Adele 2:
A Support to Large Software Development Process *

Noureddine Belkhatir Jacky Estublier Walcélio L. Melo †
L.G.I. BP 53X, 38041 Grenoble Cedex, France
{belkhatir, estublier, wmelo}@imag.imag.fr

Abstract

After years of use of Adele1 (a Data Base for Version and Configuration management [11]), we noticed that we lack concepts and mechanisms related with activities: work environment control, users coordination and synchronization, method and tool control, etc. We notice also that, currently, a very large amount of work is needed to adapt a SEE to user requirements. From this experience, we implemented Adele 2 to provide a general support for defining and managing the dynamic aspects of a SEE, and easing the building of new SEE. This paper describes, from an example (workspace control), the concepts and mechanisms involved. It is shown how a deep integration of an activity manager with the Adele DB data model fulfills the basic requirements, and how a high level task manager coupled to a configuration manager can be developed.

1 Introduction

The development and maintenance of a software product is a complex task. On the one hand, there exists a large amount of different objects type with complex structure (the static aspect of SEE), and on the other hand, SEE is the place where activities take place; all together converging toward a deliverable product.

Experience until now has shown that support is needed for clarifying, formalizing, supporting and coordinating the involved activities. Current environments try to offer some assistance to support activities but results are very limited. Usually ad-hoc solutions are implemented and little assistance is offered. We lack a unified assistance for product development, which can be formally specified by the administrators or team leaders, and enforced by the SEE.

The Adele environment is specifically designed for supporting CASE applications in a multi-user and multi-version context. All the components of the environment are stored in a central versioned **database**. The Adele database is based on the entity relationship model extended with composite objects, OO features, long transactions and user defined commands and activities.

Users need to use standard **tools** as compilers, linkers, browsers that operate on **files** while the database manages different objects and concepts : aggregates, attributes, triggers, etc.

A **work environment** (WE) fills the gap between both needs: it is a sub data base where the files view is emphasized and extended with (very) long transactions (a work environment itself can be seen as a very long transaction) and management constraints and policies.

*Published in the *Proc. of 1st Int'l Conf. on the Software Process*, Redondo Beach, CA, October 21-22, 1991. IEEE Computer Society Press.

†Melo is supported by Technological and Scientific Development National Council Brazil (CNPq)

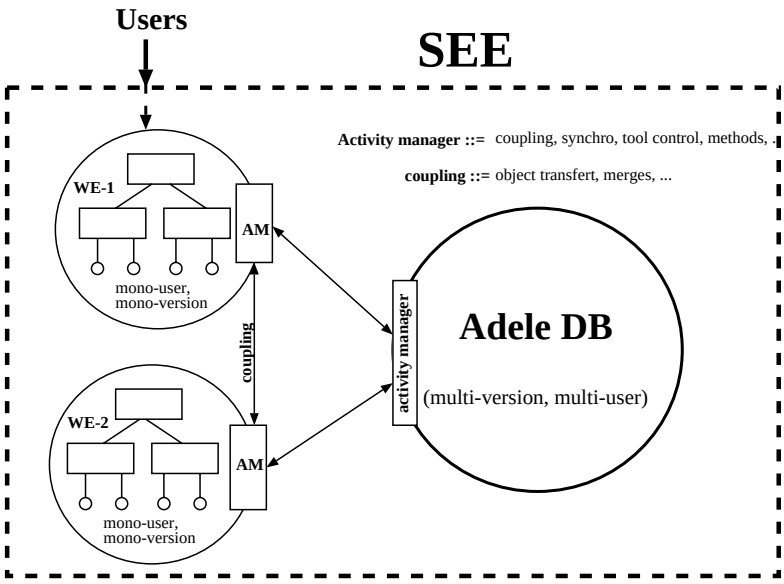

Figure 1: Software Engineering Environment in Adele

The coordination between work environments is managed by the **activity manager**(AM) according to each work environment management constraint (see figure 1). The activity manager provides the services needed to coordinate the different work environments and to maintain database and object integrity. A **task manager**(TM) controls and (sometimes) automates tool execution within a work environment, supports and promotes the reusability of already produced objects, with the objectives of increasing productivity and improving software quality.

Adele-DB is an active multi-user versioned program database. This database may be distributed on different sites connected by a local network and can be used by application programs through an RPC interface, by a command language (through the Unix shell interface or by a graphic interface). Adele manages arbitrary complex objects, ranging from elements (associated to a file) to projects and sub-projects. The objects are structured by relations and constitute aggregates. A module is a basic predefined complex object constituted by separated interfaces and bodies.

This paper presents the Adele mechanisms for activities and process support, with an example showing how a user specified work environment strategy can be specified.

2 Work Environment management

Since classic tools only know the concepts of file and directory and do not deal with versions, a WE must be a mono-versioned sub data base where these "low level" concepts are emphasized. Thus **a WE is a mono versioned sub DB** with :

- a set of objects (mainly seen as a tree of directories and files),
- a set of tools
- a set of methods and policies.

It is important to notice that while the file view is emphasized, a WE is still a DB. It is possible to ask for the file attributes and relationships and to manage any other non file object. A main

design decision is to clarify the relationships between the DB and the WEs. In previous Adele versions, WEs were implemented inside the DB itself, as provided by most current tools; then we noticed that the structure of the different sub DB is different. For instance, a given file may be under a given directory in a WE, and under another directory for another WE. The DB itself, for versions control purpose associates together the related objects. These different views hardly coexist in the same DB. In Adele, each WE is an independent tree of files and directories, with its own tools, methods and policies. In practice only the directory tree is duplicated; objects, tools and methods are shared in a transparent way between WEs. Depending on management method, the WE can be directly accessed by tools (open WE, as in NSE[8], DSEE), are completely controlled through specific command application (closed WE as in most tools : PCTE[4], Palas, software Backplane and so on). In the first case there is no overhead (assuming Unix symbolic links do not make overhead).

Since objects may be shared between several work environments, there is a consistency problem if concurrent changes are performed. WEs must be coordinated; we call this kind of synchronization **coupling**. Different kinds of coupling can be performed :

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

Tight coupling. The change of 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 change (say A) is stored in the date base, but storing the second change (B) triggers a merge between A and B changes (the NSE policy).

No-coupled. It is not possible to modify an object in a concurrent way (almost all SEE).

Coupling is not symmetric. A development WE may be coupled on an official WE (i.e. when a file is changed in the official WE, the change is propagated to the development WE), but not the contrary. Experience proved that effectiveness is greatly improved when coupling is allowed in a team working on the same product. Unfortunately, the lack of control and synchronization makes these unpractical solutions. In Adele, all the coupling are possible.

The goal of Adele is to support (almost) all policies. In order to validate our approach, we implemented the Work space management in use in some known tools. We were surprised to notice their weakness in supporting active behavior and that no coupling is allowed (except NSE). In this experiment Palas (a SEE for embedded real time Software), Andromede (an industrial SEE) and NSE (mainly used in academic) have been implemented, each one described in less than 10 pages of Adele language. This experiment proved that all SEE rely on a tiny set of concepts and mechanisms; and that improvements on Work space management policies are easy to perform, provided the right underlying mechanisms.

In the following we describes how a simple hypothetical WE can be defined, as an example of how can be implemented a SEE on top of Adele.

2.1 Work Environment modeling

WE sharing the same tools, methods and policies are said to be of the same type. Usual types are development WE, integration WE, official WE, etc. In Adele, WEs are aggregate objects (components are the objects to deal with); the WE type defines the tools to use, and the management method and policy.

Since Adele data model includes O.O. features, WE specific commands (as is "acquire" in NSE) are represented as the WE methods; the semantics of the aggregate is defined both in the constraints of the WE type, and in the relation type that links the WE object to its components. Being first class objects, WE can be sub-typed, refined, extended, etc.

```
TYPEOBJECT validation_WE ISA WE ;
   ATTRIBUTES
      user = STRING ;
      coupling := no-coupled ;
--TOOLS
      tester := tool>test:newtest ;
      link := tool>comp:link ;
END validation_WE ;

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

The example shows the "`validation_WE`" and "`develop_WE`" WE type definitions in Adele. Attributes of the kind "`coupling := no-coupled`" are constant attributes: all the instances of type "validation_WE" will have the value "no-coupled" as value for attribute "coupling".

In the "`ATTRIBUTES` section we define the attribute "`coupling`" that defines the coupling for WE of that type; WEs of this type are not coupled. TOOLS are modeled by attribute whose value is an Adele object name. The tools are also stored in the base like any other object. For instance, the C compiler is contained in an envelope (defined by the attribute "compC") and stored in the base as the object "$tool > comp : cc$". As the tools (the envelopes) are stored and managed by the base, the tool evolution history is controlled and we can propagate the tool modifications to their dependent objects.

2.2 Relationship between Adele objects and Work Environment

Adele relies heavily on relations. Relationships, as objects, have a type, with multiple inheritance, attributes, constraints, propagations, etc.

It has been found that a large number of different aggregates must be defined and controlled in a SEE. A configuration, a WE, a module are aggregates, but the associated constraints are fairly different. Existential constraints (the component disappear with the aggregate), sharability constraints (a component pertains to only one aggregate), and other constraints are defined depending on the kind of aggregate. In Adele, we define the semantics of the aggregate with the relations that link the aggregate to its components.

For that reason a WE is an aggregate. Its specific semantics is defined in the relations "`link`" and "`copy`".

In figure 2 we show how the WEs are represented in the Adele structure and their relations with other objects.

The relation "`link`" links a logical copy of a file to the same in the central DB, while "`copy`" links a physical copy of a file to the same in the central DB. These relations allow the activity manager (through the trigger mechanism) to synchronize and control WEs. We show below how these relations are defined in Adele.

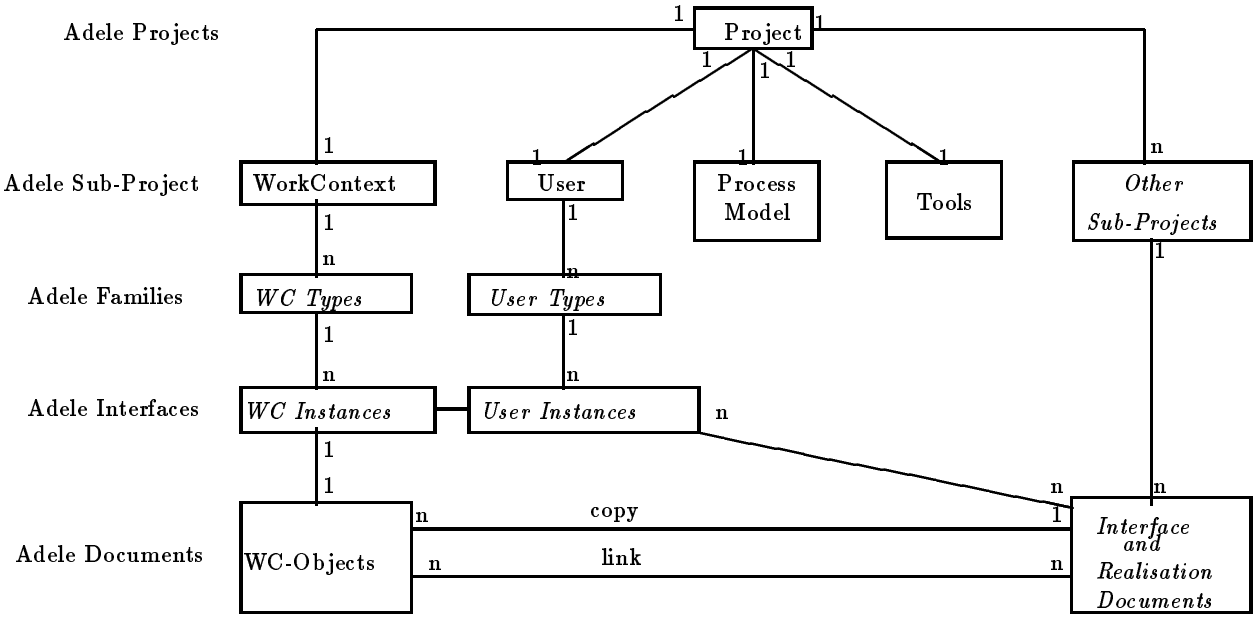

Figure 2: Relationship between Adele objects and Work Environment

```
DEFRELATION link ;
        DOMAIN [type = WE] -> [type = any ] ;
        CARD N:N; TRIGGER ...
        ATTRIBUTES
                status = invalid, valid := invalid ;
END link ;

DEFRELATION copied ;
        DOMAIN [type = WE ] -> [type = any ] ;
        CARD N:N; TRIGGER ...
        ATTRIBUTES
                state = exp, tested, released, official := exp ;
END copied ;
```

These descriptions mean that the relations are defined between a WS and any object. `CARD` denotes cardinality, the key-word `TRIGGER` will be explained in the section 3. Adele allows attributes to be associated with relations. In this example attributes 'status' and 'state' express the state and status of the relation.

Adele allows users to define new commands, interpreted by Adele as built-in commands. We defined the command "mkwe" that creates a WE with the name of the WE to create, the type of WE after the "-t" option and the configuration to start after the "-c" option:

```
mkwe mywe -c formatter.cone -t validation_WE
```

This command asks for the creation of a `validation` WE called "mywe" for the configuration "`formatter.conf`".

Now the infrastructure is defined, we will see how the Adele activity mechanism allows definition of a Work Environment policy.

3 Activity management

Software DBMS manages a large amount of dynamic shared data and requires assistance when managing crucial situations. For instance when a module interface is modified, we need to evaluate the impact on modules using this interface, to notify the impacted modules and eventually 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-actions mechanisms. Trigger mechanisms allow definition of actions to be executed automatically when some conditions hold, as for instance checking integrity constraints or propagating changes. Adele-DB provides a formalism to describe events and triggers, and an activity manager that specifies trigger processing and synchronization in DB transactions.

3.1 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 is raised. 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 work environment). These concepts extend the classical trigger mechanism. We describe briefly the extension of the mechanism in Adele and its evolution as an activity manager.

3.2 Trigger description: benefits of the object orientation

The object orientation of Adele offers many advantages in the modeling of trigger concepts [2]. A **trigger** is the (dynamic) association of an event with an action; and is expressed as "ON event DO action". Triggers are associated with object and relation **types**, and thus as object types, they can be aggregated, inherited (refined) and classified.

- Event defined on object types. These events are raised each time 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 for dealing 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 fall in one of the following categories:

Pre actions. Before the execution of an operation on an object of type T, an event is raised 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 to 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 the operation committed, other triggers are executed. These actions allow to modify the database after the command (for instance asserting new states).

Abort action. If the operation fails or aborts, all the 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 behaviors.

3.3 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 the WE.

In order to specify this example, we define first 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 ;
```

Priority indicates 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 on events `Delete_Official` and `valid`.

Line 1 means: 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.

Line 2 means: 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 relations "`link`" have the attribute "`status = valid`".

```
DEFACTION Check_Official ;
      IF [~*|Link|%name%status == valid ] THEN "officialize %name" ;
                                ELSE return;
END Check_Official ;

DEFACTION officialize ;
4       "ada %name -a state = official" ;
END officialize ;
```

A revision can be official only if all the related relationships "Link" to the current object are qualified as "status= valid" (see below line 3 and 4); The Adele language is a powerful metasubstitution language designed for managing multivalued attributes : "$\sim X\%A$" means the values of attribute A of object X, and "$\sim S \mid R \mid D\%A$" means the values of attribute A of relationship

R between objects S (source) and D (destination). All Unix wild card are allowed. If "$\sim X$" or "$\sim S \mid R \mid D$" are omitted, current object or relationship are assumed. Operator "=" means one value is identical, while operator "==" means all the values are identical.

```
TYPERELATION link ;
   TRIGGER -- Propagation on relation link
      POST
         ON replace DO
          "mail -s \"revision to test: %name \" %author";
         ON valid DO
            IF %author = !username THEN "adar -a status = valid" ;
         ON invalid DO
            IF %author = !username THEN "adar -a status = invalid" ;
      AFTER
         ON officialize DO
            "mail -s \"module %name is official\" %author" ;
END Linked_Obj ;
```

Author is an automatic attribute that contains the user name that created the object, and "`!username`" is the name of the current user; "`adar`" is an Adele command that adds an attribute on the current relationship. After a replace, a mail is sent to all the owners of a WE having a logical copy of that object; after the "`validate`" command, attribute "`status=valid`" is set on the relationship that links the WE and the object; after the "`invalidate`" command, attribute "`status=invalid`" is set on the relationship that links the WE and the object.

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 Task manager

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

- The association of behavior with object type on the one hand, the use of relations in the other hand results in a dispersion of the information that makes 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 (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.

For these reasons we are currently experimenting how a rule based tool (the task manager) can cooperate with the activity manager. The task manager is implemented as an external tool, accessing Adele through the RPC programmatic interface. This experiment has already the following results.

First, the activity manager proved to be efficient. It has been immediately heavily used by customers (it is partially released in the commercial product). With a good use of high grained objects (as are WE), high level policies and long transactions are easily managed. The dynamic creation of relationships (Adele has some specific features for that), allows to take into account the dynamic aspects of activities. For instance, the way a C program is managed depends heavily of which configurations it is part of. If in an official configuration, changes are to be performed much more carefully, validation must be deeper etc. This is known by the fact a relation links an official configuration to the C program. It is not directly a property of C programs.

Secondly, we want the Adele kernel to be as little as possible; the high level concepts to provide to users are unclear. It looks clear to us that the kernel must not know high level concepts (for instance WE or coupling are not Adele concepts), so the task manager should provide these concepts along with their Adele representation.

We decided to follow along our tracks: the kernel provide basic and powerful services from which (almost) all the SEE can be easily built, task manager(s) can coexist, providing different concepts and formalisms better adapted to an application domain.

5 Related Work

Different approaches have been proposed for modeling and automatically executing software processes. The main trends are the followings[6].

5.1 Trigger

Triggers are well adapted to DB management. It is the only approach able to cope with versions and configuration, a very fundamental aspect of Software Engineering. It is both efficient and versatile, but too low level for clearly programming high level policies. Adele is one of the very few practical systems based on a trigger mechanism[3]. Appl/A [21] is a system that extends the Ada language to support Software Process Programming (SPPL) that also provides programmable relations, propagation control and some transaction constructs. Other research prototypes are Alf [16], CML [19].

5.2 Graph processing

A graph is traversed and the actions associated with its nodes are executed. Depending on the action performed on a node, the event is "fired" along the edges starting from the current node. In Adele, trigger associated with relation types are typically in this case. **Make**, **Odin** [5] and **Build**

[23] are some systems that use this approach, but dedicated to special task (rebuilding) and with some embedded policies.

5.3 Contractual approach

The *contractual approach* is a way to model software process used only by the **ISTAR** project [9]. The principal idea of this approach is to see every software development activity as a *contract* between a *contractor* and a *client*. The role of **ISTAR** is limited to control contract protocol between contractors and clients, because it does not know enough about the nature of contracts. The practical use of this approach is unclear. To some extend, an Adele WE can be assimilated to a contract : the contract is the WE purpose, the WE user is the contractor; sub contracts are sub WEs.

5.4 Process programming approach

Process programming is an approach to software process modeling proposed by Osterweil [18]. In this approach, the complete software process is defined as a meta-program. It is described by means of a formal language, which is written by the environment administrator before the activation of processes. This description is considered as a specification of how a development process is to be conducted. This ambitious approach has not produced practical results so far. **Arcadia** [22] and **Oikos** [1, 15] are examples of this approach.

5.5 Rule-based approach

In this approach, the knowledge about the activities and tasks of a generic software development process is explicitly modeled by production rules. The tools are integrated to the environment through the utilization of pre and postconditions over their inputs and outputs. The rules may be different depending on the implementation chosen by the system (backward and/or forward reasoning, static or dynamic planning, hierarchic and sequential/parallel planning). This approach is mainly used for high level tasks. This approach is employed by **Marvel** [13], **Grapple** [12], **Agora PM** [20] **Epos** [7, 14], **Merlin** [10] and **Alf** [17]. The Adele Task manager pertains to this class of systems.

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 the DB housekeeping (consistency, extensibility, customization, ..) and simple policies. The second layer (the task manager) uses a rule based strategy and is dedicated to high 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 now trying to define a formalism that will generate code for both levels simultaneously. It will become the user interface to Adele Process programming. We expect, that way, both good efficiency and high level control.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. In *4th ACM SIGSOFT Symposium on Software Development Environments*, Irvine, CA, December 3–5 1990. SIGSOFT Software Engineering Notes, 15(6):183–192.

[2] N. Belkhatir and J. Estublier. Software management constraints and action triggering in Adele program database. In *1st European Software Engineering Conference*, pages 47–57, Strasbourg, France, Sept. 1987.

[3] N. Belkhatir, J. Estublier, M. A. Nacer, and W. L. Melo. Activity coordination in Adele: a software production kernel. In *7th International Software Process Workshop*, San Francisco, CA, October 16–18 1991.

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

[5] G. M. Clemm and L. Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–15, January 1990.

[6] R. Conradi and C. Liu. Process modeling paradigms: an evaluation. In *First European Workshop on Software Process Modeling*, pages 39–52, Milan, Italy, May 30–31 1991.

[7] R. Conradi, E. Osjord, P.H. Westby, and C. Liu-Beijing. Software process modelling in EPOS: design and initial implementation. In *3rd International Workshop on Software Engineering and its Applications*, pages 365–381, Toulouse, France, December 3–7 1990.

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

[9] M. Dowson. ISTAR and the contractual approach. In *9th International Conference on Software Engineering*, pages 287–288, Monterey, CA, March 30-April 2 1987.

[10] W. Emmerich, G. Junkermann, B. Peuschel, W. Shafer, and S. Wolf. Merlin: knowledge-based process modeling. In *First European Workshop on Software Process Modeling*, pages 181–186, Milan, Italy, May 30–31 1991.

[11] J. Estublier, S. Ghoul, and S. Krakowiak. Premilinary experience with a configuration control system for modular programs. In *ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Software Practical Development Environments*, Pittsburgh, April 23–25 1984. In *Software Engineering Notes, 9(3):149–156, May 1984.*

[12] K. E. Huff and V. R. Lesser. A plan-based intelligent assistant that supports the software development process. In *3rd ACM Symposium on Software Development Environments*, Boston, Massachusetts, November 28–30 1988. In *ACM SIGPLAN Notices*, 24(2):97–106, February 1989.

[13] 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.

[14] C. Liu. A software process planner in Epos. In *Norsk Informatikk Konferanse 1990*, pages 203–215, Bergen, Norway, November 1990.

[15] V. Ambriola C. Montangero. Hierarchical specification of software processes. In *First European Workshop on Software Process Modeling*, pages 139–145, Milan, Italy, May 30–31 1991.

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

[17] F. Oquendo, J.D. Zucker, and P. Griffiths. The Masp approach to software process description, instantiation and enaction. In *First European Workshop on Software Process Modeling*, pages 147–155, Milan, Italy, May 30–31 1991.

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

[19] J. Ramanathan and S. Sarkar. Providing customized assistance for software lifecycle approaches. *IEEE Transactions on Software Engineering*, 14(6):749–757, June 1988.

[20] R.Bisiani, F. Lecouat, and V. Ambriola. A tool to coordinate tools. *IEEE Software*, pages 17–25, November 1988.

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

[22] R. N. Taylor and *et al.* Foundations for the Arcadia environment architecture. In *3rd ACM Symposium on Software Development Environments*, Boston, Massachusetts, November 28–30 1988. In *ACM SIGPLAN Notices*, 24(2):1–13, February 1989.

[23] R. C. Waters. Automated software management based on structural models. *Software–Practice and Experience*, 19(10):931–955, October 1989.