A Software Engineering Environment driven by Event-Condition-Action Rules and its Trigger Mechanism [*]

Walcélio L. Melo
University of Brasilia
Dep. of Computer Science
Brasilia, DF, Brazil
e-mail: melo@cic.unb.br

Noureddine Belkhatir and Jacky Estublier
Laboratoire de Genie Informatique
BP 53
38041 Grenoble France
e-mail: belkhatir@isis.imag.fr

Abstract

Recently, PSEE's (Process-Centered Software Engineering Environments) have been investigated as a new architecture of SEE's in which the software processes are explicitly described and drive the user interactions. A typical PSEE is composed of two components: a resource manager and a process manager. The resource manager is responsible for the management and control of all objects manipulated during the software processes. The process manager is the component supporting an explicit formalism to describe software processes. ADELE/TEMPO is a sample of this new tendency. This paper presents the main components of the kernel of the ADELE/TEMPO system, i.e., its resource manager and process manager. Special attention is given to how these different basic components are integrated into a platform where software process models can be explicitly described by event-condition-action rules and supported by an active software engineering database.

key-words: trigger, active software engineering database, role concept, process modeling, event-condition-action rules, knowledge-based software engineering environments.

[*]This paper is a revised and extended version of "Software process model and work space control in the ADELE/TEMPO system", N. Belkhatir, J. Estublier and W. Melo, published in the *Proc. of the 2nd Int'l Conf. on the Software Process*, pp. 2–11, Feb. 1993, IEEE Press.

1 Introduction

During the last decade, projects on SEE (Software Engineering Environments) pointed out the importance of data integration and the role of an object management system as a kernel of SEE's, e.g. ADELE [Belkhatir and Melo, 1994] and PCTE+ [Boudier et al., 1988].

Recently PSEE's (Process-Centered Software Engineering Environments) have been investigated as a new architecture of SEE's in which the software processes are explicitly described and drive the user interactions. A typical PSEE is composed of two components: a resource manager and a process manager. The resource manager is responsible for the management and control of all objects manipulated during the software processes. The process manager is the component supporting an explicit formalism to describe software processes. In general, the resource manager uses an object-oriented database for controlling the allocations of resources to the software processes.

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 should not be defined statically; it is context dependent, i.e the object behavior depends on the processes in which it is used.

Faced with the problem of multiple behavior definitions, 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.

Based on our industrial experiments with ADELE 2 [Belkhatir and Melo, 1994] (a software engineering database), we developed a prototype of a PSEE, called ADELE/TEMPO. TEMPO makes intensive use of OO (Object-Oriented) concepts and adapts them to software process description and enactment. The main characteristics of the software process language used by the ADELE/TEMPO system are the heavy use of trigger rules and OO concepts, such as inheritance, late-binding and methods.

In the remainder of this paper, we will show that the rule trigger is an efficient paradigm to be used by a process manager and illustrate how an active software engineering database can support the dynamic points of view of software objects. We shall present the formalism used by ADELE/TEMPO for modeling software policies and the enforcement of those policies in the environment by a trigger mechanism integrated with a software engineering database. Section 2 presents the lessons learned from other active software engineering databases. Section 3 discusses the kernel of the ADELE software engineering database, focusing on its active part. Section 5 gives an overview of the TEMPO software process modeling language.

2 Lessons learned from software engineering databases

Software artifacts involved in software processes are complex and highly inter-related. Many projects note the inadequacies of file systems to manage software artifacts and focused early on database technology [Belkhatir and Melo, 1994, Kaiser et al., 1988, Oquendo et al., 1991]. The database technology has been used at two levels of software development: programming-in-the-small (hereafter PITS) and programming-in-the-large (hereafter PITL).

PITS is related to the activity of the individual developer that creates and maintains monolithic programs. PITL deals with activity of development of large systems composed of modules and involving a team of several persons.

Referring to PITS, database technology has been used to make the structure of a program explicit as a monolithic entity. The structure represents fine grained details based on an abstract syntax representation of programs. These environments show mainly efficiency problems related to a low level of granularity of objects as reported by many projects [Linton, 1984, Lamsweerde, 1988].

Referring to PITL, many benefits have resulted from the use of database technology. These have been achieved in the representation of program structuring involving many modules and the management of evolution in versions and configurations. Early approaches combining a file system to store content of objects and a database system to manage their semantic informations as attributes and relationships have been used, e.g. PMDB [Penedo, 1991].

3 The ADELE Kernel

Software engineering databases must manage a large amount of shared data and, as such, require assistance when managing crucial situations. For instance, when a module interface is modified, the impact on modules using this interface must be evaluated and the impacted modules notified and eventually recompiled. Dynamic aspects have been investigated in many software databases as a way to provide this kind of assistance. An active database is useful for implementing management policies in a general and flexible way. The information to manage is essentially a versioned database. In this context, trigger-actions mechanisms seem to be appropriate. Trigger mechanisms allow the definition of actions to be executed automatically when some conditions hold; for instance, checking integrity constraints or propagating changes. ADELE/TEMPO provides a formalism for describing events and triggers and an activity manager that specifies trigger processing and synchronization in database transactions.

The ADELE database is based on an entity relationship database, extended with Object-Oriented facilities and an Activity Manager based on triggers [Belkhatir and Melo, 1994]. We focus here on those features useful for our topic: managing overlapping work environments, i.e. software activities, objects, users and tools whose behavior is highly context dependent.

3.1 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. The branch as well as each revision are first class objects; all characteristics (attributes, relationships, triggers, rights list, etc.) of a version branch are shared by all of 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.

3.2 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 ;
```

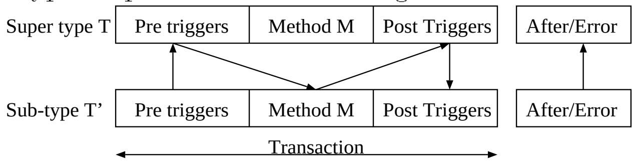
A trigger is declared in a relation type definition or an entity type definition in the following way:

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

A trigger program is executed each time the corresponding event is true. Four classes of triggers 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. Methods can be inherited along the inheritance graph and by consequence, overloaded.

Triggers are also inherited along the inheritance graph. However, they cannot be overloaded. When inherited, triggers are executed from the most specific to the most general. By default, method M on an object or relationship of type T' produces the following execution.



A trigger is executed whenever the associated event becomes true. 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-triggered are activated by events, not necessarily a given command.
- Different methods can share some pre- or post-conditions.
- Events have priorities. Triggers are executed in 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.

4 Managing contextual behavior

In general, large software systems are organized in one or more configurations which can have overlapping software objects. Some configurations are developed in parallel for different clients to perform functional extensions or technical adaptations. Some components of overlapping configurations can be in the design phase while others are in development, test, validation or release. In some cases, a team using the same configuration can share all components because some activities can be done in parallel.

In this framework, we must handle the following problems:

- 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).

To provide a solution to these problems, we have extended the ADELE data model with active relationships. In the next section, we will illustrate how active relationships can support contextual behavior.

4.1 Active relationship extensions

In an Entity-Relationship model, relationships have attributes; in a pure OO model, relationships are implemented by attributes, so we can not define further attributes associated with them. In ADELE, relationships are first class citizens and may have attributes as well as triggers and methods that execute when the relationship itself "receives a message".

ADELE relationships are binary, always linking an Origin object (called `!O`) with a Destination object (`!D`). ADELE extends the relationship concept further by allowing a relationship to change the type definition of its origin and/or destination object. It is possible to define triggers and methods in a relationship type that will dynamically extend and override those defined in the origin or destination object of that relationship. For example, the composition relationship 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} ;
```

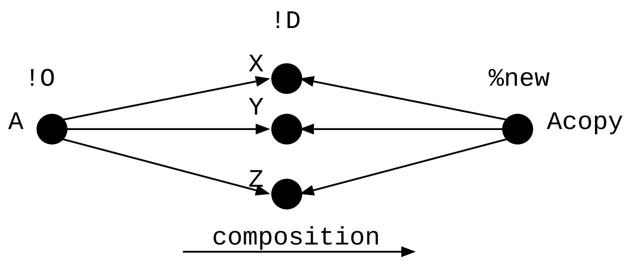
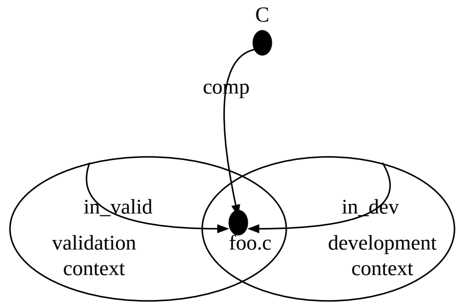


Figure 1: Containment relationships.

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. (This restriction may seem bizarre to some readers. We would like to point out that this is only an example of the utilization of the trigger on relationships for implementing a particular configuration management policy. Using trigger and methods on relationships, we can implement other kinds of policies.) 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). The instruction in the line 4 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 support teams working simultaneously on shared resources as well as evolution of shared resources:



```
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 or the archive method. The archive method will be available only in a context containing both the configuration and its components.

4.2 Contextual behavior

When used in conjunction, OO, Entity-Relationship and Contextual Behavior mechanisms allow a separation of concerns:

- OO defines the structural and behavioral properties of objects of the same class. All instances of a class have identical 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, constraints) 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 groups of entities. Since some of the object semantics are 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 the schema level. The object behavior becomes context dependent.

It is interesting to note that the creation and deletion of relationships are often performed for other reasons other than software process control. This is the case in the above example 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.

4.3 Evaluation

ADELE 2 [Belkhatir and Melo, 1994] has been in practical use for several years. Based on this utilization, we can observe the following weaknesses:

1. Concept level. The language manages concepts such as object and relationship attributes. There are no high level concepts such as long term software activities, user roles, communication strategies, etc.

2. Fragmentation. The description of a process is often split into different types of object and relations or between methods and triggers.

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.

To overcome these drawbacks, we implemented a software process programming language, the TEMPO language [Melo, 1993], and a process engine, the TEMPO system, to interpret this language on top of the ADELE system [Belkhatir and Melo, 1994]. The ADELE/TEMPO system is thus the result of the integration of ADELE 2 [Belkhatir and Melo, 1994] with TEMPO.

ADELE 2 plays the role of resource manager in the current version of ADELE/TEMPO. ADELE 2 [Belkhatir and Melo, 1994] 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 [Estublier et al., 1984] and NOMADE projects [Belkhatir and Estublier, 1987]. ADELE 1 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 [Melo, 1993] is, in fact, the successor of NOMADE. TEMPO is able to deal with user defined software process models, multiple points of view of software artifacts, temporal events, long-time duration activities, and support for communication of software activities.

TEMPO defines a software process formalism based the role and connection concepts. A **role** allows the redefinition of 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. The TEMPO language is presented in the following section.

5 The TEMPO language

A software activity or task is modeled in TEMPO by a user defined object type — a work-environment type. Using the standard multiple inheritance mechanisms, a work-environment type can be refined and specialized. Therefore, to some extent, software process customization is achieved by type specialization.

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

5.1 Process and Role definition

A role is a 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 mean attributes while behavior means methods and constraints. There is no strict relationship between roles and types: (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.

(We employ the term "role" in a different way than people from software process community. In the software process community, the role concept is commonly used to refer to users only, not objects fulfilling a particular software development role (e.g., programmer, manager, testing staff, etc.). At the beginning of our work, we arbitrarily restricted the role concept to the commonly used definition. Based on work from other domains, e.g. database and OO Persistent Programming Languages, we decided to expand the role concept to all objects manipulated during software process execution. It is interesting to note that IPSE 2.5 [Warboys, 1989], one of the first PSEE, also employs the term "role" different from the typical manner, i.e. a role in IPSE 2.5 is a software activity.

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 (Work-Environment) 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;

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

WETYPE 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 the `to_consult` role, then, those modules with attribute "responsible" will be moved to the `to_change` role. This policy is expressed by the following expression:

```
ROLE to_change =

     to_consult/(responsible=!username);
```

We define a `Work Environment` (WE) as a set of software objects that will be manipulated inside the border of the WE, 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 our example, the `state` attribute is extended and may now contain two additional values: `compiled` and `edited`. A modification of the `state` attribute is local to the WE. 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 specific module 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.

5.2 Process and Role connection

When a user is logged in a WE, she/he feels like she/he was performing her/his activities alone; which is clearly false. Our basic hypothesis is that numerous activities are carried out in parallel. Some of the 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 are, to some extent, collaborating to the evolution of these objects. In all of these cases, the relationship between WE's 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, e.g. NSE [Miller, 1989], EPOS [Conradi et al., 1991] and ALF [Canals et al., 1993]; we claim that a 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 SEE's. This idea is shared by some advanced SEE's, for instance Oz [Ben-Shaul and Kaiser, 1994].

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

To illustrate how role collaboration can be defined, consider 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's and a single* `validation` *WE can collaborate on that release process. Each development WE can change only certain objects, but can 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, the copy of M in each of the other `to_change` roles must be merged, and their owners 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.

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

```
WETYPE 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. In Line 2, the expression `implement.to_change.%name.state` evaluates to the set of values of the `state` attribute of the object that produced the ready event. Operator "==" means set equality. In other words, line 2 means that all copies of the object `%name` have the ready state. Similarly, line 3 says that all of these object copies must take the `available` state. The Line 4 expression `implement.to_change.state` returns all `state` values of all objects in all `to_change` roles. The expression in the line 4 means that when all objects attain the `available` state, a `valid` role (i.e. a validation WE) must be created.

A connection is a special kind of relationship that must be instantiated between pairs of role instances. A connection is intended to define how each pair of connected objects 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.

Some of these basic behaviors are provided in the standard library, such as `notify`, `resynch` and `merge`. Using the 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 line 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.

5.3 Role discussion

One may claim that this kind of contextual behavior can be achieved by standard object-oriented techniques. Roles and types look similar. This raises the questions: Can roles be implemented in terms of typing and sub-typing? Is the concept of role needed at all? We claim the role concept has the following advantages:

1. It prevents type explosion.

   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 a member of different roles (classes) simultaneously. Both roles and types 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. One would need to create a sub-type for all of the possible combinations of roles for a single type and 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 be shared and play different roles simultaneously, dynamic typing cannot be used. We introduce the possibility of changing types dynamically. In an OO system the type definition is created first, and then the instances of the type. In TEMPO, on the other hand, the instances are usually created first, and are dynamically associated temporarily to a (set of) role(s).

2. Object identity is not altered.

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

3. It facilitates schema evolution.

   Schema evolution support is an important facility for a software engineering environment because we need to have the capability of evolving 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 OO languages. The role concept offers two kinds of evolution:

   (a) role definition can change generating a role version;

   (b) objects can change their roles dynamically.

6 Related work

6.1 The role concept

Other SEE's have recently integrated concepts similar to the **role** concept proposed in TEMPO (for example, MERLIN [Peuschel et al., 1992], ES-TAME [Oivo and Basili, 1992] and ALF/PCTE with its "*Work Scheme*" concept [Canals et al., 1993]).

MERLIN allows the restriction or expansion of the user's view depending on his rank/role (manager, programmer, engineer, etc.). It also allows the determination of actions that can be applied to the objects which form the user's view. Thus, methods can be masked/hidden depending on the role of the user.

In PCTE [Boudier et al., 1988], the same object can have different properties (attribute and relations) depending on the "Work Scheme" under which it is handled.

In the field of databases, the problem related to the modeling of object roles tends to be used as a means of improving the description of object evolution phases or object utilization facets [Bachman and Daya, 1977]. Several studies have been examining this problem. In general, the strategy used is to adopt a persistent object-oriented language and then extend it with the view concept, like the Aspect languages [Richardson and Schwartz, 1991], Fibonacci [Albano et al., 1993] and Views [Shilling and Sweeney, 1989]. Therefore, object handling is carried out via its view. Compared to our approach, these approaches have the following limitations:

- They offer no concepts for modeling activities where objects can be handled. Therefore, the way in which an object is perceived and handled is described without considering the context in which the object will be used.

- The choice of the use of a view is left to the application's programmers. This decision is based on the description in the program of the view that must respond to the messages sent to the object.
- In general,these languages do not offer concepts for aggregation of different points of view.

6.2 Rule-based systems

Other SEE's use event-condition-action (ECA) rules, such as AP5 [Goldman and Narayanaswamy, 1992], ALF [Canals et al., 1993], Triad [Sarkar and Venugopal, 1991] and APPL/A [Sutton et al., 1990]. ALF and TEMPO provide four ECA rules execution modes (PRE,POST,AFTER and EXCEPTION) whereas AP5, MARVEL [Barghouti and Kaiser, 1992], Triad and APPL/A seem to support only one mode of execution, the mode POST.

ALF's event-condition-action rules are similar to those offered by ADELE/TEMPO. However, ALF does not have the concept of method. As in MARVEL, all of the actions must be defined by the operators (production rules according to the MASP formalism). The execution of an operator can trigger a forward chaining process in the user's private space (ASP in the ALF terminology). ECA rules are defined elsewhere and executed by another mechanism called "trigger". In TEMPO, we adopted only one concept to define both the constraints on the execution of methods and the constraints on the utilization of objects.

6.3 Process modeling

ADELE/TEMPO proposes an approach which combines the three major process modeling approaches: rule-based, procedural, and behavioral. In our case, rules are derived from an event-condition-action formalism and are enacted by triggers. We describe the software process as an aggregation of objects with roles and associate constraints with each role such as pre and post-conditions which can be used to control the consistency of object roles. From the procedural approach, rule descriptions can involve functions and procedures. The basis of integration of these mechanisms is an object manager supporting inheritance, aggregation, late binding and identification of objects.

7 Conclusion

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

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, but for different purposes, in distributed working environments. It is the description of our "abstract process machine" based on a software engineering, object-oriented database, and 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 uses OO concepts for the structuring of the static and persistent objects. TEMPO also uses OO 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 industrial prototype; the top layer, TEMPO, is an academic prototype.

Acknowledgements

We would like to express our thanks to Roseanne Tesoriero, Bill Thomas and to the referees for suggesting substantial and helpful revisions to the original text, to Rubby Cassalas for extending ADELE 2 data model with the inheritance mechanism, to Christophe Gadonna for maintaining ADELE 2, and Jean Chouanard for extending ADELE 2 with a client-sever architecture which facilitated the implementation of the ADELE/TEMPO prototype.

This work, in part supported by the Technological and Scientific Development National Council of Brazil (CNPq) under grant No. 204404/89-4, is heavily based on [Melo, 1993].

References

[1] Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. (1993). An object data model with roles. In Agrawal, R., Baker, S., and Bell, D., editors, *Proc. of the 19th Int'l Conf. on Very Large Data Bases*, pages 39–51, Dublin, Ireland.

[2] Bachman, C. and Daya, M. (1977). The role concept in data models. In *Proc. of the 3rd Int'l Conf. on Very Large Data Bases*, pages 464–467.

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

[4] Belkhatir, N. and Estublier, J. (1987). Experience with a database of programs. *ACM SIGPLAN Notices*, 22(1):84–91.

[5] Belkhatir, N. and Melo, W. L. (1994). Supporting software development processes in Adele 2. *Computer Journal*, 37(7):621–628.

[6] Ben-Shaul, I. and Kaiser, G. (1994). A paradigm for decentralized process modeling and its realization in the Oz environment. In Fadini, B., editor, *Proc. of the 16th Int'l Conf. on Software Engineering*, Sorrento, Italy. IEEE CS Press.

[7] Boudier, G., Minot, R., and Thomas, I. M. (1988). An overview of PCTE and PCTE+. In Henderson, P., editor, *Proc. of the 3rd ACM Software Enginnering Symposium on Practical Software Development Environments*, volume 24 of *ACM SIGPLAN Notices*, pages 107–109, Boston, MA.

[8] Canals, G., Boudjlida, N., Derniame, J.-C., Godart, C., and Lonchamp, J. (1993). A short tour through ALF. In Finkelstein, A., Kramer, J., and Nuseibeh, B., editors, *Software Process Modelling and Technology*. Research Studies Press.

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

[10] Estublier, J., Ghoul, S., and Krakowiak, S. (1984). Premilinary experience with a configuration control system for modular programs. *ACM SIGPLAN Notes*, 9(3):149–156.

[11] Goldman, N. and Narayanaswamy, K. (1992). Software evolution through interative prototyping. In Montgomery, T., editor, *Proc. of the 14th Int'l Conf. on Software Engineering*, Melbourne, Australia. IEEE CS Press.

[12] Kaiser, G. E., barghouti, N. S., Feiller, P. H., and Schwanke, R. W. (1988). Database suport for knowledge-based engineering. *IEEE Expert*, 3(2):18–32.

[13] Lamsweerde, A. V. (1988). Generic lifecycle support in the Alma. *IEEE Transactions on Software Engineering*, 14(6).

[14] Linton, M. A. (1984). Implementing relational views of programs. *ACM SIGPLAN Notices*, 19(5):132–140.

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

[16] Miller, T. (1989). Configuration management with the NSE. In Long, F., editor, *Proc. of Int'l Workshop on Software Engineering Environments*, volume 467 of *LNCS*, pages 99–106. Springer-Verlag, Berlin, 1990, Chinon, France.

[17] Oivo, M. and Basili, V. R. (1992). Representing software engineering models: the TAME goal oriented approach. *IEEE Transactions on Software Engineering*, 18(10):886–898.

[18] Oquendo, F., Boudier, G., Gallo, F., Minot, R., and Thomas, I. (1991). The PCTE+'OMS: A software engineering database system for supporting large-scale software developpement environments. In *Proc. of the 2nd Int'l Symp. on Database Systems for Advanced Applications*, Tokyo, Japan.

[19] Penedo, M. (1991). Acquiring experiences with the modeling and implementation of the project life-cycle process. *IEE Software Engineering Journal*, 6(5):285–302.

[20] Peuschel, B., Schafer, W., and Wolf, S. (1992). A knowledge-based software development environment supporting cooperative work. *Int'l Journal of Software Engineering and Knowledge Engineering*,

2(1):79–1–6.

[21] Richardson, J. and Schwartz, P. (1991). Aspects: Extending objects to support multiple, independent roles. In *Proc. of the Int'l Conf. on Management of Data*, volume 20 of *ACM SIGMOD Record*, pages 298–307.

[22] Sarkar, S. and Venugopal, V. (1991). 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. IEEE CS Press, Software Track, v. II.

[23] Shilling, J. and Sweeney, P. (1989). Three steps to view: Extending the object-oriented paradigms. In *Proc. of the OOPSLA'89*, volume 24, no. 10 of *ACM SIGPLAN Noticies*, pages 353–361.

[24] Sutton, S. M., Heimbigner, D., and Osterweil, L. J. (1990). Language constructs for managing change in process-centered environments. In Taylor, R., 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.

[25] Warboys, B. (1989). The Ipse 2.5 project: process modelling as the basis for a support envrionment. In Madhavji, N., Schafer, W., and Weber, H., editors, *Proc. of the 1st Int'l Conf. on System Development and Factories*, Berlin, Germany. Pitman Publishing, London, March 1990.