Supporting Software Development Processes in Adele 2

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

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

Published in *The Computer Journal*, 37(7):621-628.

Abstract

After years using of Adele [3], a configuration management system, it became apparent that it lacks activity-related concepts and mechanisms like work environment control, user coordination and synchronization, method and tool control, etc. It was also clear that considerable work is required to adpat a Software Engineering Environment (SEE) to user requirements. Using this experience, Adele 2 has been implemented to provide a general support for defining and managing dynamic aspects of a SEE and facilitate the building of new SEE's. This paper describes, using an example (work space control), the concepts and mechanisms involved. We show how close integration of an activity manager with a software engineering database fulfills the basic requirements and how a high level task manager coupled to a configuration manager can be developed.

Key words *CASE; software development process; team coordination; programming-in-the-large; software engineering environment; process model; process control; groupware mechanisms.*

1 Introduction

The development and maintenance of a software product is a very complex task. In general, we have many people working in parallel for a long period. These people handle a large number of interdependent software objects that typically undergo numerous changes during the software life cycle. Thus, the emphasis is on the resolution of the programming-in-the-large problems [9, 25], i.e. how and when we shall enforce the ordering, synchronization and communication of the activities that are concurrently carried out within the environment by different users. Though these policies need to be enforced, we cannot force a project team to adopt the predefined policies provided by a Software Engineering Environment (SEE). It is not reasonable to force a project team to use the coordination policies provided, for instance, by Dsee [17] which enforces the coordination only when the information are extracted from or deposited in the Object Management System (OMS). Although, this kind of policy may be very adequate for small projects, it is clearly not sufficient for very large projects. In order to support programming in the large, a SEE must provide features to specify policies, and mechanisms to interpret and enforce these policies; a SEE needs be driven by an executable software process model to allow automated assistance [13].

Bearing this in mind, we present in the remainder of this paper the Adele system, showing how Adele can be tailored for software process management. In order to achieve this, we have divided this paper in the following way: in section 2, we give an Adele overview. In section 3, we show how (very) long transactions are carried out. Next, in section 4, we describe the Adele Activity Manager, used to describe and enforce both coordination policies and constraint integrity. We show how this component is used to coordinate parallel software development activities using an example application.

2 Adele background

In CASE (**C**omputer **A**ided **S**oftware **E**ngineering) applications the development of a software product is a complex task. Software products need to be structured, components need to be processed, tools adapted, developments traced and users coordinated and synchronized. Current environments try to offer some assistance to support these activities but results are very limited. Usually ad-hoc solutions are implemented and little assistance is offered. We lack formal, unified assistance for product development, which can be formally specified by the administrators or team leaders.

The Adele system has been specifically designed and implemented for supporting CASE applications in a multi-user and multi-version context. Adele is a configurable open framework within which we can build third generation SEEs, i.e. process-centered SEE's. Although the Adele system may be used by all phases of the software life cycle, it is with SEE's specializing in change activities that the Adele system has shown its main capabilities [8]. Nowadays, a project manager can define the static aspects of a SEE (objects, relationships among objects, etc.) as well as the dynamic aspects associated with it using the Adele modeling language [4]. The static aspects are defined by the Adele data model which is based an entity-relationship model extended with objects, version, multiple inheritance, and schema partition/evolution supported by a multi-user/multi-version software engineering database. The behavior aspects of the SEE are described by an event-condition-action supported by a trigger mechanism [5].

2.1 Adele architecture

Adele was, in its previous versions, a configuration manager, and was really used for large software systems. These experiments demonstrated that methodology and management problems are often more crucial than technical problems, and that there is little support for that. This led us to evolve towards product and process support. In this section, we present Adele's architecture explaining how this type of architecture can be used to build specific Software Engineering Environments using an example which show how a user specified work environment (WE) strategy can be specified. As shown in the figure 1, the Adele kernel comprises:

- a software engineering data base — Adele-DB. Adele-DB is an active multi-user versioned program data base. This base may be distributed on different sites connected by a local network and it can be used by application programs via an RPC interface, by a command language via the Unix shell interface or by a graphic interface.

  Adele-DB supports an entity-relationship data model which is extended with object-oriented concepts like inheritance, methods and encapsulation. Simple and composite objects, ranging from elements (associated to a file) to projects, with attributes and relationships can be described and managed. Composite objects are aggregated by relations. For instance, a module is a complex object constituted by separate interfaces and bodies, and other objects (derived objects, etc.).

- a configuration manager — Adele-CM. Traditional software production tools cannot manipulate versioned objects. So, it is necessary to have a tool able to extract from Adele-DB any mono-version configuration. In the Adele system, it is done by the configuration manager [11]. It calculates the configurations according to a set of constraints, over the objects and over the relations, supplied by the user, unlike other approaches that work over a given configuration [29]. To Adele, a configuration is an object comprising of a set of interfaces, and a set of realizations.

- an activity manager — Adele-AM. Adele-AM is driven by temporal-event-condition-action rules (TECA) and supported, in part, by Adele's trigger mechanism [5]. We have enhanced Adele's trigger mechanism with the ability to manipulate temporal expressions [6]. we present in greater detail Adele-AM in section 4

Adele 2 [4] is a commercial product which is the result of the union of two long term projects in the framework of the *Laboratoire de Génie Informatique de Grenoble*. Adele 2 integrates the results produced by the Adele 1 and Nomade projects [2]. Adele 1 [2] was a version management system hard-coded with a configuration builder quite similar to the one of Rcs [30]. 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. Adele 2 is in fact the commerical version of the Nomade system.

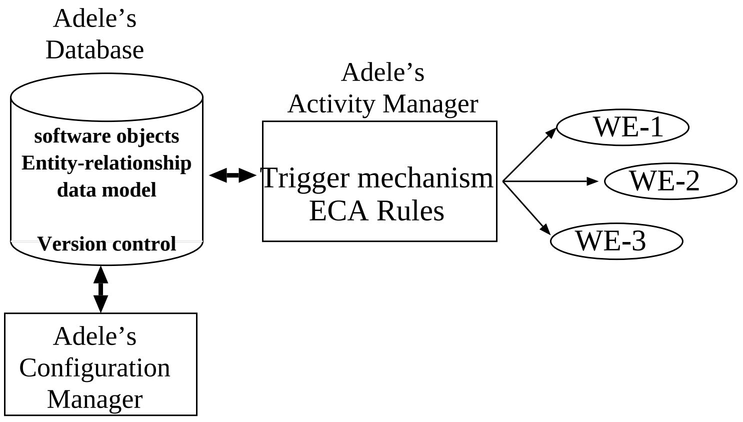

Figure 1: Adele 2's conceptual architecture.

3 Controlling long transactions

In order to support the software development processes, a SEE must provide mechanisms to control short and long transactions [12]. All DBMSs support short transactions as an atomic unit of work. That is, when an operation is performed on a database object a short transaction is opened. If the transaction finishes successfully, the effect of all the operations is made permanent in the database. If it aborts, all the database updates performed by the transaction are cancelled. It is typically assumed that the work done during a transaction can be redone in case of a failure and it is possible to wait for the transaction commit. This mechanism is only appropriate where transactions spend a few seconds.

However during the software development process, many tasks have a very long duration. Thus, short transaction mechanisms (*waiting*, *deadlocks*, and *rollback*) are not applicable. Therefore in a SEE context we need the mechanism to control and support long transactions. In general, long transaction management is based on a *check-in/check-out* model. Such a model supposes a versioned central database where baseline objects are recorded and are accessible in read only mode to all users. When an object update is request the object that will be affected by the change is *checked-out* into a work place (in general into a user's directory). The objects copied by one user are write locked for all other users. However as the database keeps the last version of the object, other users can access without *waiting* problems. When the operations on the checked-out object are finished, the object is *checked-in* back to the database, and a new version is created [14].

In Adele, short and long transaction mechanisms have been implemented for supporting actions performed inside and/or outside the Adele-DB. While short transactions are implemented in a similar way to conventional DBMSs — with lock, rollback, and recover facilities —, long transactions are implemented by work environment mechanisms. However, long transactions in Adele are more flexible that the conventional check-in/check-out model, because it is possible to update in parallel checked-out objects depending on work environment coupling (see section 3.1). Adele also provides the composite model. That is, while in the traditional check-in/check-out model only one object is handled each time, in the composite model a set of objects is handled. In this model, a long transaction is considered as the time between the first checked-out object and the last checked-in object.

3.1 Work environment management

As mentioned before, many tasks have a very long duration in a software engineering environment. This kind of activity is not performed in the database but inside the work environments and is

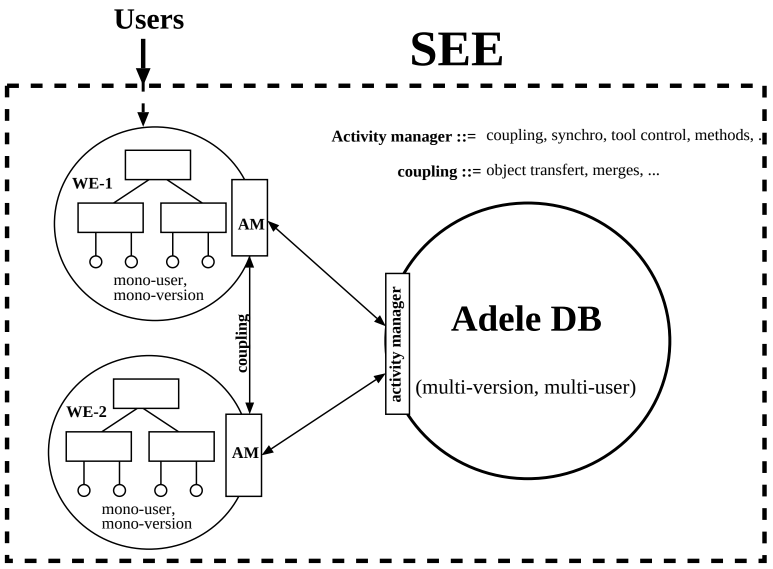
Users
SEE
Activity manager ::= coupling, synchro, tool control, methods, ...
coupling ::= object transfert, merges, ...
WE-1
AM
mono-user, mono-version
coupling
activity manager
Adele DB
(multi-version, multi-user)
WE-2
AM
mono-user, mono-version

Figure 2: Software Engineering Environment in Adele

managed by the work environment manager. An **environment** in Adele is a central database and a set of WEs with their management constraints and policies (figure 2)

Using the configuration building facilities, the symbolic links to the base and the contexts supplied by the Adele-DB, it is possible to create a mono-version view of a project. With this view, the user can work and use traditional software development tools. The symbolic links allow transparent read access to the objects in the base (a logical copy). In this way, only the objects intended to be modified are physically copied into the WE.

A WE is an Adele sub-database. The user can have access to all the information related to the objects in his WE, such as: the attributes and relations. A WE, associated with a user, comprises a mono-version view of the data base, a set of directories, a set of files, a set of tools and a task to do.

Objects may be shared between several WEs; we need to coordinate the WEs when shared objects are changed. We call this kind of synchronization **coupling**. Different kinds of coupling can be performed:

Hard coupling. a change to a shared object is immediately propagated to all copies of this object.

Tight coupling. a change to a shared object is propagated to the other copies of this object only when the changed copy is stored in the base (a merge of changes may be needed if concurrent modifications are performed)

Loose coupling. Given two modified copies, A and B, of the same object, nothing happens when the first changes (say A) are stored in the datebase, but storing the second change (B) triggers a merge between A and B changes.

No-coupled. It is not possible to modify an object in a concurrent way.

3.1.1 Work environment modeling

WEs are represented in Adele by special object types. Therefore development WEs, integration WEs, validation WEs have different characteristics (objects, tools, policies are usually different). The coupling policy, tools, tasks, semantic restrictions and WE identification attributes are defined in WE types, while the user and the objects are related to the instances of each WE. For example:

```
TYPEOBJECT WE_validation ISA WE ;
   DOMAIN
      WE>WE_validation:* ;
   ATTRIBUTES
      user = STRING ;
      directory MSUB := ~/VALIDATION/ ;
      purpose = STRING;
      coupling := no-coupled ;
--TOOLS
      tester := tool>test:newtest ;
      link := tool>comp:link ;
END WE_validation ;

TYPEOJECT WE_develop ISA WE ;
...
--TOOLS
      compC := tool>comp:cc ;
      compP := tool>comp:pc ;
      tester := tool>test:newtest ;
END WE_develop ;
```

The example shows "`WE_validation`" and "`WE_develop`" WE type definitions in Adele. Attributes of the "`coupling := no-coupled`" kind are constant attributes: all instances of the "`WE_validation`" type will have the same "`attribute = value`" pair. STRING means the attribute may have any string as a value; a value enumeration means the attribute can have any of the listed values as a value. A WE type definition is divided into three parts:

`DOMAIN` The domain describes where and how the WE instances will be stored in the base

`ATTRIBUTES` In this section we define the attributes:

- `directory` : defines from which directory this WE will start
- `purpose` : must be filled in with the WE objective
- `coupling` : defines the coupling for that WE type. By default, the WEs of this type are not coupled

`TOOLS` In this section we define the tools used by the WEs. The tools are also stored in the base like any other object. For example, the C compiler is contained in an envelope (defined by the attribute `compC`) and stored in the base in the document `tool>comp:cc`. As tools (envelopes) are stored and managed by the base, we have tool evolution history and so we can propagate tool modifications to their dependent objects.

An instantiation example.

We suppose that the Sun "`formatter:I:confUnix`" configuration is to be validated by three different users, each one in charge of some of the modules. For example, the WE for user "karim" is instantiated by the following command :

```
makewe   WE1 -c formatter:I:confUnix
         -t WE_validation
         -p formatter:I:confUnix = valid
         -u karim
```

This command asks for the creation of a WE called `WE1` (`WE_validation` type). This WE shall delivery the "`formatter:I:confUnix`" configuration in "`valid`" state.

3.1.2 Relationship between Adele objects and work environments

In figure 3 we show how the WEs are represented in the Adele structure and their relations with other objects.

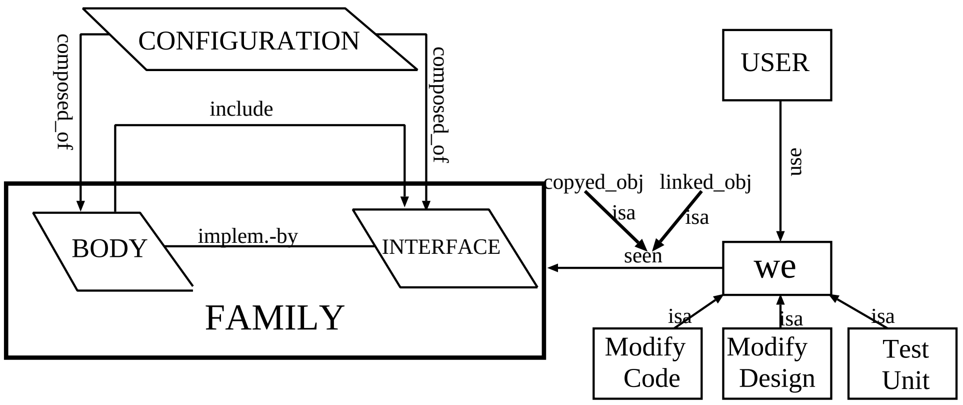

Figure 3: Relationship between Adele objects and WE

In this paper, we are only interested with "`linked-obj`" and "`copied-obj`" relations since they are intensively used by the activity and task managers.

The "`linked-obj`" and "`copied-obj`" relations associate WEs with the objects contained in that WE; "`copied-obj`" relation associates WEs with objects *extracted* from the database (i.e. *physically copied*); whereas "`linked-obj`" relation assoicates WEs with objects referenced by soft links (i.e. *logical copies*). These relations allow the activity manager — via the trigger mechanism — to synchronize and control the processes between WEs. We show below how these relations are defined in Adele.

```
DEFRELATION linked-obj ;
        DOMAIN [type = ws] -> [type = doc ] ;
        CARD N:N;
        TRIGGER ...
        ATTRIBUTES
            status = invalid, valid := invalid ;
```

```
END linked-obj ;

DEFRELATION copied-obj ;
        DOMAIN [type = ws] -> [type = doc ] ;
        CARD N:N;
        TRIGGER ...
        ATTRIBUTES state = exp, compiled, tested,
        released, officiel := exp ;
END copied-obj ;
```

These descriptions mean that relations are defined between a WS and documents. `CARD` denotes cardinality: any number (`N`) of associations may come from or go to a given node. The key-word `TRIGGER` will be explained in the section 4. Adele allows attributes to be associated with relations. In this example '`status`' and '`state`' attributes express the state and status of the relation, where

`status = invalid, valide := invalid`

means that the possible values of the attribute status are `invalid` and `valid` with a default of `invalid`.

Now that the infrastructure has been defined, we will see how the Adele activity mechanism allows definition and enforcing of a WE policy.

4 Activity management

Software DBMS's manage a large amount of dynamically shared data and require assistance to manage crucial situations. For instance when a module interface is modified, we need to evaluate the impact on modules using this interface, notify the impacted modules and if necessary to recompile them. Dynamic aspects have been investigated in many software Databases as a way to provide this kind of assistance. An active DB is useful for implementing management policies in a general and flexible way. The information to manage is essentially a versioned DB; the only efficient mechanism in such a context is the trigger mechanism associated with an event-condition-action formalism. This formalism allows action definition to be executed automatically when some conditions hold, as for instance checking integrity constraints or propagating changes.

4.1 Adele and the event-action concepts

The formalism involves two concepts: **event** and **action**. An event signals a state change during a database operation. The action is the code to execute when an event occurs. Adele includes concepts borrowed from object oriented languages (types, inheritance, encapsulation, etc.); mechanisms for propagation control and a tight control of external tools and objects (the WE). These concepts extend the classical trigger mechanism. We shall describe briefly the extension of the mechanism in Adele and its evolution as an activity manager.

4.2 Trigger description: benefits of the object orientation

The object orientation of Adele offers many advantages in the modeling of trigger concepts. A **trigger** is the (dynamic) association of an event with an action; and is expressed as "ON event DO action". Events and actions are independent, user-defined objects, while triggers are associated with object and relation **types** and, like object types, they can be aggregated, inherited (refined) and classified.

- Event instantiated on object. These events are triggered whenever a DB operation accesses an object. With this kind of event, semantic rules related to object types may be expressed.
- Event instantiated on relations. These events allow the management of the ripple effects produced by an action on an object related to other objects. This kind of event allows definition of a policy to deal with inconsistent situations. For instance, the modification of a module propagates effects on the configuration that includes it. The DB detects automatically this inconsistency via an event on relations.

Triggers are similar to production rules since they define the dynamic behavior of all the objects of a given type: the encapsulation principle is respected.

The actions associated with triggers fall into one of the following categories:

Pre actions. Before the execution of an operation on an object of type T, an event occurs and the triggers defined in the type T as pre actions are executed (those for which "evt" in "ON evt DO Action" is true). This kind of action allows testing of preconditions and command extensions.

Post actions. After the execution of the operation but before its commit, triggers in post-action are executed. These triggers can analyze the consequences in the database and, since they are executed inside the transaction they can undo (rollback) the operation. They can also extend the command by performing other computations.

After actions. After an operation is committed, other triggers are executed. These actions make it possible to modify the database after the command (for instance asserting new states).

Abort action. If the operation fails or aborts, all actions including those performed by pre- and post-triggers are undone, then abort actions are executed. This mode allows execution of actions in response to abnormal behavior.

Pre triggers and post triggers must succeed for an activity instance to be allowed to start and commit.

4.3 An application example

We want to define a "development WE" as the place where the following policy is enforced:

a module can be copied (Checked-Out) in a WE or referenced directly in the database by soft links. When a changed module is replaced in the Data Base (Checked-In: new revision) it must be immediately available in all the other WEs where it will be tested. A revision is considered official when validated in all WEs.

In order to specify this example, first we define the relevant events and their relative priority:

```
DEFEVENT
   Delete_Official = [!cmd = delete, state = official] PRIORITY 1;
   replace  = [!cmd = replace]  PRIORITY 2;
   valid  = [!cmd = validate ]  PRIORITY 3;
   %changes are validated by a WE

   invalid = [!cmd = invalidate]  PRIORITY 4;
   % changes are invalidated by a WE

   officialize  = [!cmd = officialize ] PRIORITY 5;
   % changes are validated by all WE
END ;
```

Priorities indicate in which order the events are to be taken into account (lower number first).

```
TYPEOBJET prog ;
   TRIGGER
1     PRE ON Delete_Official
  DO ABORT ;
2     AFTER ON valid
    DO "Check_Official" ;
```

This trigger, associated with all the programs ("`type = prog`") specifies what to do for `Delete_Official` and `valid` events.

1. before (PRE) executing the action (destruction of an official object), the action is aborted (primitive ABORT): it is not possible to delete a program in the official state.

2. after a "validate" command ("`!cmd`" is the current command), Adele has to check if the revision can be set into the official state (user defined command "`Check_Official`"). A revision can be official only if all the "`linked_Obj`" relationships have the "`status = valid`" attribute (line 3 and 4 below).

```
DEFACTION Check_Official ;
3  FOR r IN *-Linked_Obj- DO -- for all relation Linked-Obj
4    { IF NOT [r%status = valid]
        THEN RETURN ; } -- exit if one is not validated
   "officialize %name" ;
```

```
END check_official ;

DEFACTION officialize ;
   "MakeAttribute %name -a state official" ;
END officialize ;
```

For any relationship 'r' of "Linked-Obj" type leading to the changed object (Line 3), if 'r' does not have valid as status then the user defined command "officialize" is not executed. The triggers of a relation are executed when the event occurs on the object destination of the relation. The current object becomes the object source of the relation (the event has been propagated from the destination to the source of the relation). In Adele, a WE is represented by an object. For our application we consider the linked-obj relation: the source object is a WE, destinations are the objects which the corresponding WE refers to by a symbolic link (logical copies).

```
TYPERELATION Linked_Obj;
   TRIGGER -- Propagation on relation Linked-Obj
      POST
         ON replace DO
      mail -s "revision to test: %name " !Sourcename%author;
         ON valid DO
            IF !Sourcename%author = !curentuser
            THEN MakeAttributeRelation !Destname -a status valid ;
         ON invalid DO
            IF !Sourcename%author = !curentuser
            THEN MakeAttributeRelation !Destname -a status invalid ;
      AFTER
         ON officialize DO
            mail -s "module !name is official" !Sourcename%author ;
END Linked_Obj ;
```

After a replace, mail is sent to all the owners of a WE with "Copied_Obj" relation on the replaced object; after a "validate" command, the "status=valid" attribute is set on the relationship that links the WE and the revision; after an "invalidate" command, the "status=invalid" attribute is set on the relationship that links the WE and the revision.

The whole "PRE, command, POST" is a transaction; any failure completely undoes the command. In our example, every Work Context may reject a replace command, when evaluating the pre-condition (PRE) or the post-condition (POST).

This application shows how it is possible to:

- enlarge existing commands (`replace` in our example),
- define new commands (the actions are user defined commands),
- associate the object type definitions with their consistency controls,
- automate propagation.

4.4 Evaluation

The activity manager is a basic mechanism, efficient, versatile, but it has little knowledge of what is done. We found the following weak points in the activity manager:

- The association of behavior with object type on the one hand and the use of relations in the other hand results in a distribution of the information that makes it difficult to have a general view of the activity control.
- When long transactions are involved, it is not natural to use the activity manager. However this difficulty is partially overcome, when creating high grained objects (such as a WE) representing the long transaction and controlling its state.
- The activity manager works fine when the behavior can be statically expressed from well known information. We found the need to express more fuzzy policies, and thus to generate dynamically activities depending on multiple conditions (a planner). However, in Adele the context is taken into account by the dynamic creation of relationships, since propagations are performed by relationships.
- Reasoning and interactive activity support (answering questions, guiding users) are not natural in the activity manager.

5 Related Work

Among the several kinds of process language that the software process community has been using to model the software processes into process-oriented software engineering environments (POSE), the rule-based, the procedural, and the event-condition-action languages are the most representative.

5.1 Rule-based POSE

Rule-based POSEs are advantageous because the software process can be described by using logical declarations allowing the users to specify that they want rather than a detailed specification of how the results are to be obtained [31]. Using this behavioral approach, various prototypes of rule-based POSEs have been built, e.g., Pcte/Alf [18], Peace [1] and Marvel [16].

Although our approach is not completely declarative, we consider that rule-based facilities are important when executing software processes. Therefore, we have used rules in order to control method execution as well as to allow the Adele system to take initiatives when possible, based on the rule conditions.

5.2 Programming-based POSE

Osterweil [24] has proposed the procedural approach. The key idea is a complete algorithmic description of software process by means of a formal language. This description is considered as a specification of how a software process is to be managed in the SDE by users and tools. Several ongoing projects have been influenced by this idea, resulting in the construction of some experimental POSEs — for example, Triad-CML [26] and Arcadia-Appl/A [28]. Both these POSEs have extended the Ada language with new capabilities to support software processes. The main drawback with this approach is that no algorithm of a particular software process can be described completely in advance. Another technical problem with these systems is the need to modify the Ada compiler.

Of course, we also have been influenced by this idea — the process type is described, in part, by a procedural formalism — however, our solution is more flexible than the systems we have quoted. Our language is interpreted and provides late-binding facilities. With these characteristics, it is easier to adapt the changes in the environment, without changing the process description.

Other POSEs, although much influenced by the procedural approach, have investigated other kinds of programming paradigms — for example PSS-PML [7]. Adele language is also inspired from object-oriented languages and systems. The software processes are described using an OOER [1] formalism. We have broadly used type inheritance, methods, and triggers. Like PSS-PML, we use roles to control software activities. Unlike PSS, which uses roles only to model user activities, we have extended this notion to capture all resources manipulated by the activities.

5.3 Trigger-based POSE

In the trigger approach, software processes are modeled by a set of event-condition-action rules that are interpreted by a trigger mechanism tightly connected with a software database — for example, Arcadia/Appl-A, Alf/Masp [22] and AP5 [20]. Appl/A has extended the Ada language with programmable trigger-upon relations. The automation of the software process is done by these triggers.

AP5 [15] is an active, in-core, relational database extension to Common Lisp. AP5 users can register triggers with the database. A trigger consists of a condition, written in first-order logic with temporal extensions, and a body, written in Lisp. Triggers can guarantee data base integrity by modifying or rejecting database transactions. They can also invoke non-database activities in response to transactions. An AP5 trigger condition defines a database event to be announced, while a trigger body represents the code executed when an event is announced. AP5 events are announced after transactions are submitted but before they commit, allowing transactions to be modified or aborted.

In PCTE+/Alf, triggers control communication among parallel tasks by capturing changes on

[1]Object Oriented Entity-Relationship-Attribute

database objects [21]. These tasks are modeled, however, as in the Marvel 2.0 — i.e., pre- and post-conditions enveloping foreign tools — and managed by a specialized expert system shell connected to PCTE+.

Unlike Arcadia/Appl-A and Alf/Masp, the Adele trigger mechanism can be attached to both entity and relationships to envelop methods. Four types of trigger coupling can be used (pre, post, after, and error), thus providing greater flexibility.

6 Conclusion

The Adele project proposes an architecture for activity coordination in a software production environment based on two layers: the underlying layer (in the Adele kernel) is an efficient trigger mechanism with propagation control based on graph management. This layer is used for database housekeeping (consistency, extensibility, customization, etc.) and simple policies. The second layer (the task manager) uses a rule-based strategy and is dedicated to higher-level policy management.

Currently, to write a high level task, the user has to define the needed triggers and propagation, and then the task manager program that will be triggered by the low level triggers. We are defining a formalism that will generate code for both levels simultaneously [19]. It will become the user interface to Adele Process programming. We expect, that way, both good efficiency and high level control.

References

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

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

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

[4] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In Dowson [10], pages 159–170.

[5] N. Belkhatir, J. Estublier, and W. L. Melo. Software process model and work space control in the Adele/Tempo system. In Osterweil [23], pages 2–11.

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

[7] R.F. Bruynooghe, J.M. Parker, and J.S. Rowles. PSS: a system for process enactment. In Dowson [10], pages 128–141.

[8] S. Dart. Concepts in configuration managements systems. In *Proc. of 3rd Int'l Workshop on Software Configuration Management*, pages 1–18, Trondheim, Norway, June 12–14 1991.

[9] F. DeRemer and H. Kron. Programming-in-the-large verus programming in the small. *IEEE Transactions on Software Engineering*, 2:80–86, June 1976.

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

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

[12] P. H. Feiler. CASE and CAPE: conflict of interest. In Schafer [27].

[13] P. H. Feiler and W. S. Humphrey. Software process development and enactment: Concepts and definitions. In Osterweil [23], pages 28–40.

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

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

[16] 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.

[17] D. Leblang and R. P. Chase. Parallel building: experience with a case for workstations networks. In *Proc. of the Int'l Workshop on Software Version and Configuration Control*, Grassau, FRG, January 27–29 1988.

[18] A. Legait, M. Menes, F. Oquendo, P. Griffiths, and D. Oldfield. ALF: its process model and its implementation on PCTE. In K. H. Bennett, editor, *Proc. of the 4th Conf. on Software Engineering Environments*, Software Engineering Environments — Research and Practice, pages 335–350. Ellis Horwood Books, Durham, UK, April 11-14 1989.

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

[20] K. Narayanaswamy. Enactment in a process-centered softwre engineering environment. In Schafer [27].

[21] F. Oquendo, G. Boudier, F. Gallo, R. Minot, and I. Thomas. 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, April 1991.

[22] F. Oquendo, J.-D. Zucker, and G. Tassart. Support for software tool integration and process-centered software engineering environments. In *Proc. of the 3rd Int'l Workshop on Software Engineering and its Applications*, pages 135–155, Toulouse, France, December 3–7 1990.

[23] L. Osterweil, editor. *Proc. of the 2nd Int'l Conf. on the Software Process*, Berlin, Germany, February 1993. IEEE CS Press.

[24] 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.

[25] C.V. Romamoorthy. Programming in the large. *IEEE Transactions on Software Engineering*, 12(7):1145–1154, July 1986.

[26] 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.

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

[28] 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.

[29] W.F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proc. of the 6th Int'l Conf. on Software Engineering*, Tokyo, Japan, September 1982. IEEE CS Press.

[30] W.F. Tichy. Rcs — a system for version control. *Software—Practice and Experience*, 15:637–654, 1985.

[31] 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.