

Cooperative Work in Large–Scale Software Systems ^{*}

Noureddine Belkhatir

Jacky Estublier

Walcélio L. Melo [†]

Laboratoire de Génie Informatique
BP 53X, 38041 Grenoble, FRANCE
{belkhatir, estublier, wmelo}@imag.imag.fr

Software maintenance is responsible for approximately 80% of the total cost of software. In order to reduce maintenance costs by improving both software productivity and quality, several Software Engineering Environments have been built. In this paper, we present the main characteristics of the Adele system, showing how this system is able to take account of software maintenance problems. We concentrate on the ability of this system to accept the specification of project specific policies and how these policies can be enforced automatically. We illustrate this with examples which deal with software maintenance. The second example is presented in the annex and shows an example of cooperative work in an aerospace company. We conclude this article by presenting the lessons learned using our approach and our projects for the future.

Key words: CASE; software engineering environment; maintenance; software process; programming-in-the-large.

1 Introduction

An accepted definition is that software maintenance is used to keep a software system operational and responsive after it has been accepted and put into production. Maintenance characteristics, in a large software system are the following:

- Maintenance is centered around communication and coordination. Studies have proved that 2/3 of time is spent on coordination and communication, and only 1/3 on development itself. Thus, quality and productivity can essentially be improved by a better support of cooperative work.

^{*}To appear in the *Journal of Software Maintenance: Research and Practice*, 1993

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

- Maintenance is no longer considered as a single step at the end of the life cycle (e.g. waterfall life cycle model). Evolutive Maintenance involves and subsumes all the phases identified during development (requirement, design, etc.). Recent software life cycle models (e.g. spiral life cycle model) propose approaches which consider maintenance as an activity present in all stages of software life.
- Change is intrinsic in software maintenance. Because of the long working life of software, almost anything can change: software design and functionalities, of course, but also the tools used, the people in charge of the work, the maintenance methodology, and even the company.

Various software engineering environments (SEEs) have been built in order to provide a solution to software maintenance. These environments fall into two categories:

1. the toolkit approach.

In the toolkit approach, a set of independent tools are put together in order to support the project life-cycle activities. There is no appropriate mechanism for integrating and coordinating them. However, almost no maintenance specific tools are available.

2. monolithic tool.

In this approach, all functionalities are supposed to be provided by a single, large tool. In such systems some predefined (and a priori) policies are used to control the use of system capabilities, coordinate team work, enforce team communication, e.g., Dsee[Leblang and Chase, 1988], Infuse[Perry and Kaiser, 1987], and Nse[Miller, 1989]. Such SEEs do not provide sufficient facilities (if any) for tool integration and customization; they are quite inflexible because the policies supported by the system cannot be tailored, and functionalities cannot be extended. Furthermore, because of their complexity, these tools address a reduced portion of the Software life cycle.

Software Maintenance consumes considerable effort and is not sufficiently assisted. At least three aspects of current SEEs are not satisfactory:

1. they do not have appropriate mechanisms for tool integration and customization,
2. they do not provide facilities for team support,
3. they do not provide sufficient facilities and flexibility for evolution.

In this paper we shall concentrate on these three aspects.

2 The Adele approach

The inflexibility of the current approaches has been recognized [Madhavji, 1991, Osterweil, 1987, Zucker, 1991]. In order to improve power and flexibility, progress must be made in two directions: Data Modeling (DM) and Process Modeling (PM). Data Modeling, driven by the DBMS evolution towards CASE, tries to propose a Data Model that

fits Software maintenance needs. Process Modeling, on the other hand tries to model the Software life cycle (manage methods, tools, users and activities).

The Adele system tries to integrate both aspects, the static aspects (data and product modeling) and the dynamic aspects (process modeling), thus producing a Process-Oriented SEE (POSE)[Belkhatir et al., 1991, Belkhatir et al., 1992]. In a POSE, activities are driven by an explicit description of the processes under way. Nowadays, the software community agrees about the need for a POSE and research for an appropriate, general and interpretable formalism for modeling such aspects is active [Balzer, 1991, Conradi et al., 1991, Deiters and Gruhn, 1990, Heimbigner, 1989, Katayama, 1989, Madhavji and Schafer, 1991, Sutton et al., 1990].

2.1 The Adele Data Model

Strict requirements are set on the underlying Data Model, for support maintenance, such as file, version, complex object, propagation and long transaction management. These features are not described here.

Since the Adele Data Model is based on the generalization of the aggregate concept, we only concentrate on this concept. An aggregate is an object that contains other objects organized in a particular way. An aggregate may be homogeneous, containing only objects of the same type (or sub-types), or heterogeneous. Composite objects, sets, lists, class extents are different kind of aggregates.

2.1.1 Aggregate structure

In Adele, an aggregate is an object related to the objects it contains by relationships of a given type; aggregate characteristics are defined inside these relationship types. For example, the component type is defined in the destination relationship type; if the relationship cardinality is multiple, the components can be shared.

As a result, aggregates are built incrementally. To add a component to an aggregate, a relationship must be created between the aggregate and the component.

In order to define a new type of aggregate, it is sufficient to define a new type of relationship and if necessary a new relationship source object type. The composition of an aggregate can evolve over time just as the relationship definition evolves.

2.1.2 Aggregate behavior

It is essential to ensure consistent behavior for each operation defined for a composite object or its content. An aggregate may

- delegate operations it is supposed to execute; for example, “copy” on a composite object produces a “copy” on its components (on a heterogeneous composite object; a record),
- iterate on each component (on a homogeneous set),
- select one of the components (in a type generalization/refinement aggregate): only one copy operation is to be selected for a given instance).

Furthermore, consistency constraints can be fixed between the aggregate and its content, for example “it is forbidden to delete the content of a given kind of aggregate”. Clearly, it is the expression of this semantics which defines aggregate behavior.

Once again the dynamic behavior of an aggregate is defined in the relationship type. In Adele, relationships play a key role; they are used to model:

- aggregate structure (hierarchy, DAG, etc.) and content.
- object lifecycle by providing version history (modeled by `revision_of` and `variant_of` relationships)
- the file under test in a given work environment.

Thus the relationship models the context in which an object is used. In Adele, all relationships define an implicit aggregate (a configuration is an aggregate as well as a Working Environment or a version graph). We believe the main difference between the first design and maintenance is the fact that objects depend to a large extent on their context. This is why aggregate management, which models the context, plays such an important role in our model.

In view of the fact that all the aggregate semantics are defined in the relationship, we decided to draw a sharp boundary between the following features:

1. context free object behavior (i.e. whether or not the object is part of an aggregate(s)). This is encapsulated in the object type, and uses the classical O.O. paradigm.
2. context dependent object behavior (i.e. the behavior of an object as part of an aggregate(s) which models the context of the object). This is encapsulated in the relationship type defining that aggregate. Relationships are also structured into an OO (multiple) inheritance hierarchy.

We believe that this distinction is fundamental. In this way, a complex system can evolve only as a side effect of the creation and destruction of relationships, and it will always remain consistent. This is how a system can be scaled up. Since a relationship is established between two objects, the behavior of both objects is automatically modified, without any need to define new sub-types, to change type on objects, even if the association was not foreseen. It is also easy to model new context behavior by creating new relationships and changing relationship definitions.

As an example, let us define a “**ContRel**” relationship type for the definition of complex objects:

```

TYPERELATION ContRel;
    ON SOURCE copy DO ‘‘copy !D’’;
    ON SOURCE delete DO ‘‘delete !D’’;
END ContRel;

```

```

TYPERELATION Protected;
    ON DEST delete DO ABORT;
END Protected;

```

This simple (and incomplete) example shows how an operation (copy and delete) on the source of a containment relationship instance (i.e. the aggregate) is delegated to the aggregate content (“!O, !D” being the name of the object Origin and Destination of the relationship). The “Protected” relationship type specifies that the content must never be deleted. The expression `~obj%A` is substituted by the list of values of attribute A of object obj .

```

TYPERELATION ConfContent IS Protected, ContRel;
    DOMAIN [type = configuration] -> [type = prog];
    ON DEST modified DO
        IF [ ~!O%state = released] THEN ABORT;
END;

```

The `ConfContent` relationship is defined between a configuration and its programs or documents (types `prog` and `doc`; configuration is itself a subtype of `prog`). This relationship is a protected containment relationship, extended by a constraint: the content must not be changed (event `modified`) if the configuration (!O) is in `release` state. Constraints are not overloaded; they are all executed, starting with the ones which are local to the type and then in the order in which the super-types appear in the definition.

2.1.3 Hard aggregates

In Adele, aggregates can range from very loose (relationship with no semantics) to pretty tight aggregates, as with composite objects. We found it necessary to use very tight aggregates where component cohesion is very high.

We call aggregates “hard” when all characteristics of the container instance are “inherited” by the contained object instances (and recursively). It is an example of instance inheritance, as opposed to classic type inheritance. Hard aggregates provide a convenient and efficient way of defining and managing all objects which share a significant proportion of their characteristics.

Here, characteristics include attribute values, relationship instances, constraints, methods and access rights. In addition, content cannot be shared.

The hard aggregate concept has proved very useful for implementing versioning. An object is (implicitly) a `version_set` and is implemented as a hard aggregate containing all the individual versions. It is thus possible to define which characteristics are to be versioned and which are not; those not versioned are common to all components and thus will be associated with the

version_set (i.e. the container), those specific to each individual versions will be associated to that version (i.e. the content).

Since Adele versioning uses Hard Aggregates, for efficiency reasons the `has_type` relationship used to build hard aggregates is predefined. Thus additional properties are provided, like a dot notation, to reach the content from the container with improved management of access rights and views (not presented here).

Example: Imagine a chest; it may contain other chests. If you are entitled to use a chest you have the implicit right to use its content. If it is a top secret chest, its content is also top secret and so on.

2.2 The Process model

In order to introduce some degree of discipline into the process, policies must be formalized and enforced to control, monitor and assist teams when performing their activities. Several prototype systems have been built to support these dynamic aspects, but no consensus has been reached on the paradigms and approaches to use. It seems that no single approach can solve all the problems. The current paradigms are the *Rule-based* paradigm, where the knowledge about activities and tasks of a generic software development process is explicitly modeled by rules as in Marvel[Kaiser et al., 1990] and ALF[Zucker, 1991]; the process programming paradigm, where an imperative language is used to describe the software development process as in Arcadia[Taylor and *et al*, 1988] or Triad[Sarkar and Venugopal, 1991]; and the *active data base* paradigm, where the underlying data base uses triggers.

Adele triggers take the following form:

```
ON event DO Action ;
```

Where “event” is a predicate over the system state, object state and the current activities (query, navigation as well as changes) occurring on objects.

```
EVENT delete = [ !command = remove ] ; PRIORITY 5 ;
```

An Action is a program in the Adele Language. An Adele language instruction can be a logical expression, an Adele command or a Unix command. This language is a simple imperative language, tailored to access the Adele database and to permit easy navigation through arbitrary relations. It is a meta-substitution language (late binding of parameters and variables) that resembles the Unix shell, except that variables are multi-valued attributes, with provision for complex query and operator sets.

Triggers, as defined, are a very basic mechanism, useful mainly for maintaining consistency constraints. This mechanism becomes very powerful when integrated into both the data model and the recovery mechanism.

Triggers are defined in the type definition of objects and relations. They are structured along the type hierarchy, thus they are inherited. A trigger defined for an object type is executed

if the event is true for an instance of that type or an instance of any sub-type. Triggers cannot be overloaded, they are mandatorily inherited by sub-types.

For a relationship of type R, there is a different case:

Commands can act on a relationship itself, as for example `delete_relation` or `add_relation_attribute`;

```
ON Delete_relation DO {A1};
```

means: If the event `DeleteRel` becomes true for a relationship instance of type R, execute A1.

```
ON ORIGIN delete DO {A2};
```

means: If the event `delete` becomes true for the origin object of a relationship instance of type R, execute A2.

```
ON DEST delete DO {A3};
```

means: If the event `delete` becomes true for the destination object of a relationship instance of type R, execute A3.

2.2.1 Triggers and transactions

In Adele, users can define methods (i.e. actions associated with a given object type), as well as commands (i.e. actions not related to any particular type).

Triggers are used to check the consistency of such methods/commands. Some triggers will be executed before the action, acting as pre conditions, others after the action, as post-conditions. Since triggers are (originally) intended to enforce consistency, any inconsistency found by a trigger must be able to undo (roll-back) the action. Thus for any action the following instructions will be executed:

```
PRE list of triggers
    Action (Command or Method)
POST list of triggers
```

the whole execution is always a single transaction, even if the triggers or the action call other actions. The execution of a primitive `ABORT`, anywhere in a block (`PRE/Action/POST`) will undo everything that was done in this block. After transaction validation; `AFTER` triggers are executed; they are used to execute actions when sure that the transaction succeeded, as for example sending notifications or to execute new actions whose failure must not undo the main action.

If the transaction failed, `ERROR` triggers are executed.

Thus for each object and relation type, there are five blocks:

```

PRE list of triggers
METHOD list of methods
POST list of triggers
AFTER list of triggers
ERROR list of triggers

```

3 An example: coordinating changes in large software systems

Using an example, we show the potential of Adele for coordinating changes in a large development environment. Our point of departure is a modification request, approved by the Configuration Control Board, and an initial configuration. A team and the appropriate organizational structure are set up to implement the changes.

In this example, the development areas are organized into a tree structure in which level i controls the developments carried out on the level immediately below ($i + 1$). The development area level 0 integrates the entire software product. Intermediate development areas maintain partial configurations while module-level modifications are made in the development area leaves.

We suppose that a specialized tool like [Maarek and Kaiser, 1988, Selby and Basili, 1988, Shwanke and Platoff, 1989] was used to generate the tree from the given configuration.

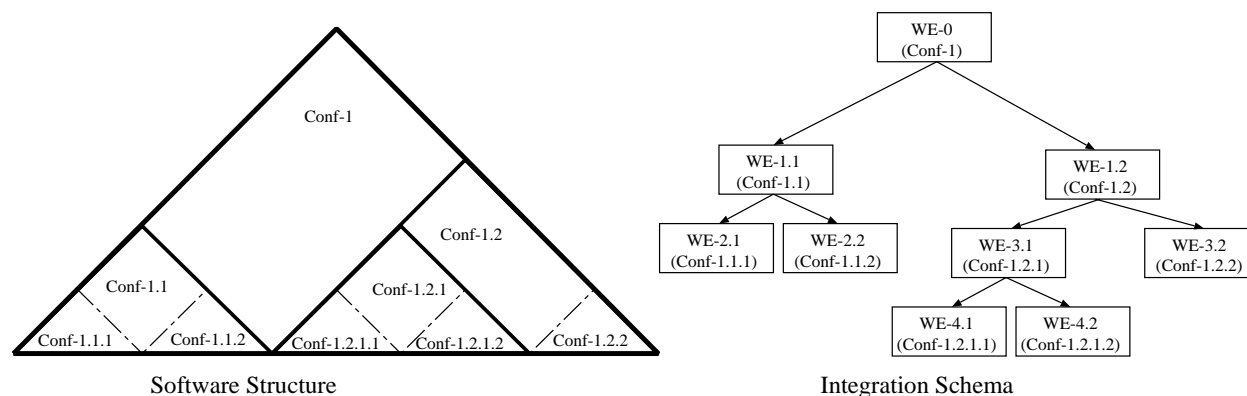


Figure 1: the product and Work Environment organization

In this example we concentrate on the integration of various parts of the software product. No details on the modification process, within a development area, will be given, but we show how local modifications are integrated into the higher and lower levels.

A WE is valid if all its sub-WEs are valid, and if the test applied to it are successful. A validation request for the current development area is implemented by the `mvr` command (Modification Validation Request).

Two policies can be implemented. In the top-down policy, a `mvr` request on a WE is automatically propagated down to the sub-WEs to try to validate them; in the bottom-up approach, as soon as a WE is validated (a successful `mvr` request), we try to validate the father WE, and recursively. In the following solution both are implemented.

This strategy will be implemented by defining the `mvr` command, and triggers on the WE objects and on the `sub-we` relation, which links a WE to its sub WEs.

```

DEFEVENT valid-attempt = [!cmd = mvr];

DEFACTION mvr; -- the MVR command executes the test on the WE;
  IF [ %state = tested ] THEN RETURN ;
  IF 'test %name' THEN "mda %name -a state = tested"
  ELSE ABORT;
END mvr;

TYPEOBJECT WE;
  ERROR ON valid-attempt DO "mda %name -a state = untested";
END WE;

TYPERELATION sub-we;
(1) PRE ORIGIN valid-attempt DO "mvr !D";
(2) AFTER DEST valid-attempt DO "mvr !0";
END sub-we;

```

The `mvr` command only execute a test program `test` on the current WE and sets the state to `valid` if it is successful. If not successful, the `mvr` command is aborted, which produces the `ON ERROR valid-attempt` trigger to be executed: the WE state is turned to `untested`.

The `sub-we` type expresses that

1. before executing the `mvr` command `PRE ORIGIN mvr`, if a sub-we is not in a valid state, `~!D%state != tested`, we try first to validate it `mvr !D`;
2. if a WE has been validated `AFTER DEST valid-attempt`, (in case of failure, the `ERROR Triggers` are executed, not the `AFTER Triggers`) we try to validate the father WE by applying `mvr !0`.

4 Conclusion

Adele is a kernel designed to facilitate the building of Software Engineering Environments. It was found that the key requirement is the ability to describe and enforce specific software policies. To meet this requirement, we adapted concepts borrowed from Object-Orientation and semantic models.

We extended the E-R-A (Entity Relation Attribute) models, with a third entity: **procedures**. Procedures assume many of the roles devoted to process management such as tool

invocation and integration. Each one of these three entities is modeled independently.

We extended Object-Oriented concepts by promoting relationships to the status of first class citizens. Object method and triggers model object behavior, as an isolated entity, while relation method and triggers model object behavior as part of a complex highly interrelated system. Behavior (for both objects and relationships) is modeled using event-condition-action (ECA) rules and interpreted by a trigger mechanism.

We extended the database concept, integrating triggers with the traditional concept of transaction. The tuple (Pre Rules, Procedure, Post rules) is a single transaction, while the tuple (After rules, Error rules) is executed outside the original transaction. Our goal in adding triggers is to develop active Database Management Systems (DBMS) for Software Development Environments. In Adele, the constraints are not a simple boolean predicate but can include complex programs and be executed outside the database. Triggers are used to control state transitions as well as to model process control.

Adele provides several other features to support cooperative work, such as version control and configuration management, RPC and graphical interface, logical copying, merging, etc.

This system was tested for one year. Two kinds of experiments were conducted: local prototypes, such as the implementation of current systems such as NSE, DSEE, PALAS on top of Adele; and industrial experiments, as for instance the HERMES (the European shuttle) software environment where thousands of triggers were written by the Hermes team to automate and simplify the development of their specific CASE. From these early experiments we can extract the following lessons.

- The Adele language is low level. It manages essentially attributes and its meta-substitution syntax is not always easy to understand. Since the Adele kernel does not include predefined high level concepts such as work environment, user policy, synchronization, deadline, the level of the concept managed by the language is not high enough. However this was intentional, the Adele kernel is designed to support “any” high level concept of WE, Task, Team, User, Planning, Policy, etc.
- The fragmentation of information into different object types and relation types makes it difficult to have a clear picture of a complete process. As we can see in the section3, a single policy is often described using an object type, a relationship type and a command simultaneously. For human understanding, it would be preferable for it all to be unified in a special entity.
- High level reasoning, learning explanation is almost impossible with a trigger description, because of its low level abstraction and fragmentation. As pointed out in [Legait et al., 1989, Sutton et al., 1990], a software process manager must be able to answer, among others, the following questions: which are my tasks in the process? which resources are available to my task? what should I do now, how can I delegate my task to a friend? etc. User friendliness, including a graphical interface, as found in Archipel Agenda [Fernstrom and Ohlsson, 1991] and Galois[Sugiyama and Horowitz, 1991], is felt to be important.
- The control of triggers sometimes proved tricky. There are risks of event explosion,

loops or duplicate actions. Clearly we need support for trigger programming and debugging, as well as a programming methodology.

We are currently trying to solve the problems we found during our experiments. Two directions have been followed:

- The design of languages supporting high level concepts, such as Task, Team, Schedule, Work Environment, Synchronization, etc. These languages are partially compiled in terms of Adele Triggers. The purpose of such languages is:
 - To allow more natural definition and control of SEE,
 - Using its high level concepts, to allow reasoning and explaining,
 - To avoid fragmentation.
- The developing of a set of tools to support event programming: validation and vision based on petri nets, debuggers, etc.

These extensions, fortunately, use the Adele kernel as it is. Multiple experiments will not interfere with kernel stability and efficiency. On the other hand the Adele kernel itself is extended: Trigger extensions such as direct calls to external (C) procedures, integration of a Broadcast Message Server, definition of invariant and variable triggers, and Data Model extensions, including the development of a PCTE interface.

We expect this work to become an industrial framework in which evolving (Process Oriented) Software Engineering Environments can be easily and rapidly built, customized and tuned.

References

- [Balzer, 1991] Balzer, R. (1991). Process virtual machine. In *Proc. of the 7th International Software Process Workshop*, San Francisco, CA.
- [Belkhatir et al., 1991] Belkhatir, N., Estublier, J., and Melo, W. L. (1991). Adele 2: a support to large software development process. In Downson, M., editor, *Proc. of the First International Conference on the Software Process*, pages 159–170, Redondo Beach, CA. IEEE Computer Society Press.
- [Belkhatir et al., 1992] Belkhatir, N., Melo, W. L., Estublier, J., and Nacer, M. A. (1992). Supporting software maintenance evolution processes in the Adele system. In Reeves, D. S., editor, *Proc. of the 30th Annual ACM Southeast Conference*, Raleigh, NC.
- [Conradi et al., 1991] Conradi, R., Osjord, E., Westby, P., and Liu, C. (1991). Initial software process management in Epos. *IEE Software Engineering Journal*, 6(5):275–284.
- [Deiters and Gruhn, 1990] Deiters, W. and Gruhn, V. (1990). Managing software processes in the environment MELMAC. In *Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments*, Irvine, CA. SIGSOFT Software Engineering Notes, 15(6):193–205.

- [Fernstrom and Ohlsson, 1991] Fernstrom, C. and Ohlsson, L. (1991). Integration needs in process enacted environments. In Downson, M., editor, *Proc. of the First International Conference on the Software Process*, pages 142–158, Redondo Beach, CA. IEEE Computer Society Press.
- [Heimbigner, 1989] Heimbigner, D. (1989). P4: a logic language for process programming. In *Proc. of the 5th International Software Process Workshop*, Kennebunkport, Maine.
- [Kaiser et al., 1990] Kaiser, G. E., Barghouti, N. S., and Sokolsky, M. H. (1990). 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.
- [Katayama, 1989] Katayama, T. (1989). A hierarchical and functional software process description and its enactment. In *Proc. of the 11th International Conference on Software Engineering*, pages 343–352, Pittsburgh, Pennsylvania.
- [Leblang and Chase, 1988] Leblang, D. and Chase, R. P. (1988). Parallel building: experience with a case for workstations networks. In *International Workshop on Software Version and Configuration Control*, Grassau, FRG.
- [Legait et al., 1989] Legait, A., Menes, M., Oquendo, F., Griffiths, P., and Oldfield, D. (1989). ALF: its process model and its implementation on PCTE. In Bennett, K. H., editor, *Software Engineering Environments — Research and Practice*, pages 335–350. Ellis Horwood Books. *4th Conference on Software Engineering Environments*, Durham, UK, April 11-14, 1989.
- [Maarek and Kaiser, 1988] Maarek, Y. and Kaiser, G. E. (1988). Change management in very large software systems. In *Phoenix Conference on Computer Systems and Communications*, pages 280–285. IEEE computer Society Press.
- [Madhavji and Schafer, 1991] Madhavji, N. and Schafer, W. (1991). Prism — methodology and process-oriented environment. *IEEE Transactions on Software Engineering*, 17(12):1270–1283.
- [Madhavji, 1991] Madhavji, N. H. (1991). The process cycle. *IEE Software Engineering Journal*, 6(5):234–242.
- [Miller, 1989] Miller, T. (1989). Configuration management with the NSE. In Long, F., editor, *International Workshop on Software Engineering Environments*, pages 99–106. Springer-Verlag, Chinon, France. Lecture Notes in Computer Science, vol. 467.
- [Osterweil, 1987] Osterweil, L. J. (1987). Software processes are software too. In *9th International Conference on Software Engineering*, Monterey, CA.
- [Perry and Kaiser, 1987] Perry, D. E. and Kaiser, G. E. (1987). Infuse: a tool for automatically managing and coordinating source changes in large systems. In *Proc. of the ACM Computer Science Conference*, pages 17–19, St. Louis, Missouri.
- [Sarkar and Venugopal, 1991] Sarkar, S. and Venugopal, V. (1991). A language-based approach to building CSCW systems. In *24th Annual Hawaii International Conference on*

System Sciences, pages 553–567, Kona, HI. IEEE Computer Society, Software Track, v. II.

- [Selby and Basili, 1988] Selby, R. W. and Basili, V. R. (1988). Error localization during software maintenance: generating hierarchical system descriptions from the source code alone. In *Conference on Software Maintenance*. IEEE Computer Society.
- [Shwanke and Platoff, 1989] Shwanke, R. W. and Platoff, M. A. (1989). Cross references are features. In *2nd International Workshop on Software Configuration Management. ACM SIGSOFT Software Engineering Notes*, 17(7):86–95, November 1989.
- [Sugiyama and Horowitz, 1991] Sugiyama, Y. and Horowitz, E. (1991). Building your own software development environment. *IEE Software Engineering Journal*, 6(5):317–331.
- [Sutton et al., 1990] Sutton, S. M., Heimbigner, D., and Osterweil, L. J. (1990). Language constructs for managing change in process-centered environments. In *Proc. of the 4th ACM Symposium on Software Development Environments*, Irvine, CA. In *ACM Software Engineering Notes*, 15(6):206–217, December 1990.
- [Taylor and et al, 1988] Taylor, R. N. and et al (1988). Foundations for the Arcadia environment architecture. In *Proc. of the 3rd ACM Symposium on Software Development Environments*, Boston, Massachusetts. In *ACM SIGPLAN Notices*, 24(2):1–13, February 1989.
- [Zucker, 1991] Zucker, J.-D. (1991). ALF: accueil de logiciel futur. In Long, F., editor, *Software Engineering Environments - volume 3*, pages 21–52. Ellis Horwood Books. *5th Conference on Software Engineering Environments*, Aberystwyth, UK, March 25–27, 1991.

5 Annex 1: an example of cooperative work

5.1 The problem

Currently, cooperative work is based on only two paradigms:

1. Sequential work. A step is performed after completion of the previous one.
2. Parallel work. Work is performed independently in parallel. The next task waits for completion of all parallel tasks.

Imagine the following situation. After a requirement analysis (not taken into account), a hardware must be created. This piece of equipment is built from a processor board (hardware artifact), and software supported by that board. The design of both elements is parallel, but since design decisions can produce constraints on the physical characteristics of the processor, and software design decisions can produce constraints on hardware characteristics, it would be useful to have cooperation between these two manufacturing steps. In this example, both sequencing and independent parallelism will produce conflicts, found later, involving major redesign.

Cooperation consensus is envisioned, where each time a designer (either hardware or software) takes a major design decision, the other interested designers are automatically notified and receive an explanation of the proposed design decision. From that point communication is established between all designers, potentially impacted by the change, and the designer who proposed the change. The impacted designers can propose modifications to the proposed design decision, or complain. Communication must be partially automated (a document, visible by every designers participant in the process, must contain subsequent comments and design modifications, each new comment is automatically added to this document), but informal communication is not prohibited (better encouraged).

This communication may take some time but must end in an agreement between the designer who took the decision and all designers impacted by that decision. An agreement must be found within a predefined period of time. After this period of time (the deadline) the team leader is notified, and must arbitrate the conflict; he will automatically receive all the correspondence between designers. In the mean time, designers are free to do what they want, and in particular to assume the decision will be accepted (the most frequent case) and to follow the design.

We assume that a board can support many different software solutions, and that a specific software can be supported by different processor boards. It is also assumed that a designer may not know who is interested in his work, but conversely must know which design work may impact his own work.

5.2 The Adele solution

We assume that an arbitrary number of Work Environments (WE) are working in parallel. To each WE is associated a user; each WE may subscribe to the Work Environments that can produce Design Decisions (DD) of interest. Conversely a WE is not assumed to know which WE subscribed to it.

When a designer takes an important Design Decision (DD), whatever its WE type, he writes the text explaining the decision, and proposes the DD; that DD must become visible to all subscribers, regardless of subscriber WE type and location (they can be distributed over the network).

Once a DD is proposed:

- All subscribers are notified by mail. The notification indicates which document contains the design proposition and the deadline.
- All subscribers will have a copy of the proposed design decision.
- The design decision will have the “proposed” state.
- The design could be in “accepted” state only if all subscribers have accepted that design decision.
- No other design decision from WE_i will be accepted.

- If the design decision is not accepted before the specified deadline, the team leader is notified and will be responsible for solving the conflict.

Some hints on the Adele command language are needed. A major facility in this language is meta-substitution, allowing late-binding of parameters. Like other traditional languages, the Adele command language has some iteration constructors (IF THEN ELSE, WHILE, FOR, etc.).

Meta-substitution is not easy to understand. Meta-substitution always returns the values of attributes. Since attributes may be multi-evaluated, a substitution may end in a list of values. Expressions are always of the form `~objectname%A` or `~objectname!A`; the result is the set of values of attribute A of object `objectname`. `%A` is a user defined attribute, while `!A` is a system defined attribute. `Objectname` may be either an object name or a relationship name in the form `(O|R|D)` with O for the origin object of the relationship, R the type of relationship and D the destination of the relationship.

5.2.1 The command `synchro`

This command defines synchronization between the current WE and another WEi.

```
DEFACTION Synchro LEVEL 8; NNAME NCHECK
    IF "mkr !userdir.!ctxtname -r synchro -d !optvald" THEN {
        "enct !optvald " ;
        "mkln !optvald " ;}
END Synchro ;
```

This command will instantiate a `Synchro` relationship between the current WE and the WE target of the synchronization. The target object of the synchronization will be logically copied in the current WE. `!userdir` is substituted by the user directory associated with the current WE, `!ctxtname` is the name of current context.

Example:

```
Synchro SoftDes -d HardDes
```

By applying this command, the `SoftDes` WE will be synchronized with the `HardDes` WE. That is, a `synchro` relationship is instantiated between `SoftDes` WE and `HardDes` WE, and `SoftDes` WE is enlarged with the design document of the processor board carried out by the `HardDes` WE.

5.2.2 The propose command

This command is used to propose a new design decision. The user specifies which is the proposed design and, optionally, the deadline. For example `propose SoftDes -d "now + 1 month"`

Below we shown the definition of this command:

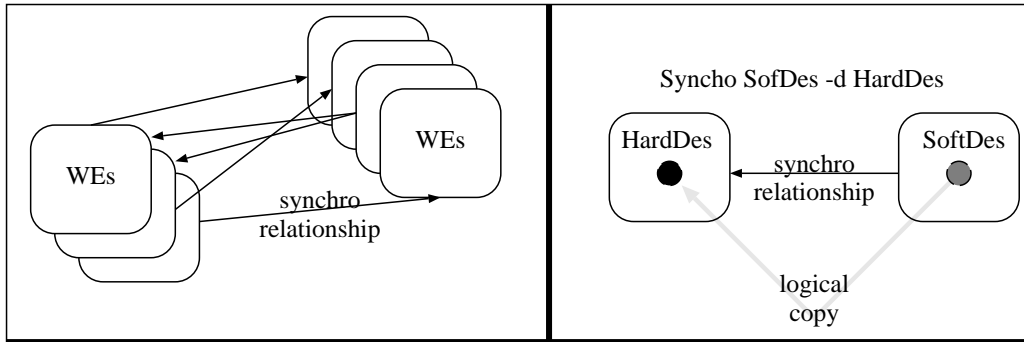


Figure 2: An application example of the Synchro command

```

DEFACTION propose LEVEL 3 ; NNAME
1      "rpe (!userdir.!ctxtname) -f (!ctxtname.des) -res";
2      "mka ~(!userdir.!ctxtname)!revname -a  status proposed
        deadline !optvald" ;
3      "at ~(~(!userdir.!ctxtname)!revname)%deadline <<+
deadline \" !userdir.!ctxtname \"
+\" ; } ;
END propose ;

```

This piece of program stipulates that:

1. the decision is recorded as a new revision of the WE Design Decision Document.
2. attributes `state = proposed`, and `deadline` are associated with the Design Decision Document.
3. using the Unix command `at`, the command `deadline` will be executed when the deadline is reached. The integration of Unix tools with Adele is simple; we can call any tool from Adele in the same way as an Adele command. Integration with the HP Broadcast Message Server is under way for improving tool communication.

Using the `synchro` relation type, we can define the policies corresponding with the `propose` command. Below, we show part of the definitions that are described in this relation.

```

TYPERELATION synchro ;
PRE DEST proposed DO
1      IF [ ~(~!destname%!revname-1)%status = proposed ] THEN ABORT ;
2      IF [ !username != ~(~!destname%!revname)%author ] THEN ABORT ;
POST DEST proposed DO
3      "mail -s \"Design modification\" ~!sourcename%author <<+
        WARNING: a design modification is proposed in ~(!destname)!revname
        Deadline for acceptance is ~(~!destname%!revname)%deadline .
+\" ;
4      "mdar !relship -a status proposed" ;

```

This program text means:

1. If a Design Decision is proposed but the previous proposition is not yet validated (or rejected), the command **propose** is aborted;
2. Only the user responsible for a WE is allowed to propose Design Decisions in this WE.
3. When a change is proposed, a mail is sent to all subscribers (all the values of the author attribute of the relationship source object): `~!sourcename%author;`
4. then **proposed** status is set on the current relationship (`!relship`).

5.2.3 Approve proposition

This command informs that the modification is **approved**.

```
DEFACTION approve LEVEL 3 ;           |           Example:
END approve ;                         |           approve HardDes
```

Note that no instruction is defined in this command (an empty command), however by its application, the trigger defined in the `synchro` relation does the work.

```
TYPERELATION synchro ;
DOMAIN ** -> ** ;
DEFATTRIBUTE
    status = null, proposed, refused, approved := null ;
CARD N:N ;
    DEST approved D0
1      "mdar !sourcename -d !destname -r synchro -a status approved" ;

2      IF [ ~(**|synchro|!destname)%status == approved ] THEN {
2.a    "mda !destname -a status approved" ;
2.b    "mail -s Approved ~!destname%author <<+
        CONGRATULATIONS: Your design ~!destname!revname is accepted.
+"; } ;

    ...
```

This piece of program stipulates that when the design is approved (`DEST approved D0`):

1. `synchro` relationship is turned to approved (`mdar` is the command Modify Attribute of Relation);
2. The expression `(**|synchro|!destname)` is substituted by all relationships ending on the current object; the expression `~(**|synchro|!destname)%status` means the set of attribute `status` of all relationships ending on the current object. `==` is the strict equality operator between two sets: all relationship status must be equal to approved.
 - (a) the design is approved

(b) a mail is sent to the designer.

5.2.4 Deadline

This command is called automatically by Unix system when a deadline is reached.

```
DEFACTION deadline LEVEL 1 ;
1  IF [ %status != approved ] THEN
2    "mail -s \"deadline\" ~%name%chef <<+
    Deadline reached for proposed modification described in %name
+" ;
END deadline ;
```

If the status of that DDD is not approved (1), a mail is sent to the process administrator. Note that in the mean time a new design decision may have been issued, we are looking for the original design decision, which is why we use a revision name.

5.2.5 Refuse

This command is used to refuse a Design Decision. The responsible user must include the reason. When this command is executed, the following actions are undertaken:

```
DEFACTION refuse LEVEL 3 ;
1    "mke !revname.res -f !optvalf -t response" ;
2    "mdar !userdir.!ctxtname -r synchro -d %name -a status refused" ;
END refuse ;
```

1. the reason is recorded in the database, as a file associated with the DD;
2. the status attribute of the `synchro` relationship is set to `refused`.

When a refuse command is issued, the author of the refused DD must be notified. This is performed by the `synchro` relationship:

```
TYPERELOCATION synchro ;
...
POST DEST refused DO
    "mail -s \"Design rejected\" ~!destname%author <<+
    WARNING: Your design modification is rejected by !username.
    See ~!destname%!revname.res please.
+";
```

This piece of program says that when a design is refused (it is always the dest that is refused, a mail is sent to the designer (the author of the destination `~!destname%author`).