Supporting Software Maintenance Evolution Processes in the Adele System

Noureddine Belkhatir [*] Walcélio L. Melo [†] Jacky Estublier Mohamed A. Nacer

Paper published in the *Proc. of the 30th Annual ACM Southeast Conference*, Raleigh, NC, April 8-10, 1992.

Abstract

One of the major problems encountered when developing large systems is related to maintaining an operational and responsive software system, once it has been accepted and put into production. This problem is referred to as Software maintenance. Evolution is central to Software Maintenance, responsible for ensuring a longer working life. Many Software Engineering Environments (SEEs) have been constructed in order to support maintenance activities. In this paper, we will first present major development in SEEs to support Maintenance. Afterwards, we will describe the main ideas behind the design and implementation of the ADELE system, a third generation SEE. Following this, we will give an example to illustrate the way a process model can be described on top of ADELE using an example of evolution maintenance. This approach is heavily based on event-condition-action formalism. We will illustrate the way in which communication and coordination of the activities carried out by different users are automated.

Key words CASE; software engineering environment; maintenance; software process; programming in the large; cooperative work; event-condition-action; trigger.

[*]Address: Laboratorie de Genie Logiciel, BP 53X, 38041 Grenoble, FRANCE. E-mail: {belkhatir, wmelo, estublier, nacer}@imag.imag.fr

[†]Melo is supported by Technological and Scientific Development National Council of Brazil (CNPq).

1 Introduction

Controlling the evolution of software products in order to extend their working life and reduce maintenance costs is a major commercial and scientific challenge. Significant academic and industrial studies have proved that maintenance evolution in large software: (a) is an activity that involves different groups of people working over a long period of time; (b) is an activity present in the whole software life cycle, because software changes throughout its life; (c) consumes much effort and is not sufficiently assisted; (d) is centered on communication and coordination concerns.

In this paper we will concentrate on the three following aspects:

1. In order to support activities for large projects, the environment designers are left with the following questions: "What is the structure of the environment? How are policies and activities supported and enforced ?". Therefore, many Software Engineering Environments (SEEs) have been constructed in response to the following requirements: to guide and assist project teams when carrying out their activities according to the project policies; to control the ordering, synchronization, communication, and concurrency among those activities; to take into account the objects produced and consumed by activities; and to describe the user's capacities and responsibilities in order to control the different roles they can play when performing activities. We will present the main developments, highlighting the recent approach in software management.

2. A description of the main ideas behind the design and implementation of the ADELE SEE. ADELE provides a language in which the static aspects of a SEE are described (objects, relationships among objects, etc.) as well as the dynamic aspects (consistency rules, meth-

ods, etc.). The static aspects are defined by the ADELE data model which is based on the entity-relationship model extended with version, multiple inheritance, and schema partition supported by a distributed multi-user and multi-version software engineering database. The dynamic aspects of the objects and the behavioral aspects of the SEE are described by an event-condition-action (ECA) formalism supported by a trigger mechanism.

3. An illustration of the way a process model can be described on top of Adele. We give the ADELE solution for an example of evolution maintenance to control the changes in the data and processing environment. The emphasis is on the programming-in-the-large problems. We will illustrate the way in which communication and coordination of the activities, carried out by different users, are automated.

2 Related works

Considering the work reported in the literature concerning SEEs for the management of the software process, we may distinguish three generations:

The toolkit generation. In the systems of this generation, the tools surround the standard file system in order to support project life-cycle activities, but without appropriate mechanisms to integrate and coordinate them. The main advantage of such SEEs is their low cost constructions, but since a global process model and the mechanisms to support it do not exist, the policies concerned with coordination and control of activities can neither be expressed nor enforced automatically.

The hardwired generation The key idea of this approach is the development of a basic support to manage the objects produced during the software life cycle. Here, an SEE is built from scratch and custom-built tools are integrated around a common object management system (OMS) and/or a common user interface system (UIS). In such systems policies are enforced to control the use of system capabilities, coordinate concurrent activities, and enforce team communication policies, e.g. DSEE, INTERLISP, and NSE. However, such SEEs are typically quite inflexible because the policies supported by the system cannot be tailored.

The customizable environment generation Many studies [9, 12] have highlighted the drawbacks of the hardwired generation with its inability to adapt to the constant changes in the software process. Thus, in order to improve productivity and quality during maintenance, studies based on the development of the customizable SEEs have been conducted using DBMS (Data Base Management Systems) technology. New conceptual models adapted to Computer Aided Software Environments (CASE) systems have been developed[7]. Semantics and object oriented models are used[4]. The object base evolves towards a DBMS which supports not only the objects produced and consumed by activities but also methods and policies, users in their different roles, and the tools used to perform those activities. The originality of this approach relies on the integration of two aspects, the static aspects — data and product modeling — and the dynamic aspects, i.e. the process modeling. Nowadays, the software community agrees on the necessity of a DBMS able to support all aspects of the software process, and the research of an appropriate, general and interpretable formalism for modeling such aspects is underway [6, 10, 11].

3 Customizable environment objectives

The customizable approach represents a significant and novel approach to software development paradigms. Gandalf was an early example of this new approach, providing a general language, based on attribute grammar, for the specification of the specialized programming SEE. However, it only took into account the programming-in-the-small problems (edition-compilation-interpretation-debug). Now, since many other prototype systems have been built, the requirements for a customizable environment can be highlighted. In general a customizable environment must support : (a) an explicit description of data, users, tools, activities. (b) all the software life cycle steps; (c) the mechanisms to describe and enforce the communication and coordination policies; (d) and assist the users when performing software activities;

In order to achieve these requirements, there is agreement on the capabilities that must be provided by such a system:

- A versioned repository where all software artifacts can be managed. Such a repository must be driven by data model incorporating both

Entity-Relation and Object-Oriented concepts. The former concept is better adapted to modeling software structure, and the latter concept to modeling behavioral aspects of the SEEs.

- The explicit description of the activities, activity breakdown, and the coordination policies among them. Many formalisms have been proposed for the description of these capabilities, each one with its advantages and drawbacks. Among the more significant, we can distinguish three approaches: (a) the prototyping and simulation approach, e.g., MELMAC [3]; (b) the planning and reasoning approach, e.g., MARVEL [5] and EPOS [2]; (c) the communication and synchronization approach, e.g., APPL/A [11] and ADELE [1];

- A Sub_DataBases mechanism[7]. That is, a central shared database, and a set of possibly overlapping Sub_DataBases, supporting Long Transactions, along with the mechanism needed for their synchronization.

4 Adele background

ADELE was, in its previous versions, mainly a configuration manager and since 1987 it has been used in European aeronautics (for instance the Airbus and Ariane projects). ADELE is now a commercial product and it is used by industrial applications and European projects (Airbus, Hermes, Rose, Reboot...).

The ADELE [1] system is composed of:

- A multi-version and multi-user software engineering database. This base may be distributed on different sites connected by a local network and it can be used by application programs through an RPC programmatic interface, by a command language through the Unix shell interface or by a graphical interface. Both short and long transactions (based on check-in/check-out mechanisms) are supported.

- A configuration manager. The configuration manager is able to compute the list of objects needed to build a configuration based on a set of constraints over the objects and using a multiple graph closure. Defining a configuration takes only a few lines of constraints and not a large user defined list of components.

- An activity manager (AM). This is the active component of the ADELE-DB. It is used as a general purpose rule formalism to maintain database integrity, to integrate external tools in the ADELE environment, to synchronize work environments (WEs) and to monitor activities that are carried out in the WEs. The AM is based on a trigger mechanism allowing to execute actions in the database as well as to communicate with external tools. The user can define object behaviour as well as propagation along relationships.

These components are integrated using the Adele Modeling Language which provides ways to define the static and dynamic aspects of almost "any" SEE. The static aspects are modeled by an "object-relation" data model, which is derived from the Entity-Relationship model extended with composite and versioned objects, and multiple inheritance. This means that users can create new entity or relationship types, to relate these types in a direct acyclic graph, to define new functions on these types or inherit them from other types in the graph, and to encapsulate those functions with the type.

In order to describe the dynamic aspects, an ECA formalism is provided. With these two formalisms (data model and ECA) we can program CASEs on the top of the Adele kernel. For instance, the WE manager is programmed by the ECA formalism and modeled by entity and relation types.

Adele behavioral model

The dynamic aspects of the environments built on the ADELE kernel, i.e. environment policies, are defined using the ECA formalism. This formalism involves two basic concepts: events-condition and actions. *Events* are used to control activities (navigation as well as modification) in the database. An *action* is a set of operations activated by a trigger when an event occurs. Actions can abort transactions, or perform further modifications to the database, which may in turn fire triggers.

Many other database systems have used trigger formalisms in order to describe and automatically enforce integrity constraints [8]. In our case, triggers are used as a general enaction mechanism, for consistency control, of course, but also for the enaction and control of the Software Processes, as is achieved in the APPL/A and ALF[12] projects. ADELE was one of the first systems to have experimented triggers on full size projects. We extend the traditional trigger mechanisms to deal with several types of coupling modes. Trigger coupling is used to determine when actions must be undertaken in relation to the moment when the events occur. In ADELE four coupling modes can be specified:

PRE mode. In this mode the actions are executed before the execution of the operation that has fired the trigger;

POST mode. In this mode the actions are executed after the execution of the operation that has fired the trigger, but before this operation has been committed. Thus the operation can be completely recovered .

AFTER mode. In this mode the actions are executed just after the operation has been commited.

ERROR mode. In this mode the actions are executed in the event of an operation failure.

5 Software change process management

In this section we shall demonstrate some of the Adele system capabilities by using an example. We shall be concerned with the description of user coordination policies and policy enforcement in large scale programming.

5.1 The software change example

The example is scoped as a relatively confined portion of the software change process. It focuses on design, coding, testing and managing a rather localized change in the software product. This is prompted by a Change Request (CR, either for corrections or enhancements). We shall assume that the proposed CR has already been analyzed and approved by the configuration control board. The Design WE determines on which configuration baseline the change is to be performed and which modules are to change. The Manager WE decides which users will be involved in this change, which modules each one will have to change. We shall assume that some modules can be changed concurrently in different Work Environment. Each user works in a different "modify-code WE", sharing the same modules; when all modify-code WE are resumed, the Test WE can be initiated (see figure 1).

We have concentrated mainly on programming-in-the-large, therefore, we are not interested in how the process steps are carried out, but in how we plan the software process steps and how we coordinate their execution. Thus, we do not check how the software engineer modifies the module, i.e. code-edition-debug process, but how he coordinates his

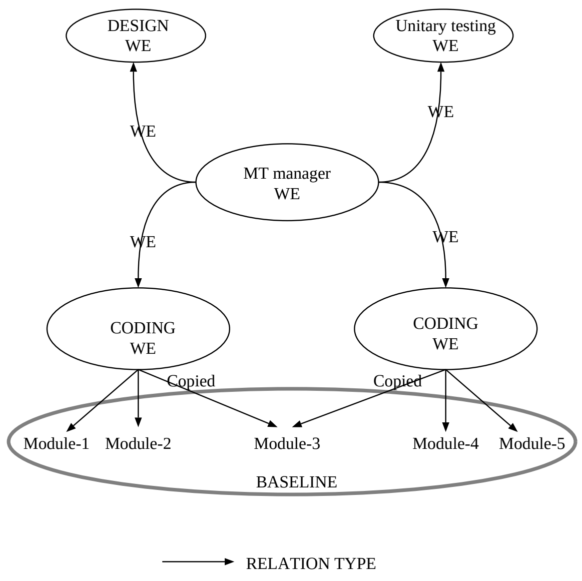

Figure 1: Fragment of the software change process

activity with the other software process change activities.

5.2 The Adele solution

We shall assume that the software product is managed by the Adele database and that a set of Work Environments supports the development. In WEs, activities are carried out using tools such as compilers, editors, etc.

Our solution uses ECA rules and customized commands to control the synchronization of activities among WEs and between the WE and the database, mainly at the beginning and at the end of each step. A step is a long transaction, thus it is executed in a Work Environment. We define a type of WE for each step of the change process model (manager, design, modify code and test).

5.2.1 Product structure

The experiment has shown that the software development process must be tightly coupled with a configuration management system. Adele provides a data model specially designed for managing the objects handled in the configuration management context. The basic type of this particular model is a **family**. A family may be thought of as a high-level module composed of a set of interfaces and a set of realizations, where each interface and realization may be versioned. Designs, tests, change request planning, etc are kind of documents associated to

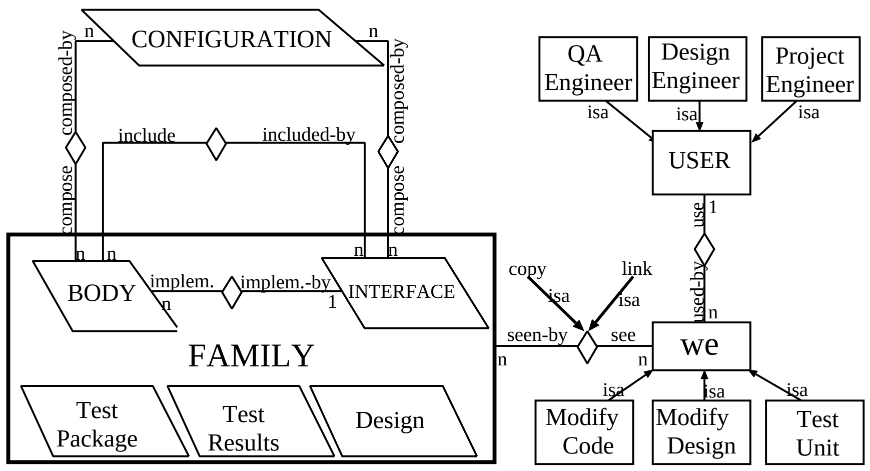

Figure 2: The ADELE data model instance for management of the modular programmes

family objects, and they can also be versioned. Figure 2 shows this structure.

5.2.2 Process change modeling: code modification

The **"modify code"** step is carried out by the software engineer. The goal of this step is the implementation of the solution proposed by the design document. The source code must be compiled and the corresponding binary must be recorded in the database with the source. Several users can code in parallel. The design and modify code steps can begin together, but the modify code step can only finish after the design has been approved. The policies associated with this step are distributed inside the following types:

- **module data type** defining the actions that must be performed when a module is *check-in/check-out*.
- **copied relation type** which stores the relationships among objects and work environments. In other words, the modules are copied to WEs to be manipulated by users, this informations is modeled in ADELE by the `copied` relation.
- **WE relation type** which links the `Manager WE` with all the `WEs` performing the same logical change.

The *begin_modify_code* action is used to create an instance of modify_code WE.

```
DEFACTION begin_modify_code;
  create-we %user -t modify-code
    -c %conf -o %module_to_change;
  madele %user -t modify-code;
END begin_modify_code;
```

Then madele is called, an ADELE tool that generates a makefile from a WE.

`create-we` is a user defined command that creates the WE from "%conf" configuration components baseline i.e. a physical copy of those components which the current user is allowed to change (`-o %module_to_change`), and a logical copy of all other components. A "`%copied`" relationship is instantiated between each copied module and the work environment instance; the `status` attribute of the copied relationship is defaulted to "`UnderWay`".

```
DEFACTION end_modify_code;
  check-in  -md;
  delete_WE %name ;
END end_modify_code;
```

The `end_modify_code` command is used to finish the modify-code step. The definition above says that when this command is executed a check-in operation is executed on each modified component (-md option). If another user checked-in the same module before, the merger tool will be automatically started by ADELE and the merged result recorded. At the end_modify_code, the WE can be deleted.

```
TYPEOBJECT module ;
DEFATTRIBUTE
  state = debug, compiled := debug ;
  design := !famname.design ;
 PRE
1 ON begin_modify_code DO
```

```
     IF [%design%state != completed] THEN
        ABORT;
2 ON end_modify_code DO
     IF [%design%state != approved] THEN
        ABORT;
END module;

DEFRELATION copied;
DEFATTRIBUTE
  status = valid, UnderWay := UnderWay;
  PRE ON check_out DO
3   mail -s "warning" !sourcename%user
            <<+ "...."

  POST ON end_modify_code DO
4   IF [%status != valid] THEN ABORT;

  AFTER ON end_modify_code DO
5   IF "compile %name" THEN
     {cg_attr !destname -a state compiled;
      ci_bin !sourcename; } ;
    ELSE rm_attr !destname -a state

DEFRELATION WE;
6 POST ON true DO propage ;
```

This piece of program stipulates that:

1. the *begin_modify_code* command is aborted if the design step has not finished. In other words, the modify-code step cannot be started before the design has been completed; however it can begin before the design has been *approved*;
2. the modify-code step cannot be closed before the design document has been approved by the staff.
3. when a module is checked out in a WE, the users of the other WE with a physical copy of that module (i.e. they have a copied relationship with it) will receive a warning message, notifying possible change conflicts.
4. an modify-code WE can be terminated only when all its modules have valid status in all the WE owning a physical copy of it.
5. At the *end_modify_code* command the module is compiled. If the compilation succeeds the associated binary is also checked-in the database and the *state* attribute is changed to `comp` (*compiled*).
6. When events occurs in a WE, the manager WE is informed using WE relationship propagation.

Manager Work Environment

The project manager works in the "Manager Work Environment". The objective of this WE is to monitor the process. Since relation WE is a propagate relation, any event occuring on a WE will be noticed in the manager WE. That way the manager communicates with other WE and receives notification when a WE completes.

The Manager WE can work in two different modes:

1. **asynchronous coordination.** In this case communication between WE and the project manager is achieved by mails. The coordination is handled manually;
2. **synchronous coordination.** Here, the coordination is synchronized and expressed by ECA rules associated with the project manager type. These rules are executed when events are triggered on the associated modify-code WE, through the WE relationships. This is implemented as follows:

```
TYPEOBJECT manager is WE ;
POST
 ON end_design DO
1  IF [ %design%state=completed]
   THEN dispatch_work %CR_planning ;
AFTER ON delete_WE DO
2  IF NOT "lsr -r WE" THEN begin_test;
```

1. If an *end_design* command is executed and the associated design document state is *completed*, the modify-code step can begin. The `dispatch_work` command analyses the CR_planning document and creates all needed modify-code WE (calling the `begin_modify_code` command).
2. When a WE is completed, the associated WE relationship is deleted. If no more "WE" relationship exist, all WE are completed; the test step can begin.

Figure 3 shows the operations sequence of the software process change example.

6 Conclusion

The activity manager was experimented for one year. Two kinds of experiments were conducted: local prototypes, as for instance the implementation of current systems such as NSE, DSEE on top of Adele ; and industrial experiments, such as the

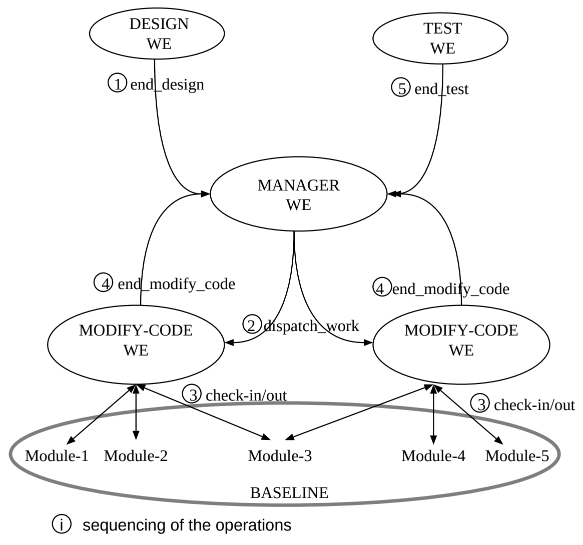

Figure 3: Operation sequencing of the software process

HERMES (the European shuttle) software environment where thousands of ECA rules have been written by the Hermes team to automate and simplify the development of their specific CASE. From these early experiments we can learn the following results.

We found the integration of ECA rules into the Data Model very valuable. From **Entity-Relation** model, we take advantage of the explicit use of relations at the conceptual level. We think the use of explicit relationships to model the context in which an object is used is one of our major features. It allows to define the intrinsic behaviour of objects independently, in the object type, and the context behaviour, in the relationship type. It makes the specification of a system more comprehensive and provides more flexibility when extending or scaling up the system.

From the **Object Orientation** approach, we take advantage of inheritance, encapsulation, etc. Clearly we obtain improved modeling both of object behaviour (object inheritance) and of context behaviour (relation inheritance). We extended the Adele system with Methods (since inheritance is different between method (overloading) and trigger (addition)), and differentiate the pre/post/after and error triggers. It is clear now that most solutions need to distinguish between pre, post, after and error triggers.

From **Data Bases** we take advantage of the transaction concept, and we integrated triggers with transactions. Pre-action-post is a single transaction, while "after" and "error" triggers are executed outside the main transaction. The transaction concept allows the association of consistency constraints with the objects and relationships directly or indirectly involved in an activity. The activity itself may ignore these constraints. A better separation of concern is achieved that way. Methods/actions only describe the intended result, the system dynamically checks the consistency constraints, depending on the actual context of the involved objects.

In classical DBMS, events are simple predicates and ECA are only used to enforce integrity constraints. This approach makes the use of triggers to control the software process difficult. In OODBMS, integrity constraints are embedded in methods e.g. ORION, CACTIS...

The Adele language is simple and efficient; in particular its late binding and delegation mechanism proved helpful in defining the instance behaviour at type level (dynamic substitution of context and object characteristics).

However, difficulties were encountered:

1. The Adele language is low level. A higher level language abstracting the details of the mechanism is needed.
2. The fragmentation of the ECA rules into different types and relations makes it difficult to have a clear picture of a complete process description.
3. The control of triggers sometimes proved tricky. There are risks of event explosion, of looping and of action duplication. Clearly we need a support for trigger programming and debugging, as well as a programming methodology.
4. High level reasoning, learning and explaining is almost impossible from a trigger description, because of its low level abstraction and its fragmentation.

We are currently trying to solve the problems encountered our experiments, by defining a set of tools for supporting event programming (validation, visualization based on petri nets, debuggers); and the definition of language(s), on top of our triggers, tailored to our application: the building and tailoring of Process-Oriented Software Engineering Environments.

References

[1] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In M. Downson, editor, *Proc. of the First International Conference on the Software Process*, pages 159–170, Redondo Beach, CA, October 21–22 1991. IEEE Computer Society Press.

[2] 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.

[3] W. Deiters and V. Gruhn. Managing software processes in the environment MELMAC. In *Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments*, Irvine, CA, December 3–5 1990. SIGSOFT Software Engineering Notes, 15(6):193–205.

[4] S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[5] 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.

[6] T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the 11th International Conference on Software Engineering*, pages 343–352, Pittsburgh, Pennsylvania, May 1989.

[7] W. Kim, N. Ballou J.F. Garza, and D. Woelk. A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.

[8] G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to Starburst: objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, October 1991.

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

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

[11] 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.

[12] J.-D. Zucker. ALF: accueil de logiciel futur. In F. Long, editor, *Software Engineering Environments - volume 3*, pages 21–52. Ellis Horwood Books, 1991. *5th Conference on Software Engineering Environments*, Aberystwyth, UK, March 25–27, 1991.