Software Process Model and Work Space Control in the Adele System *

Noureddine Belkhatir Jacky Estublier Walcélio L. Melo

L. G. I. BP 53X, 38041 Grenoble Cedex France
{belkhatir, estublier, wmelo}@imag.imag.fr

Abstract

It is advocated here that the most critical aspects for modeling and control, in a large software engineering environment, are inter/intra team communication and synchronization. We propose a solution based on a two level approach: Adele kernel supports multiple activities on shared objects, providing services like contextual behavior, active relationships, and general process support. The second level is the TEMPO formalism based on the role concept, which defines a software process step as a set of objects playing a role. Each object characteristics and behavior depends on the role it plays in the software process it belongs to, and may be part of different simultaneous software processes. TEMPO clearly separates the description of the process (what it does in a Work Environment), from the description of the interaction and collaboration between the different processes.

1 Introduction

Most work on software process management (PM) has been done by people previously involved in project management. From this point of view, the grain (the atomic task or step) is large, and a break down can conveniently be represented by a task tree, where evolution is "slow". Tasks are independent black boxes interconnected by input and output objects.

Others, like ourselves, started in software configuration management (CM) and discovered that CM largely involves process modeling and especially work environment management. Tasks are overlapping boxes with information flow and shared objects. From this point of view, activities are fine grained and may be broken down into a set of overlapping activities. The main problem is the control of multiple simultaneous overlapping activities in a very rapidly evolving world; and the management of the communication and synchronization between teams and between members involved in the same project.

Due to the fact that objects are potentially shared simultaneously by different software processes (in which they play different roles) the behavior of an object cannot be defined statically; it is context dependent, i.e the object behavior depends on the process in which it is used.

Faced with the problem of multiple behavior definition we have used a two layer approach: (1) a kernel providing a general purpose set of concepts and mechanisms, and (2) an enactable formalism for software process definition and control, oriented towards team coordination and synchronization.

This paper presents our two level approach. The bottom layer is the Adele kernel. It is the released Adele Configuration Management kernel (section 2) as discussed in [7], now extended with contextual behavior and general process support services (section 3).

The second level (section 4) is the TEMPO formalism, implemented on top of Adele kernel, designed to provide the Software Team leaders with a simple language for the definition of Software Processes without cumbersome synchronization and communication protocol.

2 The Adele Kernel

The Adele kernel is based is on an entity relationship database, extended with Object-Oriented facilities and an Activity Manager based on triggers[1]. We focus here on those features useful for our topic: managing overlapping work environments, i.e. software process steps, objects, users and tools whose behavior is highly context dependent.

*Published in the *Proc. of the 2nd Int'l Conf. on the Software Process*, Osterweil (Ed.), 25 – 26 February 1993, Berlin, Germany. IEEE Computer Society Press.

2.1 Data model

In the Adele database, entity types and relationships types are declared independently and may have multiple inheritance. In a software engineering environment, versioning is a fundamental feature. In Adele, "revision" is a kernel feature; each object may have a version branch (i.e. a sequential list of revision). The branch as well as each revisions are first class objects; all characteristics (attributes, relationships, triggers, rights list, etc.) of a version branch are shared by all its individual revisions.

The data model is based on the aggregate (or complex object) concept, any kind of aggregate can be user defined, a version branch being a kernel built-in aggregate.

2.2 Events, triggers and methods

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. An event is declared in the following way:

```
event_id = logic_expression; priority n ;
```

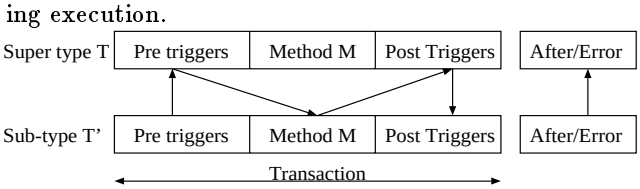
A trigger is declared in a relation type definition or an entity type definition in the following way:

```
ON event_id DO { program }
```

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 set 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.

A `method` is declared in a relation type definition or an entity type definition in the following way `METHOD method_id; signature characteristics; {program}`. The implicit associated event is the method call.

Triggers and methods are inherited along the inheritance graph. Following the same principle found in CLOS, Shood and other OO languages, triggers are inherited (they cannot be overloaded), and are executed from the most specific to the most general while methods are redefined. By default method M on an object or relationship of type T' produces the following execution.



In contrast with most other approaches, e.g. Marvel[9], Epos[5], where {pre, method, post} are declared altogether, in Adele pre- and post- are triggers governed by events, thus this picture can be more complex:

- Pre- and post-triggers are not simple predicates but can be arbitrary programs.
- Pre- and post- are triggered by events, not necessarily a given command.
- Different methods can share some pre- or post-conditions,
- Event have priority, and triggers are executed in the priority order, by default the inheritance order is used, as in the previous example.

Relationships are very similar to entities (they have attributes, triggers and methods); the only difference is that it is not possible to create a relationship between two relationships.

2.3 Contexts

A *context* is a set of object instances defined by an aggregate, i.e. the aggregate head and all its components. A relationship pertains to a context if its origin and destination objects are both in the context.

A user, when "under" a context, sees only the object and relationship pertaining to that context. Different contexts may be active simultaneously for different users.

Since a project contains a large number of object versions, the context concept was introduced to simplify human understanding (getting rid of unneeded objects and versions), to provide a protection mechanism (out of context objects are protected), and to provide a support for work spaces (a common use is to define the workspace instances by a context).

3 Contextual behavior

Adele was primarily a configuration manager system. We are interested in controlling the activities

taking place during the development and maintenance of a software product. In a typical situation, different configurations exist and share a large number of components and documents.

Some configurations are developed in parallel for different clients, to perform functional extensions or technical adaptations. Some configurations are in the design phase while others are in development, test, validation or released. In some cases, a team collaborates on the same configuration and shares all components while working in parallel.

We are faced with the problem of:

- controlling the evolution of shared resources (data consistency),
- supporting a team working simultaneously on the same resources (support different schemas of collaboration / synchronization between teams and members).

Imagine the following situation: an object, `foo.c`, of type `c_file`, is a component of a development context, a validation context, and a component of a configuration. We would like `foo.c` to behave in the following way:

- While used in the development context, the `compile` method uses the `-g` flag (debug compilation option);
- While used in the validation context, a new method, `metrics`, is available;
- As a component of a released configuration, a new method, `archive`, is available; if the configuration is in the `released` state, changes in `foo.c` are forbidden.

Methods can be defined for a type, `c_file`, which can then be applied to each instance of that type. With an O.O. approach all instances of a given type share the same behavior. Consecutively, a type can only define characteristics and behavior that are true for each instance and at all times. Editing, compiling and linking methods can be defined for `c_file`, but the O.O. approach is not flexible enough to model our current problem since all instances must share the same methods. The only solution is to sub-type the `c_file` type, for all possible uses of `c_file` instances. Because of the combination of possibilities this is unwieldy; dynamic type changing would be required, imposing a type definition change, each time a single `c_file` instance is used in a different way.

Therefore we have used active relationships to extend the Adele data model and included context-related behavior.

3.1 Active relationship extensions

In ER models relationships have attributes; in O.O. models relationships are not objects and no attribute can be associated with them. In Adele, relationships are first class citizens and may have attributes as well as triggers and method, to execute when the relationship itself "receives a message".

Adele relationships are binary, always defining an Origin object (called `!O`) and a Destination object (`!D`). Adele extends further the relation concept, allowing a relationship to change the type definition of its origin and/or destination object. It is possible to define in a relationship type, triggers and methods that will dynamically extend and redefine those defined in object type when origin or destination of that relationship.

For example, the composition relation can be defined in the following manner:

```
TYPERELATION composition;
1 ON ORIGIN delete DO {remove !D} ;
2 ON DEST delete DO
     {print "delete first its container !O";
      ABORT};
3 ORIGIN METHOD duplicate -d %new ;
     {copy !O -d %new };
4 ON ORIGIN duplicate DO
     {makerel %new -r %realtype -d !D} ;
```

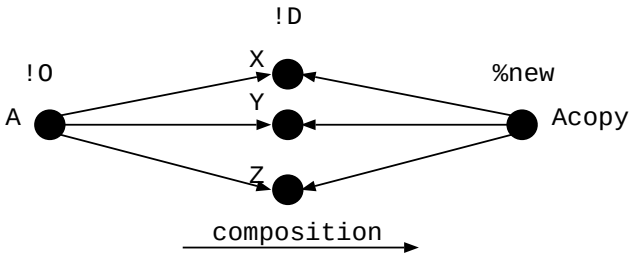


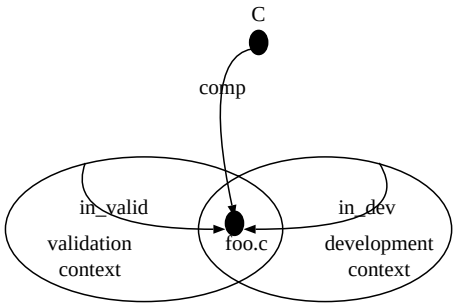
Figure 1: Containment relationships.

Line 1 stipulates that when deleting the origin of an aggregate (A) , the destinations (X, Y, Z) must be deleted too (remove is an internal command). Line 2 stipulates that destinations cannot be deleted individually, since any attempt should produce an abort of the delete command. In the line 3 we added a new method, `duplicate`, defined on the origin (A). Thus,

when "`duplicate A -d Acopy`" sentence is called, the method `duplicate`, which is defined in the relation `composition`, will be executed. If `duplicate` is also defined in the `A` object type, it is dynamically redefined by the one defined in the `composition` relation; if another relationship defines the same method on the same object instance, it is an error.

The `duplicate` method only duplicates the aggregate head and its compositions relationships (line 4). Line 4 instruction is executed for each instance of the `composition` relation type. That composition relation defines an aggregate with logic duplication (sharing the content). Any other aggregate semantics can be defined easily in the same way.

Using such a mechanism, we can solve our problem:




```
TYPERELATION in_valid ;
   DEST METHOD metrics ;
        { "logicscope !D ....."; ..} ;

TYPERELATION in_dev ;
   DEST METHOD cc ;
        { "cc -g -c !D" } ;
TYPERELATION comp ;
   ORIGIN METHOD archive ;
        { ... } ;
```

A relationship is visible only if both the origin and destination objects are visible; therefore, in a validation context, the `metrics` method is available but not the `cc` overload nor the `archive` method. The `archive` method will be available only in a context containing both the configuration and its components.

3.2 Local and global trigger

Let the `comp` relation, relating a configuration to its components, be defined as follows:

```
change_rel_conf = ( type = conf AND
    !modified = true AND state = released);

TYPERELATION comp ;
 PRE DEST change_rel_conf DO
    {print "cannot modify a released conf";
     ABORT}
 ORIGIN METHOD archive ; { ... } ;
```

But since relationship `comp` is not visible in our contexts, the trigger will not be executed. A trigger can be either **local**, i.e. executed only if the relationship is visible, or **global**, i.e. executed whether or not the relationship is visible.

In this situation we must state:

```
GLOBAL DEST modif_released_conf DO
       {program}
```

By default a trigger is local. A global trigger is executed with administrator rights, since the current user probably has no rights on external objects; for instance

```
GLOBAL DEST ch_conf DO
       { !O%state := obsolete } ;
```

means that when a configuration component is modified, the state of the configuration must become obsolete, whether or not the user who performed the change is aware of the configuration existence and has rights on it. Data consistency is enforced.

3.3 Contextual behavior

Object Orientation, Relationship and Context concepts, used in conjunction, allow a separation of concerns:

- O.O. defines the structural and behavioral properties of objects of the same class. All instances of a class are identical regarding the class properties.
- Active Relationships define additional properties for the objects they relate; they allow addition, substitution and delegation of properties as well as information flow in both directions.
- Context defines the visible objects and relationships, and in addition, the object characteristics (attributes, methods, triggers, constrains) which are relevant in a given context.

This separation of concerns has a major impact on the way software processes are described. A relationship defines the dynamic semantics between entities or group of entities. In the Adele system, they are

powerful enough to define this semantics completely, often regardless of the type of the related objects.

Since some of the object semantics is defined in the relationship type, the simple fact that a relationship instance is established or removed, changes the corresponding object behavior, without any change at schema level. The object behavior becomes context dependent.

It is interesting to note that the creation and deletion of relationships is often performed for other reasons than software process control. It is the case in our examples where the contexts and the `comp` relationships are created anyway. The process control comes "for free", there is no need for explicit software process instantiation; enaction is automatic and implicit.

3.4 Generic relationship and software process fragments

Relationship type uses multiple inheritance, and the relation origin and destination types need not be defined. Some relationships can be defined only to specify a generic semantic. For instance, relationship can acquire a composition behavior, by inheritance from the `composition` relation type `TYPERELATION my_aggregate IS composition,..` . It is easy to define a library of "standard" semantics, for instance PCTE link categories [4] (composition, reference, implicit, stabilize), a set of aggregate semantics and a set of the usual work environment inter-relationships semantics.

For work environment (WE) inter-relationship we have defined the relations "busy-propagate" between two copies of an object, with the following semantics "when a copy is modified, the change is propagated immediately to the other copy"; "conditional-propagate", "notify", "resynch"; we defined also "sub-we" relationships for nested WE management.

We have already shown that many WE management policies used by Configuration Management tools[8, 17] can easily be modeled in the Adele system using a few generic relationships [7]. NSE, Palas, Andromede and Aide de Camp have been implemented in a few pages of Adele language. From this experiment we noticed that few different basic semantics are used, but that their combination produces a different software process description.

The current work is to validate this library, by re-implementing a large set of known processes and some more difficult ones.

3.5 Evaluation

This system has been in practical use for several years (section 1), its extensions (section 2 and 3) for one year. It has already proved its power and flexibility. However some weaknesses have been found:

1. Concept level. The language manages concepts such as objects and relationship attributes. There is no high level concept such as software process steps, work environment or synchronization.
2. Fragmentation. The description of a process is often split in different types of object and relations, between methods and triggers. A global view of a process is not easy.
3. Complexity. The large number of possibilities: pre-, post-, after, local, and global triggers; multiple inheritance and relation overload, provide a flexible system. Object behavior can be defined precisely, but it may confuse users. A clear picture of what will happen during execution is not easy.

To overcome these drawbacks, we designed a language, TEMPO [3], on top of the Adele system[1]. TEMPO, defines a process model based the role and connection concepts. A **role** [2] allows to redefines the static and behavioral properties of objects when playing that role in a process; while a **connection** expresses how processes collaborate in a wider context: the complete SEE. This language is presented in the following section.

4 The TEMPO language

A software process step is modeled in TEMPO by a user defined object type, a software process step instance by an object instance in the Adele-DB. Using the standard multiple inheritance mechanisms, a process type can be refined and specialized. Therefore, to some extend, software process customization is achieved by process type specialization.

We use the role concept, which is inspired from the Actor language, to describe the software process resources. A role customizes an object type for a software process step. A role describes an object's contextual behavior, i.e. the description of the operations that can be done on the object and the rules that control these operations. A role adds temporary properties (local attributes) to the object playing this role. A software process step becomes a list of roles which

customize the objects involved in order to satisfy that software process step requirements. The properties and behavior of an object are specific of each software process in which it plays a role.

4.1 Process and Role definition

A role is the set of object instances, having the same behavior and characteristics for a given process. A role defines the common behavior and characteristics of its instances. Characteristics means valid attributes, while behavior means methods and constraints. There is no strict relationship between role and type: (1) an object instance plays a single role in a given process, (2) object instances of the same type may play different roles, (3) instances of different types may play the same role, provided their types are compatible.

A role is defined by a name, a type, local attributes, methods and rules.

```
ROLE role_id = {type | role/expression} ;
        ATTRIBUTE attribute_definition
        METHOD    method_definition
        rule_definition
END role_id ;
```

The following example shows how the module type is customized inside a **development** WE by the **to_consult** and **to_change** roles, and in a **validation** WE by the **component** role.

```
TYPEOBJECT Module ;
  ATTRIBUTE
     state = tested, untested, available ;
  METHOD
     compile ...;          -- with -C option
END Module;

TYPEPROCESS development ;
  ROLE testing = unitary_testing ;
  ROLE to_consult = module ;
  ROLE to_change =
       to_consult/(responsible=!username);
    ATTRIBUTE
      state = compiled,edited, ready;
    METHOD
      compile ... ;       -- with -g option
    AFTER ON compile DO test ...;
END development;

TYPEPROCESS validation ;
  ROLE component = module ;
    ATTRIBUTE
      test_suite = test1, test2;
  ...
END validation;
```

Role **to_change** will be bound to those modules the current user is responsible for (current user name (!username) equals to attribute "responsible"); whereas the role **to_consult** is bound to the other modules. That is, when an occurrence of the **development** process is created, all module instances will be first bound to role **to_consult** then those modules with attribute "responsible" will be moved to the **to_change** role.

We define a **Work Environment** (WE) as a process occurrence i.e. the set of object instances the process will perform on, along with the process descriptions, tools, users, etc. A WE acts as a (very) long transaction. By default any change performed in a WE is visible only in the WE (isolation property of transactions). In a WE is defined only what the process does on its instances. This property makes it easier to define a WE. In our example, the **state** attribute is extended and may now contain two additional values **compiled** and **tested**. A modification of the **state** attribute is local to the WE instance. In a **validation** WE, the **component** role is also bound to modules.

Each role has methods which are used to adapt the behavior of the object to that WE. That is, a role can redefine the original methods or define new ones in order to customize the object behavior for the WE in which it is used. For example, the **module** type has methods independent of the WE where the module instances are used. However, when a module instance is used in a given WE, other methods may be needed, e.g., the method **compile** may be different in a **development** WE than in a **validation** WE.

Triggers, when used in the software product model, are also very useful to capture integrity constraints. We have felt that triggers, when attached to a role, are also an important feature, for controlling the operations performed in the WEs. Thus, triggers, defined in the software product model, describe invariant constraints, and triggers defined in roles define the policies to apply when an object plays that role in a WE. Role triggers control the work performed inside a WE; they are executed only in response to actions performed in the WE itself.

4.2 Process and Role connection

Each process defines what succeeds in a WE as if it were performing alone; which is clearly false. Our basic hypothesis is that numerous activities are carried

out in parallel. Some of such activities collaborate to the same goal (for instance a new release of a software product), some do not collaborate to the same goal but share objects which is, to some extend, collaborating to the evolution of these objects. In all these cases, the relationship between WE must be explicitly defined.

We do not support the current approaches where a software process is described only as a tree of embedded sub-processes; we claim that SEE must be seen as a **federation of collaborating WE**, each WE being an enacted process occurrence. It is our belief that the conceptual definition of the network of collaborating WE is the major weakness in current SEEs.

We assume that the role is the grain for collaboration. The role collaboration is defined by a relationship that express the semantics of the collaboration. We provide a library of usual such semantic relationship: `notify`, (which sends a notification to the WE owner when an event `notify_when` succeeds), `resynch` (that re-synchronize two objects when event `resynch_when` succeeds), merge, duplicate, share, deadline, protect, and so on.

To illustrate how role collaboration can be defined, suppose the following scenario:

When a new release of a given software product must be developed, a general process, called `release` *is created. An arbitrary number of* `development` *WE and a single* `validation` *WE can collaborate to that release process. Each development WE can change only some objects and have read access to the other objects of the release.*

The synchronization between development WEs is as follows: When a given module M receives the "ready" state in a `to_change` *role, M copy in each other* `to_change` *role must be merged, and their owner notified. If M is in a* `to_consult` *role, the new M version automatically replaces the previous one, and notification is sent to the WE user.*

A module M receives the `available` *state only if all its copies have the* `ready` *state. When all modules have the* `available` *state, the* `validation` *WE can be created.*

Interesting collaboration succeeds then between the `development` *WEs and the* `validation` *WE, but for space reason this is not developed here.*

The following example shows how this policy is described in TEMPO.

```
TYPEPROCESS release ;
1 EVENT ready = (state := ready) ;
  ROLE USER = PMmanager;
  ROLE implement = development ;
  ROLE valid = validation ;
  ROLE components = module ; {
    ON ready DO {
2     IF implement.to_change.%name.state
                == ready THEN
3       implement.to_change.%name.state
                := available ;
4     IF implement.to_change.state
                == available THEN new valid ;
     } } ;

5 TYPECONNECTION consult_change
                IS notify, resynch ;
6   CONNECT implement WITH implement
7      WHEN to_consult.name = to_change.name ;
8   EVENT notify_when  = ready ;
        resynch_when = ready ;
  END ;
  TYPECONNECTION change_change
                IS notify, merge ;
    CONNECT implement WITH implement
       WHEN to_change.name = to_change.name ;
    EVENT notify_when  = ready ;
        merge_when   = ready ;
  END ;
END release ;
```

Line 1 stipulates that when an object gets to the state ready, the event `ready` is raised. Line 2 sentence `implement.to_change.%name.state` evaluates to the set of values of attribute state of the object that produced the ready event, as found in all `to_change` roles of current process. Operator "==" means set equality. Line 2 means that all copies of the object %name have the ready state. Similarly, line 3 says that all these object copies must take the available state. Line 4 expression `implement.to_change.state` returns all state values of all object in all to_change role. Line 4 means that when all object get the available state, a `valid` role (i.e. a validation WE) must be created.

A connection is a special kind of relationship supposed to be instantiated between pairs of role instances. A connection is intended to define how each pair of connected object is coordinated. It may be a data flow definition, a status consistency checking, notification, deadline control, message passing, object evolution control and so on. It must be emphasized that connections are not symmetric; for instance, a development WE may automatically want to get new versions of objects as produced in a validation WE, probably not the other way!.

Some of these basic behavior is provided in standard, as here `notify`, `resynch` and `merge`. Using standard inheritance mechanism, each connection can reuse these process fragments (line 5), and redefine,

for instance, the event on which some behavior must be executed. Line 8 means that notification must succeed when the object becomes ready.

The CONNECT clause expresses which pair of objects must be connected. Line 6 means that, for a given release process, two `implement` roles are connected by a `consult_change` connection. Only those instances satisfying the expression found after the WHEN clause are automatically connected . In lines 7 instances of `to_consult` in the first `implement` role are connected with instances of the `to_change` in the second `implement` role having the same name in both roles: the shared instances.

Thus, depending on the connections, the activity performed inside a WE may or may not interfere with other activities carried out in parallel during the software process.

4.3 Roles and classes

Roles and classes look similar; it rises the question: can roles be implemented in term of classes and subclasses?; is the concept of role needed at all?

A role, as well as a class, is a set of instances sharing the same definition (static and behavioral). A given object instance can be simultaneously a member of different role (classes). Both roles and classes can be seen as a viewing mechanism since a given object instance has a different description depending on the role (class) from which it is managed.

However the differences are the following: The association between an instance and its class(es) is statically defined at instantiation time, while an instance can be dynamically bound to an arbitrary role at any time. In an O.O. system the class definition is created first, and then the instances of the class; while in TEMPO, usually, the instances are created first, and are dynamically associated, for a while, to a (set of) role.

Since a given object instance can be simultaneously a member of different roles (classes) there is compatibility rules between those roles (classes) allowed to shared objects. For classes it is the *inclusion semantics* constraint which holds between a class and its sub classes (ISA relationship). For roles it is the connection semantics. A wide range of connection semantics exist (essentially data flow, synchronization, coordination), this is why the language does not impose any predefined semantics. Connection between roles, and their semantics, is user defined.

The role concept is a generalization of the class concept, intended for another purpose: the control of the multiple views of objects instances and the coordination of their concurrent use. As a special case, classic O.O. systems can be implemented in term of roles, not the contrary.

5 Related work

Among the kinds of software process programming language that the software process community has used for process-oriented software engineering environments [10] the following can be mentioned:

- the rule-based modeling software process using precondition, activity and post-conditions, e.g. Marvel [9], Epos [5];
- the procedural [11], models derived from programming languages, e.g. Appl/A [15], Triad [13], Galois [14]; and
- the behavioral approach centered on artifacts produced (activities productions) rather than the specific procedures to produce these artifacts [18], e.g. HFSP [16], Interact [12].

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

We believe that the definitions of a two levels system led to an innovative system for the programming of configuration management policies. We have extended Adele-DB, in order to closely link the static aspect of configuration management (persistent software objects management, versioning, etc) and the dynamic aspect which is software process and work environment control. One of the goals of that work is to provide the process administrator with a simple language for the definition of software procedures without cumbersome communication protocol exchanges. We used a synchronization and communication mechanism based on propagation and notification, which we believe is easy to use and understand.

The contribution of this paper is the description of our two levels:

- A set of basic mechanisms for the general problem of maintaining the consistency of objects used simultaneously, for different purpose, in different and distributed working environments. It is the description of our "abstract process machine" based on an object oriented DB, extended by a trigger mechanism.
- A higher level language, called TEMPO, based on this abstract machine. This language clearly separates products and activities, inter and intra work environment communication. The data model use O.O. concepts for the structuring of the static and persistent object. TEMPO also uses O.O. concepts for structuring software processes with roles as a basic concept at the execution (activity) level.

The result is an integration of software configuration management and software process management which we believe to be of some novelty and significance. The basic layer is a stable prototype, used for 1 years, to be commercially released in 93, the top layer, TEMPO, is under implementation.

Acknowledgement

W. Melo is supported by the Technological and Scientific Development National Council of Brazil (CNPq) under grant No. 204404/89-4.

The authors would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In Dowson [6], pages 159–170.

[2] N. Belkhatir and W. L. Melo. The object role software process model. In J.-C. Derniame, editor, *Proc. of the 2nd European Workshop on Software Process Technology*, volume 635 of *LNCS*, pages 150–152, Trondheim, Norway, 7–8 September 1992. Springer-Verlag.

[3] N. Belkhatir and W. L. Melo. TEMPO: a software process model based on object context behavior. In *Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*, Toulouse, France, December 7–11 1992.

[4] G. Boudier, R. Minot, and I. M. Thomas. An overview of PCTE and PCTE+. In *Proc. of the 3rd ACM Symposium on Software Development Environments*, Boston, Massachusetts, November 28–30 1988. In *ACM SIGPLAN Notices*, 24(2):248–257, February 1989.

[5] 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.

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

[7] Jacky Estublier. The adele configuration manager. Adele Technical Report, 1992.

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

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

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

[11] L. J. Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, Monterey, CA, March 30-April 2 1987.

[12] D. E. Perry. Policy-directed coordination and cooperation. In I. Thomas, editor, *Proc. of the 7th Int'l Software Process Workshop*, San Francisco, CA, October 16–18 1991. IEEE Computer Society Press.

[13] S. Sarkar and V. Venugopal. A language-based approach to building CSCW systems. In *24th Annual Hawaii International Conference on System Sciences*, pages 553–567, Kona, HI, 1991. IEEE Computer Society, Software Track, v. II.

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

[15] S. M. Sutton, D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. In *Proc. of the 4th ACM Symposium on Software Development Environments*, Irvine, CA, December 3–5 1990. In *ACM Software Engineering Notes*, 15(6):206–217, December 1990.

[16] M. Suzuki and T. Katayama. Meta-operations in the process model HFSP for the dynamics and flexibility of software process. In Dowson [6], pages 202–217.

[17] K. C. Wallnay. Issues and techniques of CASE integration with configuration management. Technical Report CMU/SEI-92-TR-5, Carnegie-Mellon University, Software Enginnering Institure, March 1992.

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