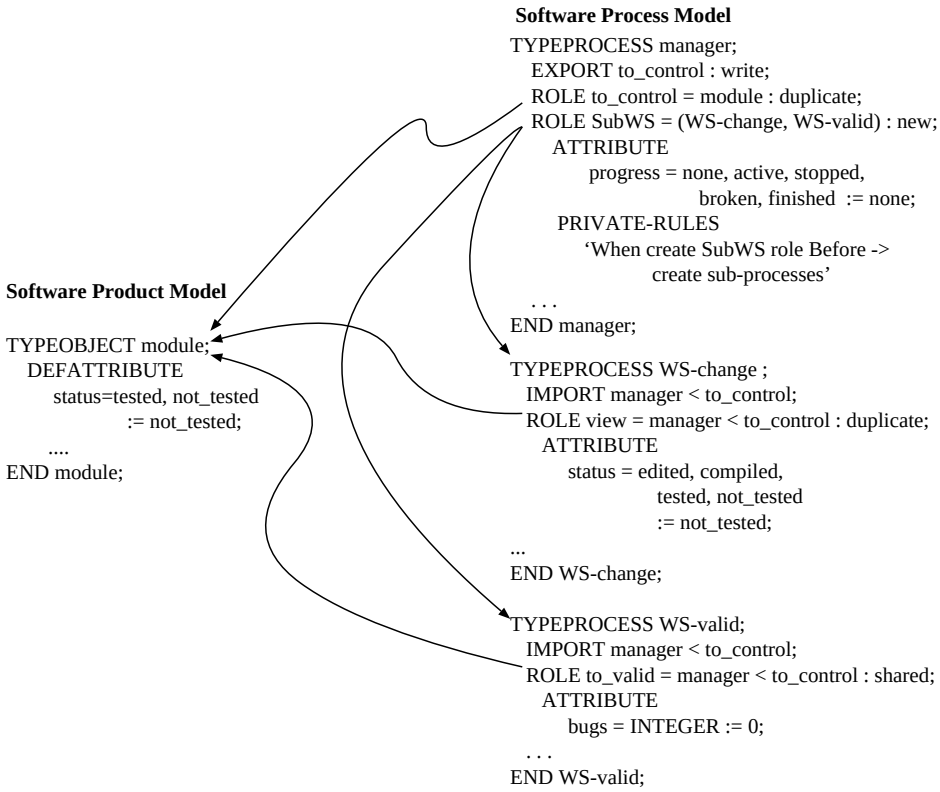


Figure 2: Structuring sub-processes.

3.2.1 Creating sub-processes

In the last section, we have described an example of software process structuring. In this section, we outline how the example is enacted.

Figure 3 presents four snapshots representing some procedures followed when creating occurrences of the software process steps. The oval rectangle represents the Adele database. This database is divided in two parts for didactic reasons, one part represents the software process world and the second one the software product world. The first snapshot presents the initial database state of our example. This initial state is changed by the software process manager (a user who plays the role of the `PMmanager`), who creates the `w0` occurrence (snapshot 2) of the `manager` process step.

As pointed out in the example description, the `manager` process step monitors two `WS-change` process steps, in which the change activities are carried out. The creation of these two sub-process is accomplished by binding the `SubWS` *roles* (see commands between snapshots one and two).

The execution of those methods triggers the privates rules described in `manager` process type, which are responsible for creating the `w1` and `w2` occurrences of the `WS-change` type. In this way, sub-process structuring is achieved by the composition of two (or more) activities: creation of process occurrences followed by role binding. Using this technique, we do not need a special semantic for the sub-process, since the sub-process semantic can be describe by standard roles and ECA rules.

Snapshot four shows the database state after the creation of all processes and sub-process and the binding of all roles. From the software process part, we can see `w0` process occurrence with its three roles: `w1` and `w2` sub-process playing `implement` roles, and `m1` module playing the `domain` role. Each sub-process has a `view` role which associates the process occurrence with the target module of the change activity.

From the software product part, we can see two new software objects that have been instantiated automatically by the role binding operation. As we see later, each process occurrence is carried out in different work spaces, wherein software objects and tools are placed in order to achieve the software process step goal. In our example, each process step occurrence is responsible for changing and testing module `m1`. In order to avoid update conflicts, each software process occurrence will have a branch of module `m1` upon which it will able to modify in "almost" total isolation. In our example, the `check-in` operation is responsible for merging the modification made in the module branch, resident in the private process space, with module `m1`, which is resident in the public database.

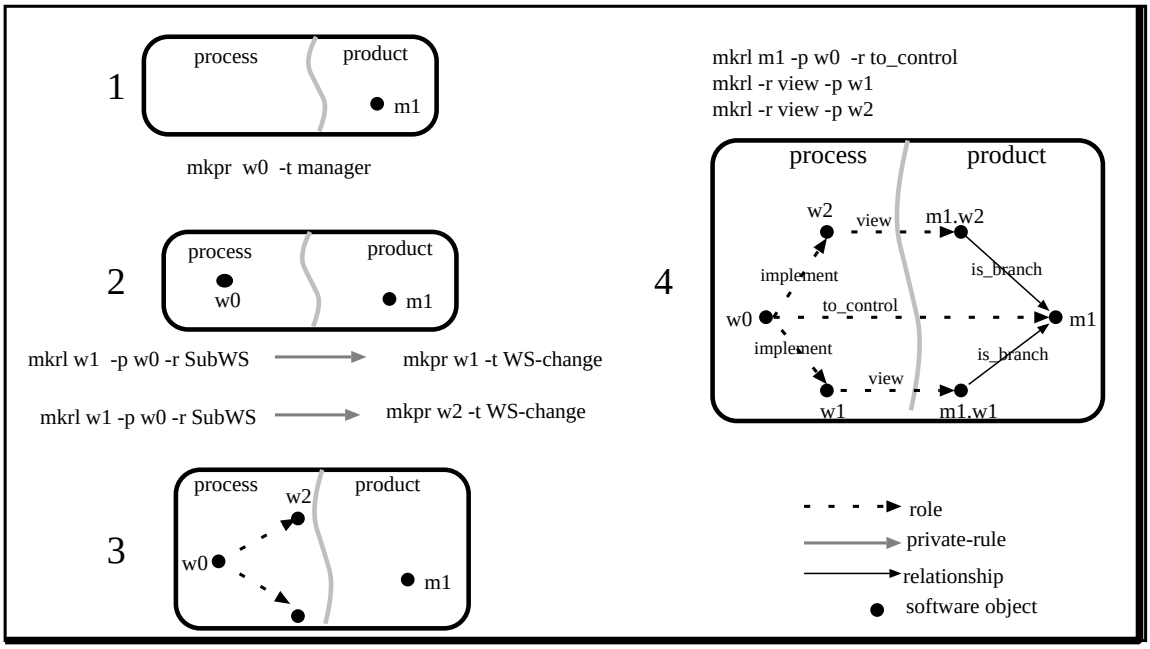


Figure 3: An example of process creation and role binding.

3.3 Modeling contextual behavior: adjusting methods

Each role has methods which are used to adjust the behavior of the original object type to a software process execution context. In other words, a role can redefine the original methods or define new ones in order to customize object behavior for the context where the object is used. For example, the module type has methods associated with it, which are independent of the context where the module instances are used. However, when a module instance is manipulated by a software process step, other methods may be needed, e.g., the method `compile` associated with the module type may be different from the one used by the implementation process step (compilation flags, etc). Figure 4 shows how the behavior of the module instances manipulated by a software process step can be tuned in order to satisfy the requirements of this software process step. The figure shows

the `role view` with two methods, `compile` and `crossref`. The method `compile` overloads the original one defined in the software product model, and the method `crossref` is added to that role. In this way, thanks to the **role** concept, we describe the contextual behavior of the objects.

Some may claim that this kind of contextual behavior can be achieved by standard Object-Oriented techniques. If a single schema is used, it would need to sub-type all the possible combinations of roles for a single type (combination numbers may be very large!), and to change instance type dynamically each time a new role is applied to it. Moreover, since the instance may be shared and play different roles simultaneously, dynamic sub-typing cannot be used.

If multiple schemas are available, each WE may use a different schema, overloading type definition with the corresponding WE role definition. However, since different roles may use the same object type in the same WE, this approach cannot be used either; in addition it would also not solve the local attribute feature.

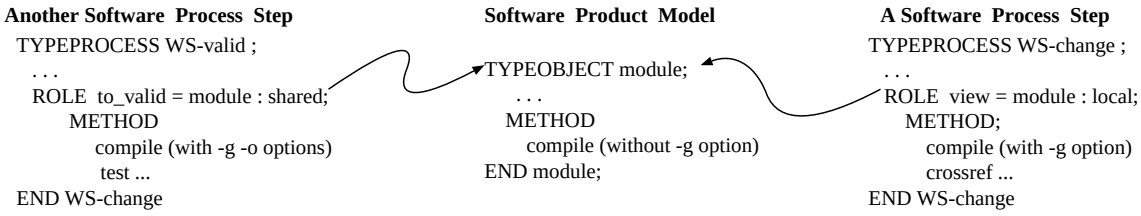


Figure 4: Contextual behavior adjust.

3.4 Private and react rules

Triggers, when used in the software product model, are also very useful for capturing integrity constraints. We have felt that triggers, when attached to a role, are also an important feature, for controlling the operations performed in the software processes. Thus, triggers, defined in the software product model, describe invariant constraints, and triggers defined in the process type (in the roles) define the policies to apply in a process of that type.

Two kinds of triggers can be defined in a role: private and react rules. Private rules control the work performed inside a process; they are executed only in response to actions performed in the process itself. React rules are executed in response to actions undertaken outside the current process. They are used to coordinate activities; for example the propagated effect of a modification on a shared object, etc.

A react rule on a local object is activated when an event occurs to the shared object from which the local object is derived. It makes it possible to work in isolation (the object is local) and to be aware of what happens to the same object in shared mode. The distinction between local and react rule improves method execution control and provides an implicit mechanism for process synchronization. For example, imagine the following software configuration management policy:

> *Two software engineers A and B work in two parallel development WS to implement a specific functionality in module M1. Suppose that A changes the M1 interface. Being a in local role, these activities do not impact B.*
>
> *When A promotes the new interface, its private rules automatically trigger a compilation; if it succeeds the module receives the* `ready` *state, if not the* `promote` *command*

is undone (ABORT). React rules in B WS will notify the change to B, and execute `resynch` *command which automatically copies the changed component to B's WS, and merges if needed (it is the NSE policy[4]).*

```
TYPEPROCESS WS-change ;
  ROLE USER = SoftwareEngineer;
  ROLE to_change = module : local;
    ATTRIBUTE state = edited, compiled, tested;
    METHOD resynch ; { check_out %name; ..} ;
    PRIVATE
      PRE promote : If NOT "compile self" THEN ABORT ;
      POST promote : "self.status := tested";
    REACT
      POST promote :
                "mail !user << warning object %name is obsolete " ;
                "resynch %name" ;
END WS-change ;

TYPEPROCESS manager ;
  ROLE USER = PMmanager;
  ROLE SubWS = development : new ;
  ROLE to_control = module : duplicate ;
  REACT
    POST promote :
                IF FOREACH implement ( %name.state = tested) THEN {
                "self.status := tested" ;
                "promote %name" ;}
END manager;
```

A father process may have access to the local variable of its sub-processes, i.e. the process created as part of its activity. Being the father process of A and B WE, the `manager` process may consult the state of the local variables of A and B processes. The `promote` command performed by A makes the module visible to the `manager`, but not to others, since the manager declared the module as local. However the manager awaits the completion of all its sub-processes (the state of the components is ready in all sub-processes), before promoting the components, i.e. making them visible to others.

Thus, depending on the policy described, the changes made inside a process may or may not interfere with other activities carried out in parallel during the software process.

4 Supporting TEMPO model in Adele

This section describes the implementation of the TEMPO model using the Adele kernel.

We have used the Adele kernel how the virtual machine for interpreting and enacting of our software process model. The Adele kernel is based is an entity relationship database, extended with Object-Oriented facilities and an Activity Manager based on triggers[1, 2].

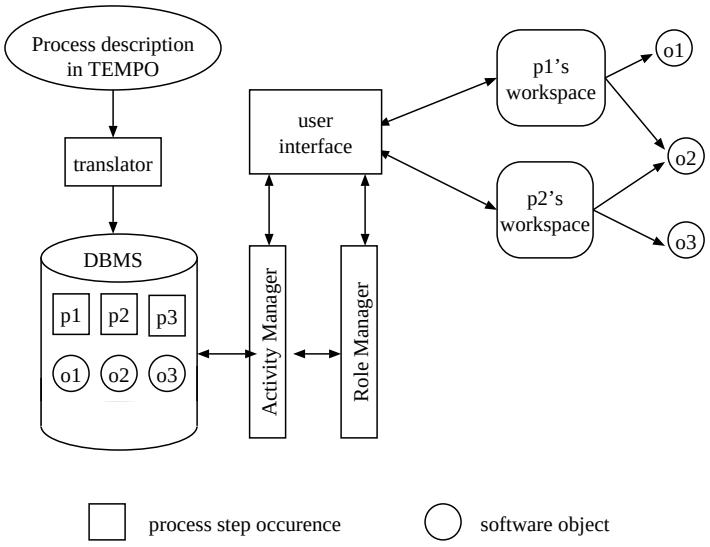


Figure 5: Overview of the TEMPO environment.

4.1 Adele data model

The static aspects of TEMPO are modeled by an "object-relation" data model, which is derived from the Entity-Relationship model extended with composite and versioned objects, multiple inheritance and active entities and relationships (entities and relationships can have methods associated with them). This means that users can create new entity or relationship types, to relate these types in a direct acyclic graph, to define new functions on these types or inherit them from other types in the graph, and to encapsulate those functions with the type.

Since a software engineering environment like TEMPO, versioning is a fundamental feature, in Adele, "revision" is a kernel feature, while high level versioning (revision tree, variants) is left to applications. Each object may have a version branch (i.e. a sequential list of revision); the branch as well as each single revision are first class objects; all characteristics (attributes, relationships, triggers, rights list, etc.) of a version branch are shared by all its individual revisions.

4.1.1 Adele triggers

In order to describe the dynamic aspects, an ECA formalism is provided. With these two formalisms (data model and ECA) we can program TEMPO on the top of the Adele kernel.

This formalism involves two basic concepts: events-condition and actions. *Events* are used to control activities (navigation as well as modification) in the database. An *action* is a set of operations activated by a trigger when an event occurs. Actions can abort transactions, or perform further modifications to the database, which may in turn fire triggers.

An event is a first order logic expression where variables are related to the database state (machine, current transactions, current user, local state) and object or relation attributes. Events are checked each time a method is called.

A trigger program is executed each time the corresponding event is true. Four classes of trigger have been defined: **pre-triggers** executed before the method execution, **post-triggers** executed after

the method. The set of pre-triggers, the methods and the list of post-trigger execution compose a transaction in the database sense. **After-triggers** and **error-triggers** are executed after the transaction is committed or aborted respectively.

Triggers and methods are inherited along the inheritance graph. Triggers are inherited (they cannot be overloaded), and are executed from the most specific to the most general while methods are overloaded.

4.2 Translating TEMPO to Adele

The implementation of the TEMPO model involves the following mapping for translating an Object oriented description of TEMPO model to an entity-relationship model extended with o.o features and triggers.

- processes are represented as object types
- roles are implemented by active relationships
- rules are implemented by triggers associated to relationships and entities.
- methods and ECA rule's action part are translated to Adele language.

5 Related works

Among the kinds of software process programming language that the software process community has used for process-oriented software engineering environments [8] the following can be mentioned:

- the rule-based modeling software process using precondition, activity and post-conditions, e.g. Marvel [7], Epos [3];
- the procedural [9], models derived from programming languages, e.g. Appl/A [12], Triad [10], Galois [11]; and
- the behavioral approach centered on artifacts produced (activities productions) rather than the specific procedures to produce these artifacts [14], e.g. HFSP [13].

Although, our approach is a combination of these paradigms, in our case, rules are derived from event-condition-action formalism and enacted by triggers. We describe software process as an aggregate of objects roles (artifacts) and associate to each role constraint as pre and post-conditions to control the consistency of object roles. From procedural approach, rule description could involve procedural functions and procedures. The basis of integration of these mechanisms is an object manager supporting inheritance, aggregation, late binding and identification of objects

6 Conclusion

The TEMPO model is designed for describing software processes in particular to implement process coordination and ressource sharing. The main capabilities of TEMPO are :

- TEMPO is an Object Oriented model. In TEMPO, the ressources (ie. roles) involved in a process are modelled as objects providing a set of operations and constraints. Software processes and software enginers are allowed to access and manipulate the ressources by using these operations and verifying these constraints. The allocation of ressources is supervised by a central workspace dedied to version and configuration management.
- TEMPO provides a new object-customization mechanism called role . In the TEMPO model an object can be involved in different situations with different behaviours. This approach is closed to delegation mechanism of O.O. languages.
- TEMPO is implemented on top of Adele system which is the abstract process machine. Software process model described in TEMPO are translated in Adele concepts (typed objects and relationships, events rules). We make a considerable use of the activity manager of Adele to support the process enaction. On this way, methods can be associated with pre and postconditions to determine the order of execution of methods. Methods are executed only when the conditions are met.

We believe that the unification of these features led to improve cooperative work among a team of developers and a properly sharing of ressources among a set of processes.

References

[1] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In Dowson [5], pages 159–170.

[2] N. Belkhatir, W. L. Melo, J. Estublier, and A.-M. Nacer. Supporting software maintenance evolution processes in the Adele system. In C. M Pancake and D. S. Reeves, editors, *Proc. of the 30th Annual ACM Southeast Conf.*, pages 165–172, Raleigh, NC, April 8-10 1992.

[3] R. Conradi, E. Osjord, P.H. Westby, and C. Liu. Initial software process management in Epos. *IEE Software Engineering Journal*, 6(5):275–284, September 1991.

[4] W. Courington. *The Network Software Environment*. Sun Microsystems, Inc, 1989.

[5] M. Dowson, editor. *Proc. of the First Int'l Conf. on the Software Process*, Redondo Beach, CA, October 21–22 1991. IEEE CS Press.

[6] P.H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Carnegie-Mellon University, Software Enginnering Institure, March 1991.

[7] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Preliminary experience with process modeling in the Marvel software development environment kernel. In *Proc. of the 23th Annual Hawaii Int'l Conf. on System Sciences*, pages 131–140, Kona, HI, January 1990.

[8] N. H. Madhavji. The process cycle. *IEE Software Engineering Journal*, 6(5):234–242, September 1991.

[9] L. J. Osterweil. Software processes are software too. In *Proc. of the 9th Int'l Conf. on Software Engineering*, pages 2–13, Monterey, CA, March 30-April 2 1987.

[10] S. Sarkar and V. Venugopal. A language-based approach to building CSCW systems. In *Proc. of the 24th Annual Hawaii Int'l Conf. on System Sciences*, pages 553–567, Kona, HI, 1991. IEEE CS Press, Software Track, v. II.

[11] Y. Sugiyama and E. Horowitz. Building your own software development environment. *IEE Software Engineering Journal*, 6(5):317–331, September 1991.

[12] S. M. Sutton, D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. In R. Taylor, editor, *Proc. of the 4th ACM Soft. Eng. Symposium on Soft. Practical Development Environments*, volume 15 of *ACM SIGSOFT Soft. Eng. Notes*, pages 206–217, Irvine, CA, 1990.

[13] M. Suzuki and T. Katayama. Meta-operations in the process model HFSP for the dynamics and flexibility of software process. In Dowson [5], pages 202–217.

[14] L.C. Williams. Software process modeling: a behavioral approach. In *Proc. of the 10th Int'l Conf. on Software Engineering*, pages 174–186. IEEE CS Press, 1988.