4

The Adele Configuration Manager

JACKY ESTUBLIER, RUBBY CASALLAS

This article proposes an overview of the Adele system and also discusses other approaches and other systems. While most of our work on Configuration Management is covered, we have deliberately chosen to discuss in greater depth the Work Space aspect and the relationship between the Repository Space and the WSs.

A Software Configuration Management (SCM) system results from harmonious collaboration between a Repository Space (RS) and Work Spaces (WSs). The WS controls the dynamic aspect of SCM, i.e., the place where activities are performed; the RS is the place wherein software objects are modelled, stored and manipulated.

It is argued here that an SCM system must satisfy conflicting requirements. These are the need for a powerful and dedicated OMS with high modelling power (and thus complexity), and the need for files and directories which are simple concepts, but with low modelling power. We show how different systems have met these requirements.

This article shows how different aspects of the Adele system contribute to meeting these requirements and how this provides a highly customizable and extensible system with services for complex data management, tool integration, inter-tool cooperation, process support and configuration management.

Key Words: Configuration Management, Version, Software Process.

4.1 INTRODUCTION

Software, and especially software quality, can no longer be understood without taking into account the Process Model, i.e., all the actors, activities, methods, tools and procedures used throughout development, from the specification of requirements up and including maintenance. Quality cannot be added during the testing phase! This has only become apparent recently, and

Configuration Management, Edited by Tichy
(C) 1994 John Wiley & Son Ltd

the goal of “process technology” is now to provide formalism methods and tools for defining, enacting (i.e. executing) and tailoring software processes.

The main difficulty in achieving this goal is a consequence of the duality between the two spaces manipulated in an SCM: the Repository Space and the Work Space. The first is intended to store the software objects and their versions according to a given conceptual product model. The second is the place where users and tools create, access and modify these software objects, usually unversioned files and directories.

Until now all SCM tools have been characterized by shortcomings in one, at least, of these two areas. At one extreme we have SCM systems which offer good services for Repository Space management, i.e., a good Object Manager System (OMS), but little control over the corresponding files; at the other extreme are WS managers supporting only files and directories as product model (i.e., a file system model).

We propose an approach that reconciles the various aspects of SCM for providing simultaneously powerful models and management for RS, WS and Processes. But, more importantly, an approach, a way of provides making these aspects cooperate harmoniously. Each object used in the WS has a corresponding object in the RS. Activities are controlled and supported by Processes whether they are in the WS or in the RS.

This paper is organized as follows: in chapter two we present the different RSs and show how versions and products can be modelled; chapter three deals with the relationship between the WS and the RS; in chapter four we discuss more Process Management related issues and finally we present a global evaluation of Adele and our conclusions.

4.2 MODELLING PRODUCTS AND CONFIGURATIONS

When modelling products and configurations, two issues are of primary importance. Firstly, there is the problem of coping with object evolution, i.e., versions, and, secondly: there is the problem of representing products and configurations, i.e., complex objects.

Both issues have been looked at by, at least three different communities, namely, the Software Engineering Environment (SEE), the Computer-Aided-Design (CAD) databases and the Historical Databases. In this section, we first show how these communities solved these issues and how they evolved towards a data rich model in which complex objects and versions are explicitly defined. We then discuss Adele System evolution in relation to these issues and the new requirements.

4.2.1 VERSIONING MODES: AN OVERVIEW

In SEEs and CAD there were several proposals for clearly distinguishing the version, revision, and variant concepts. However, these concepts are still subject to much confusion. The main reason is the intertwining of different problems and different needs such as the versioning graph, parallel work, object evolution and historical information.

Almost all current commercial Configuration Managers have poor data models: they propose *file management* where *software management* is needed, i.e., the models do not allow complex software objects to be modelled and, as a consequence, it is impossible to define an adequate and customized Product Model. The versioning problem is reduced to the evolution of a single file, regardless of its content. This makes for flexibility but, on the other hand, little control is possible.

CAD systems use more complex data models because they are based on database technology. They use extensions to entity-relation, relational and, more recently, object-oriented models.

In the following paragraphs we present a brief overview of the different approaches.

4.2.1.1 VERSIONING IN SEE

In the early seventies, the SCCS system [Roc75] proposed a solution for the management of a single file regardless of its content. An object (file) is a sequence of *revisions*, one for each change to the object.

The RCS system [Tic82] is an extension of SCCS and solves the multiple development problem. It considers that evolution is not a simple sequence of revisions but forms a tree where each branch (or variant), follows its own evolution as a succession of revisions.

Almost all commercial version managers retain the solutions proposed by SCCS and RCS for multi-user control, history management and space consumption using the delta mechanism. The distinction between variant and revision is not always clear. In practice the only common property between all versions is that they share a large number of lines!

All tools based on RCS/SCCS feature a very weak data model (revision/variant). To support enhanced versioned models we need a data model in which the concept of version is explicitly supported. Few practical systems currently have this.

Aide De Camp is one of the few commercial systems based on the Change Set Model (see 4.4.44.1.1). A change set is a complex object containing the changes (i.e. file fragments) performed in all files for a given logical change. A version here is a change set, and arbitrary change-set composition is allowed. Unfortunately, in ADC, the data model is restrictive; there is no way to model complex objects explicitly. It is possible to associate attributes with entities, but the abstraction level is low.

Damokles [Dit89] was one the first prototypes to use database technology for SEEs. The Damokles data model allows complex and arbitrary objects to be modelled and enables objects be as the union of a set of (unversioned) attributes with a set of versioned attributes to explicitly defined. This idea formed the basis for most subsequent database approaches to versioning.

[Wie93] proposes an object oriented data model with an extension for modelling version families. This involves the use of partial objects, i.e., objects with attributes of known and unknown value. A given class may define different partial objects; all instances of that class whose attributes have the same values as the bounded attributes of a partial object pertain to that partial object family.

In PCTE [Tho89] the data model is an Entity-Relation-Attribute one; complex objects are built using links of the predefined “composition” category. Special commands are required to manage complex objects like “revise”, “snapshot”, “copy” which most often duplicate links and components, and rapidly lead to a very complex graph. Here, files and attributes are versioned.

4.2.1.2 VERSIONING IN CAD

In CAD systems the emphasis has been on the structure of Composite Objects and their versioning from a database perspective. A Design Object is a coherent aggregation of components, managed as a single unit. All three major database data models have been used, i.e., Relational, Entity-Relationship and Object Oriented Models. The relational model [HL82] was first extended to represent Composite Objects as a hierarchical collection of tuples, allowing long fields to contain data of unlimited length. The E-R model uses explicit relationships, with a predefined semantics, to manage aggregates and derivation graphs (*derived-from, is-part-of, etc.).* A survey of these systems is found in [Kat90].

In recent work on Object-Oriented Models the aim has been to separate the *physical* and *conceptual* levels of versioning. The physical level is concerned with the management and storage of derivation graphs (space saving). All SCCS type systems only support physical versioning.

At the conceptual level, a version is independent from storage representation. An object is a set of versions; the system must provide mechanisms with which to structure and control this set. Almost all models are based on the Generic Object concept. A generic object has two parts: a set of attributes (shared attributes) and versions (version group). An instance of a generic object is the union of the shared attributes and one of the versions pertaining to the version group. The semantics of the version group is defined and maintained by the applications.

[Sci91] has put forward a theoretical object-oriented data model for powerful versions management. Here a versioned object is composed of a single generic type and a set of version types, thus allowing multidimensional versioning. The logical versioning model allows the application to give the semantics to versions, by means of parameterized selection predicates.

4.2.1.3 VERSIONING IN TEMPORAL DATABASES

Temporal databases deal with the problem of entity evolution in time. This is a logical versioning since versions are related to time, and the semantics of time is known by the system. It allows queries to be made at a high level, for instance using time intervals.

The main requirement in temporal databases is for queries information concerning entities in the past; and for the retrieval of time slices. Configuration Management is not a relevant problem. Usually, the data models are relational models where the time dimension has been added. Most of these models distinguish between *valid time* and *transaction time* dimensions. The former represents the time as a fact observed in reality, while the latter indicates when the fact was recorded in the database. The temporal query language TQuel [Sno93] is an example of one that supports these two dimensions.

In Software Engineering, since most entities are directly managed in the system , valid time and transaction time are identical (most software artifacts are directly controlled files). Furthermore, since all times are taken from the machine, we may assume a complete ordering of events. These two properties, along with specific SEE needs like traceability explain why temporal databases have never been used in SEE. Whatever the case may be, the control of the temporal dimension is an SEE need which is usually underestimated.

4.3 COMPLEX OBJECTS AND CONFIGURATION MODELS

In Historical Databases the configuration problem is not relevant at all. In CAD systems the problem is slightly different. The principal difference between CAD configurations and SEE configurations is the fact that, usually, the structure of a “Design Object” is known beforehand; components are stable and clearly defined; there are no derived objects. The problem is limited to the selection of the appropriate version for each one of the versioned components (this is the dereferencing problem).

It is in an SEE context that configuration management appears in all its complexity; this is due to the extremely “soft” nature of the components, their “unlimited” refinement, the fact that components are really managed by the computer (whereas a circuit is not), and the existence of tools for object derivation.

In an SEE context we find the System Model (SM) concept. It describes the structure of software, the set of components and their relationships in a generic way. Each configuration manager uses a System Model as input. System Models can be found in different forms: we have the “system model” in DSEE [LC84], [LCM85], [LCS88], the “component” of NSE [Mic88], the “makefile” in Make [Fel79], the “structure model” in [Wat89]. Modules Interconnection Languages also provide System Models [RPD86], [Per87]. See [Tic88], [Est88], [Dar91] for a survey of configuration management.

What all these systems have in common is that the SM must be provided directly by the user. To build a System Model manually requires extensive work and its consistency is very difficult to ensure: the SM must be updated when software changes. In a large software product this is an error-prone process since the SM is large and it person who changes the software is not usually the one who updates the SM.

4.3.1 THE ADELE VERSION AND PRODUCT MODEL

We present here the commercial Adele Data/Version model and the current Configuration Manager.

4.3.1.1 THE BASIC MODEL

The Adele data model is object-oriented. The model extends this formalism with relationships, objects and relationships being treated in almost a homogeneous way.

Objects can represent files, activities and functions as well as simple values like strings or dates. Relationships are independent entities (external to objects) because they model associations with different semantics such as derivation, dependency and composition. A relationship is set up from an *origin* object to a *destination* object.

A Type describes the common structure and the behavior of the instances of that type[1]. The structure corresponds to a set of *attributes* modelling properties of the instance. Relation Types have additional information related to the *domain* of the relation, i.e., constraints defining between which object types the relationship can be established. The behavior is encapsulated into operations (*methods*) and extended by means of triggers (see 4.5.1)

External names must be assigned to objects at creation time; a relationship is also identified by a unique name. This name comes from the concatenation of the Origin object name (O), the Relation type name (R) and the Destination object name (D) to give the string, O|R|D.

Attributes model the properties of the instances. An attribute is defined by a name, its domain type (integer, date, boolean, string, enumeration) and other information such as the default value, the initial value, whether the attribute can be multi-valued or not, whether or not its value is dynamically computed, etc.

Types are organized in a type hierarchy corresponding to specialization. A type can have one or more super types (i.e multiple inheritance).The subtype relationship is a relation of inclusion between corresponding extensions.

1. When confusion does not arise, we use the term “instance” to refer either to an object or an association and “type” to refer either to an object type or a relation type.

Methods are used to model the behavior of the instances. They are defined by a name, a signature and a body or implementation. Methods and triggers are inherited down through the type hierarchy but unlike methods, triggers can not be overloaded.

The Adele Version Model is based on the *branch* concept. A branch models the evolution of a file and its attributes. It is a sequence of *revisions* where each revision contains a snapshot of the object.

In this Adele version, Composite Objects are not explicit in the model; the attribute domain cannot be an object type.

Different kinds of derivation graphs can be defined, establishing explicit relationships between branches; version groups can thus be easily defined. The Adele kernel provides the generic object concept (a branch is an object) and a mechanism for sharing attributes. Arbitrary versions and composite objects are created and managed using explicit relationships between the different components.

However, version groups, versioned objects and composite objects are not explicit in the model, their semantics and building are left to the application.

4.3.1.2 CONFIGURATION PRODUCT MODEL

The Adele Configuration Manager relied on a predefined product model. This model was designed to represent modular software products and automatically compute software configurations.

This model borrowed the *module* concept from programming languages: the association of an *interface* with a body or *realization*. The distinction between interface and realization is interesting for numerous reasons, and has been widely recognized [Par72], [Sha84], [Kam87]. This Product Model has three basic object types, viz. Family, Interface and Realization. The *Family* object maps the *module* concept. Families allow Interfaces to be grouped together. It gathers together all the objects and all their variants and revisions related to a module. A Family contains a set of interfaces [EF89].

Interfaces associated with a Family define the features (procedures, types, etc.) exported by that family. An interface contains realizations [HN86]. Realizations are functionally equivalent but differ in some other (non-functional) characteristics. None is better than the others, and none is the main variant; each one is of interest and needs to be maintained in parallel.

An *is_realized* relationship is defined between an interface and each one of its realizations. The dependency relation is also predefined. X and Y being variants, X *depends_on* Y means that X needs Y directly (usually to be compiled). Often it simply mimics the #include directive found in programs.

4.3.1.3 THE CONFIGURATION MODEL

Building a configuration in Adele involves selecting a complete and consistent collection of objects. The collection is complete with respect to the structure of a software system defined by the dependency graph closure and consistent with selection rules. The dependency relation being defined over variants, a system model contains variants; it is a generic configuration [BE86].

The *depends_on* relationship is an AND relation (all destination objects must be selected) while the *is_realized* relationship is an OR relation (only one destination must be selected). The dependency relation is an acyclic graph.

To be consistent, a system model must satisfy (at least) the following rules:

R1 = Only one interface per family is allowed.
R2 = Nodes must satisfy the constraints defined by the SM.
R3 = Nodes must satisfy the constraints set by all node ancestors in the graph.

Point 1 ensures that functionalities will not be provided twice. Point 2 ensures that all components are in compliance with the characteristics required by the model. In Adele, the characteristics required by the model are expressed in a first order logic language (and, or, not connective) constraining component properties. Examples of such constraints can be *recovery=Yes and system=unix and messages=English* [Est85].

Point 3 ensures that all components are able to work together. All compatibility constraints between components are associated with the component ancestor in the graph. Such constraints in Adele use the same language and can for example be *if (arguments=sorted) then (system=unix_4.3) or (recovery=no).*

4.3.1.4 CONFIGURATION INSTANCE.

An instance of a generic configuration, also called a bound configuration is a set of revisions, one for each variant of the generic configuration. The instance is defined by the constraints to be applied in order to select the convenient revision for each variant. This selection is made in Adele on the basis of the revision properties, using the same first order logic language. An example of this sort of expression could be

```
((reserved=john) or (author=john) or (state=official)) and (date>18_02_89).
```

In Adele, a configuration is a standard realization: its interface is the root of the graph, its source text contains the constraints, its components are the destinations of the relation *composed_of* and its revisions are its instances. As a consequence a configuration also evolves as variants and revisions in the usual way.

Evolution can be seen as a twin speed process. Firstly, we have fast evolution, when a configuration process is used as baseline. This is every day evolution; it covers bug fixes, minor enhancements and development, in other words all changes leading to new revisions. Slow evolution is when a new configuration must be built. This is only necessary if new variants are to be incorporated in the model: new functionalities are needed, non-functional characteristics are desired or new internal constraints have been set.

The *current value* at time T of a configuration is its instance built at time T without constraints. Changes performed, in a configuration, by user *john* can be captured using attributes *reserved_by* or *author.* This is similar to DSEE [LC84], [LCS88] and ClearCase [Leb93]. Any configuration can be built at any time by setting a cut-off date using *date<T* as a constraint and it is easy to reuse old revisions of the SM. A special configuration instance can be built using other revision attributes: *state=official and date<88_08_23* retrieves the official instance (if it exists) of this configuration as it was on August 23rd 1988.

4.3.2 EVALUATION

This configuration builder and its associated product model were released in 1984, and have been used intensively by many customers. The experience gained brought out some drawbacks:

- The default product model does not match all software easily. In particular old products being designed with other structures in mind have problems in finding an interface/realization association.
- The version definition and evolution constraints are often too rigid.
- Other constraints for building software configuration could be defined, using roughly the same process.
- The default product model does not easily support objects other than software programs.

These considerations led us to open up the data model, based on a general model for complex object management, a general versioning model including extended history, and general process support.

4.3.3 EXTENSIONS TO THE ADELE DATA AND VERSION MODEL

We concentrate here on static characteristics, i.e., how objects are built UP, and how they are versioned. The Data Model has been extended to allow:

- Composite Object definitions of any kind.
- General Versioning and history
- Automatically built objects, including configurations.

We shall briefly cover each of these points.

4.3.3.1 COMPOSITE OBJECTS

An Object Type is a set of attribute definitions whose type can be *simple* (integer, boolean, string, data, or file) or *complex*, i.e., any other object type. The attribute value is a reference to another object. In both cases, the attribute can be multi-valued (set-of). For example:

```
OBJTYPE Module IS cobj;
          responsible     :   STRING;
          spec            :   FrameDoc;
          interfaces      :   Int_Type;
          test            :   set_ of files;
END Module;
```

Complex attributes are a special case of composition relationship. To each complex attribute name there corresponds a relation type with the same name.

```
RELTYPE spec IS...;
....
END spec;
OBJTYPE FrameDoc IS Document;
...
END FrameDoc;
```

The composition semantics is defined by the behavior of the corresponding composition relationship; i.e., by the relationship methods and triggers (see chapter 3). Using the subtyping mechanism the relationship behavior can be inherited and refined; predefined relation types like the *Has* relationships have a semantics recognized by the kernel. Using triggers it is possible to express, for example, that the components are shared or not, that a *copy* of an object produces a *copy* of its components; etc. Once a relationship is established between two objects, the behavior of both objects is automatically modified; this is what we have called *contextual behavior* [BM93].

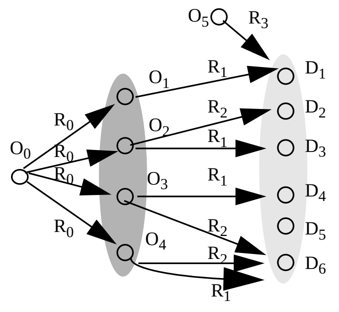
QUERY LANGUAGE

Objects and the relationships between them can be seen as graphs: objects are the nodes and relationships are the arcs. The query language uses this fact to reach the instances by navigation through graphs. It also offers a filtering mechanism for selecting the instances on which the operations will be applied.

Querying the database is takes place using *path expressions*. These are defined by a *navigation* and a *selection*:

```
path        ::= navigation
             |   navigation path
navigation  ::= nodes -> arcs [ (selection) ]
             |   nodes <- arcs [ (selection) ]
```

Navigation. Navigation allows instances to be reached starting from a set of instances and following a path through the graph. A navigation expression defines a multiple path on a graph. As associations are directed, navigation can take place from the origin object to the destination object (noted ->) and in the other direction, from destination to origin (noted <-). *Nodes.* This term denotes the set of objects from which the navigation starts while *arcs* refers to the set of relationships to be crossed. This means that navigation take place along various paths at the same time. For example:



nodes = { O_1, O_2, O_3, O_4}

arcs = {R_1, R_2 }

O-> R_0->R_1, R_2 = { D_1, D_2, D_3, D_4, D_5, D_6}

D_6<-R_1, R_2 = { O_3, O_4 }

O_0->R_0->R_1,R_2 = { D_1, D_2, D_3, D_4, D_5, D_6}

O_0->R_0->R_1,R_2<-R_3 = {O_5}

Selection. A selection allows the set of instances reached by the navigation to be filtered. It is a logical expression wherein the variables are instance attributes. The expression is evaluated for each instance in the set returned by the navigation. The result of this evaluation is the instances for which the expression is true. If there is no selection expression, the initial set is returned.

```
foo->dep (state=stable and language = english)
```

This returns from all the objects *foo* depends on, those satisfying the condition (*state=stable and language = english*).

Substitution allows the values of instances attributes to be accessed. The syntax is:

```
        ~instance%attribute
```

where *instance* is a set of objects or associations and *attribute* is an attribute name. The result of a substitution expression is a set of instance names. Note that substitutions can be used in any term of a *path*.

4.3.3.2 GENERAL VERSIONING

We identified three classes or versioning:

- **Temporal**. There is a great need for *traceability* in an SEE database. Historic objects, control of activities and far reaching consequences of actions must be retrievable found, even much later.
- **Logical**. Objects exist in multiple *variants* or representations.
- **Dynamic**. Multiple and concurrent *activities* are taking place in an SEE database. An SEE database must support software processes.

These three dimensions are completely orthogonal. The first is the time dimension, designed for history and traceability reasons; The second arises from the need to support logical versioning, i.e., *real* versioning. The third dimension is related only to the support of processes and so called long transactions, i.e., dynamic versioning.

TEMPORAL VERSIONING

Our experience and involvement with process modelling showed that one aspect has been underestimated in all systems, we are referring to traceability. We need a generalized history mechanism intended to answer questions involving multiple objects (some being currently deleted) over a long period of time. Given a change request CR, we must be able to answer the following questions: “what are all the actions performed as consequences of this CR”, “who decided to accept this CR”, “which changes in which objects have been performed to include that CR”; “in which configuration are they present”, “for how long is it required”, “who was involved and when”, “how expensive was it ”?.

Our temporal versioning was designed with these requirements in mind. An object is a set of attributes, including files and other objects. Any attribute may have *immutability* characteristics which means that any attempts to change its value automatically produces a new “state” (i.e., revision) of the object. Note that in most systems, a single and implicit immutable attribute is assumed: the file. Remember that, in our system, files are attributes.

An object *State* is defined as the attribute values of that object at a given point of time. Between two successive states there is at least one attribute whose value has changed. A *generic object* is the set of all its successive states, from creation until now.

```
OBJTYPE Module IS cobj;
        responsible     IMM     :   STRING;
        spec            IMM     :   FrameDoc;
        interfaces              :   Int_Type;
        test                    :   set of files;
END Module;
```

In this example, the *responsible* and *spec* attributes are immutable. It means that each time these attributes are modified, the system keeps the previous object state and creates a new current state. The value of a complex attribute, such as *spec*, is a reference to another object, we must indicate whether it refers to an object state or to the generic object.

Using the navigational notation, we can access directly past states and we can make queries involving past information, we refer to this as historical navigation. Note that, in this model, revisions constitute a special case.

A generic object is identified by its name and an object state by its generic object name followed by the "@" sign and a selection clause. For example, if *ctrlmem* is the name of a module we can write

```
1     ctrlmem@now
2     ctrmem@(responsible = john)
3     ctrlmem@ti->interfaces
```

Expression 1 returns the current state of object *ctrlmem*; expression 2 returns all *ctrlmem* states for which *john* is *responsible*, and expression 3 returns the *ctrlmem interfaces* as they were at time *ti*.

For traceability purposes, states are not the only concern. They are also arbitrary facts and events occurring in or out of the database. Adele allows history types and history instances to be recorded, when an arbitrary event happens. A history object is similar to any other object but cannot contain complex attributes. Tool executions, time events, coordination events, logs, object creation/destruction, side effects, relationships between entities can be recorded and historic relationship paths followed at any time later on.

Complex expressions can be written using historical navigation. As a special case, if no state is given in the expression, navigation is entirely on the DB "surface", as in any other DBMS.

LOGICAL VERSIONING

From a conceptual point of view, an object is a generic entity, i.e., the sequence of all its states from its creation until now, X = seq {X@1, X@2,....X@now}. At a given point in time there exists a single current state for a given object. The evolution of the object is linear. In the following the word object means generic object; its historic dimension will not be taken into account.

From a logical point of view an object can evolve simultaneously in different ways. Thus, at a given point in time, there are different current states of the same object, and thus different generic objects. These different generic objects are versions or variants of the same object.

If a complex attribute has *versioned* characteristics, its value is a *version group*, i.e., a reference to a set of objects that are versions. A *versioned object* is an object containing at least one version group.

By construction, at the relevant level of abstraction, a versioned object is perceived as a single object (unversioned). An *instance* of a versioned object is that object restricted to a single value for each one of its version groups.

In our data model, version groups are defined by the key word VERSION. In our example:

```
OBJTYPE Module IS cobj;
      responsible     IMM :   STRING;
      spec            IMM :   FrameDoc;
      interfaces          :   VERSION Int_Type;
      test                :   set of files;
```

```
END Module;
```

The interface attribute refers to a version group.

DYNAMIC VERSIONING

If different processes share the same data, conflicts will arise. Traditionally this is solved by the locking mechanism; a single activity may change an object. However, for process support, we need a higher level mechanism for controlling overlapping activities.

We need both to give each process the illusion of working alone on "its" object, and the ability to define, control and enforce user-defined collaboration and synchronization protocols. These aspects are described later.

The former aspect is supported by *dynamic versioning,* i.e. the dynamic and transparent (for users) creation of object copies as part of the protocol to solve the concurrence control problem. They are temporary versions, automatically created during process execution and automatically removed when that process terminates. When a process accesses an object, its private copy is automatically assumed. This dynamic versioning is the basic mechanism on which Adele sub databases are built (see 4.5).

We think that synchronization and collaboration protocols must be explicitly defined in the Process Model.

4.3.3.3 BUILT OBJECTS

Configuration are now generic; they are a special case of built objects. Built objects are those complex objects whose components can be found by an automatic computation directed by generic build constraints. Generic build constraints can be expressed at type level. These express the process to be followed when any object of that type is to be built, including consistency constraints. For a configuration this process is roughly defined by

```
starting_object -> (dep->is_realized(unique))+
```

which expresses that any configuration is built from the transitive closure "(path)+" of the *dep* and *is_realized* relationships. All *dep* relationships must be followed, but a unique *is_realized* relationship must be followed (*unique*). Generic consistency constraints can be specified in build constraints, for example, rules in chapter 2.3.3. can be expressed as follows:

```
R1 = IF !selected!famname = !famnanme THEN ERROR ;
R2= EVAL (%select).
```

The R1 constraint states that a single interface per family is allowed. R1 must be checked when reaching interfaces, thus after traversing *dep* relationships. R2 states that constraints expressed in the *select* attribute of the configuration instance being built must be verified. Thus a more realistic generic configuration builder can be expressed by

```
-> (dep(R1)-> is_realized(R2, unique))+
```

The selection constraint specified in a select attribute can be for example:

```
select = "recovery = yes and system = unix and message = english"
```

Unique is a built-in function returning a single one of the objects satisfying rule R2 at each navigation step.

This work is under way, but first experiments have shown that the current configuration manager is definable in 1/2 page of process definition.

4.4 WORK SPACE CONTROL

In the previous section we presented the data and version models proposed by currently available CM tools. These models are intended to define the content of the *Repository Space* (RS), i.e. which objects can be stored, and how they are versioned. It is the task of the Object Manager System of each CM tool to manage the Repository Space, i.e., to represent, optimize, access, enforce protection and internal consistency of the RS. However, this is only the static aspect of CM. Storing and representations are not sufficient.

The dynamic aspect of configuration is performed in *Work Spaces* (WS). A WS is the place where the usual Software Engineering tools are executed. The objects that can be found in a WS depend on each system, but must include files and directories, since they are the only entities known by standard tools (editors, compilers, etc.). We can say the product model governing WS includes the File System (FS) model, i.e., file, directory and link types.

Normally, a single version of files is present in a given WS, since standard tools do not recognize version concepts. It means that the RS, wherein all objects and their versions are stored, is different from the WS. The product model governing an RS is highly variable, but potentially high modelling power is allowed. Almost any concept, any kind of abstract and/ or complex object, any version model, any software and team definition and structure and history dimension can be defined. This contrasts with the simple model governing WSs.

It is clear for us that software engineering can only really be supported if WSs, RSs and Processes are closely controlled and managed, and *if they work in symbiosis.*

We shall first present the relationships between WS and RS, i.e., how they communicate and what correspondences exist between entities managed in the RS and the WS. Secondly, we will present the relationships between WSs and configurations, i.e., how activities in WSs and in the RS can be controlled and synchronized. Finally, the Adele proposition for solving the various problems caused by the duality of these two spaces is presented.

4.4.1 WORK SPACE AND REPOSITORY SPACE RELATIONSHIPS

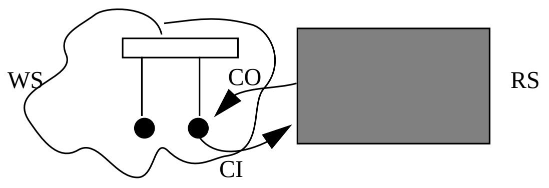
Some of the entities stored in the RS are also present in WSs when they are needed to perform activities.

We have to satisfy two opposing requirements here: The WS model must be as close as possible to a bare File System (FS), in order for users and tools not to be disturbed. The learning curve for most users should be steep, and ideally almost vertical. On the other hand, the modelling power of the RS should be as high as possible, in order to easily represent software artifacts, tools, users, teams, schedules and processes, to allow different versioning models, to support process management, to provide team leaders with adequate of information, to help responsible for configuration with services and so on. The “ideal” RS product model is not simple!

Software CM systems are responsible for securing correspondence and consistency between the two worlds. It is interesting to see how different CM tools solve these conflicting requirements. We present here an overview of how the various CM systems have solved these conflicting requirements.

4.4.1.1 BASIC TOOLS

In basic tools, like RCS and SCCS the RS model is implicit, embedded in the CM and limited to revisions and variants for a single file (there are no configuration or aggregate concepts). The WS model is the bare FS. Two basic operations are provided Check-In (CI) and Check-Out (CO).



CI/CO tools

Pros:

- WS Model = FS thus no learning overheads!
- RS Model is simple. Easy to learn.
- Can be used for any file.

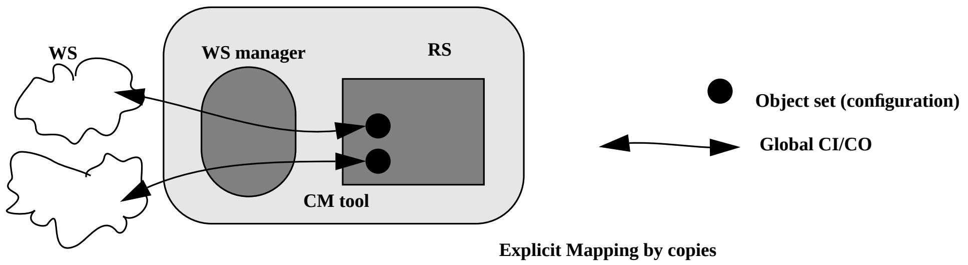
Cons:

- Very low level RS Model. No configuration or aggregate concept.
- The CM has almost no knowledge.
- Cooperative work: no support out of locks.

These simple tools oblige users to establish their own rules and conventions. All consistency control is left to users. On the other hand, they are free and necessitate little training. These tools are not CM systems but basic version managers.

4.4.1.2 EXPLICIT MAPPING BY COPIES.

In this class of tools, the mapping between the WS and the RS is explicitly managed by the CM; files are explicitly copied back and forth between both spaces.



Explicit Mapping by copies

This approach extends the previous one in a number of ways; CI/CO is global, a (hopefully) consistent set of files is simultaneously checked in/out and the CM has some knowledge of the WS. It can usually manage different WS simultaneously.

The majority of CM tools belong to this class, even if many differences can be found. Different WS management strategies can be developed in this class. The power of the CM tool depends essentially on the possibilities of the RS Model and the functionalities of the WS manager.

WSs can be controlled if all file access operations are encapsulated into WS manager commands. This is the solution most CM tools use. Even in this case the WS is a bare File System; a single file is present in a WS and no WS local versioning is available. For that reason, this approach has difficulties in managing the interactions between the different WSs and cooperative work cannot be easily supported. NSE [Mic88] was one of the few systems to propose implicit local versioning, but conversely imposes a specific cooperative policy. Most often these systems use a static locking mechanism; an object (file) can be modified in a WS only.

Pros:

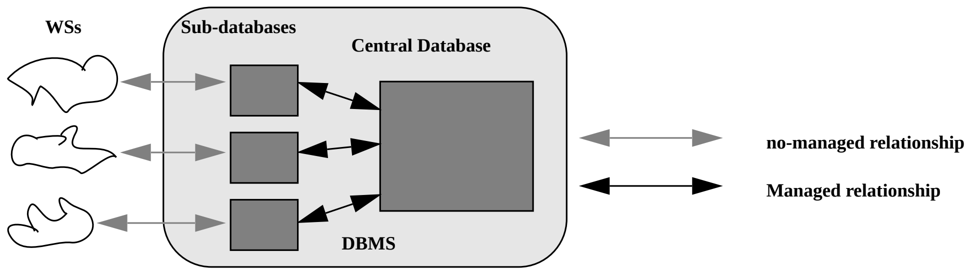
- The WS Model is the FS Model. No overheads.
- WSs can be managed simultaneously, built globally and controlled.
- The RS Model can be powerful.

Cons:

- The WS model is very low level (the FS Model)
- No (conceptual) mapping between RS Model and WS Model.
- Collaboration between WS is ill supported.

4.4.1.3 SUB-DATABASES.

The database community, for different reasons have tried to solve some of the problems of the previous approach. Versioning in databases led to the interesting concept of Sub-databases.



Sub-database approach

A sub-database (sub-database) is a sub-set of the central database instances. Each sub-database is isolated; a change to an object is visible only in the current sub-database. A sub-database is often seen as a (long) transaction, changes are propagated to the central database only when “committed”. This is similar to the usual WS, replacing *object* by *file,* but does open up interesting new possibilities.

The RS Model can be very rich as not only files can be managed. Usual DB services like (short) transactions, associative queries, distribution, protection, and so on can be used. Complex and precise protocols can be defined between the central DB and its sub-databases; transparent versioning can be provided, and concurrent modification of the same objects can be supported.

The main drawback is that a sub-database is not a WS. In these approaches there is no WS support, no control of activities performed there, and, consequently, little potential for process support. This is a new and difficult domain; current implementations provide only partial solutions.

In Orion [KBGW91] there is no direct communication between sub-databases, and a single level of sub DB is possible. In Damokles [Dit89], explicit versioning and simple sub-databases are provided.

Pros:

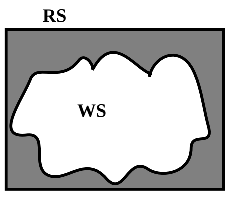
- Cooperation protocols can be defined and concurrent activities supported.
- A powerful RS Model can be supported.
- Database services can be used.

Cons:

- No WS support.
- No activity control, poor Process support potential.
- Not a mature technology.

4.4.1.4 WS = (VIRTUAL) FS = RS

Some tools decided to solve the mapping problem between WS and RS by behaving as if the RS WERE the WS. Of course, some “tricks” are needed (if only to solve the multiple version problem). Either the FS supporting the WS must be “adapted”, or tools must be encapsulated



Work Space = Repository Space approach

ADAPTING THE FS.

Different lines of approach have been followed to “extend” a file system in order to support this approach. The first attempts involving “cheating” with NFS. The FS supporting the WS is mounted by NFS, and the NFS driver, on the machine supporting the physical FS, is replaced by another one which decides dynamically, when opening a file, which version should be provided [AS93].

A more delicate approach, followed by ClearCase [Leb93], is to write a specific Unix FS driver. Any access to a file in this Virtual File System (VFS) is interpreted by this driver which decides dynamically on the file version to provide. A more powerful RS Model can also be provided. Transparency is provided at the cost of efficiency; all accesses to files are interpreted.

The same kind of trick can be done by replacing the open/close functions in the libraries by a program which selects the real file to open. A tool like 3D, COLA [KK90] does this kind of work, and should be available on most Unix machines [BK93]. It assumes that all tools use dynamic links with I/O libraries.

ENCAPSULATING TOOLS

CaseWear [CW93] prefers to let users access the CM repository directly, but to solve the versioning problem, all tools have to be encapsulated.

In PCTE [Tho89], the user must be logged under PCTE, which first interprets any command and file access. It is a totalitarian approach (complete encapsulation). Furthermore PCTE is not a CM tool and no version selection is automatic; tools need to be encapsulated if versions are used.

In any case, it must be borne in mind that, tools can be encapsulated whilst users can not.

Pro:

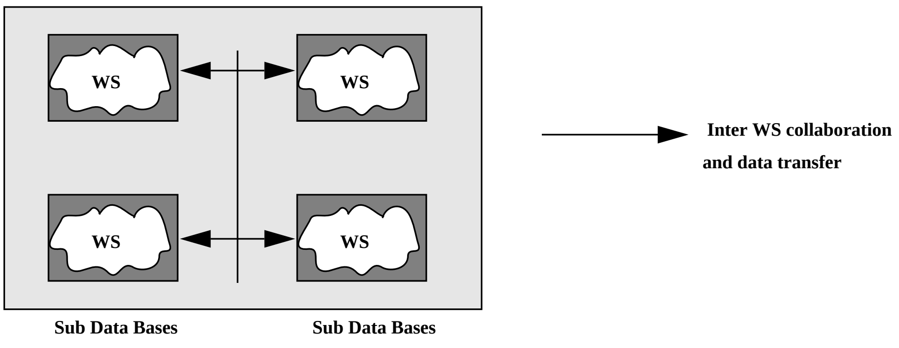
- The WS Model is extended by the RS Model.
- Implicit and transparent version selection.
- Easy adaptation for users.

Cons:

- The RS Model must be an extension of the FS Model.
- The FS must be changed, or tools encapsulated.
- Single RS/WS mapping is possible. The software structure is rigid.

4.4.1.5 WS = SUB-DATABASE = RS

Most drawbacks disappear if it is possible to directly map a WS to a sub-database.



In this solution, the WS IS a sub-database. It means that all the DBMS services are available in each WS (local versioning, queries, short transactions, protection, and interfaces). It is the availability of all these services, along with the fact the DBMS knows the various WSs well, i.e., where they are, who created them, when, for what purpose, and so on, which explains why the concept of sub-database makes it possible to define the relationships between WSs, as well as the relationships with the central data base.

It should be noted that each sub-database (WS) may have a different “view” of the same objects, both structurally (the directory/file sub tree may be different), semantically (the attribute and relationship may be different for the same object) and behaviorally (the constraints and methods for a given type may be different). The WS management policy may also be different (a different process can control each WS).

Note also that a direct consequence of the sub-database concept is that each individual user has the illusion of working alone on his objects. Here, we have both simplicity for end users and considerable modelling and control power.

Pros:

- The WS is enriched with RS concepts and DBMS services.
- Collaboration and cooperation patterns can be defined and controlled.
- Each WS defines and supports different structures, processes and properties.

Cons:

- Complex immature technology.
- Numerous open issues.

This solution opens up some interesting doors. Since the structure of each WS can be different, checking out the same configuration may end up in very different WSs. It can be envisioned that a WS is actually the internal storage for another platform or tool (PCTE or VMCS for example). In this case each platform or tool works in a different sub database, on a well defined configuration. The central tool is in charge of controlling the inter cooperation protocols and the versioning dimension.

This solutionis the one used for our project and forms the basis for the Adele Work Space manager presented below.

4.4.2 THE ADELE WORK SPACE MANAGER

Following the previous approach, the system has to solve two apparently incompatible classes of problems:

- The isolation problem between WSs (sub-database concept),
- The coordination problem between WSs.

4.4.3 ISOLATION AND SUB-DATABASES

A WS can only be a full sub-database if it is isolated, i.e., if it has the usual ACID properties of transactions: Atomicity, Consistency, Isolation and Durability. However, ACID, in the framework of short transactions, and ACID, in the framework of long transactions, have a slightly different meaning.

The fundamental difference is that in short transactions isolation is provided to ensure consistency by an enforcement of sequentialization, whereas, in long transactions, isolation is provided to ensure consistency when parallelism is allowed. The isolation of the new and modified data is performed using dynamic versions, and by applying a visibility mechanism so that a transaction does not “see” the objects produced/modified by other transactions.

Setting a lock becomes the transparent creation of a copy (dynamic versioning) and releasing locks becomes the merging of the dynamic copies.

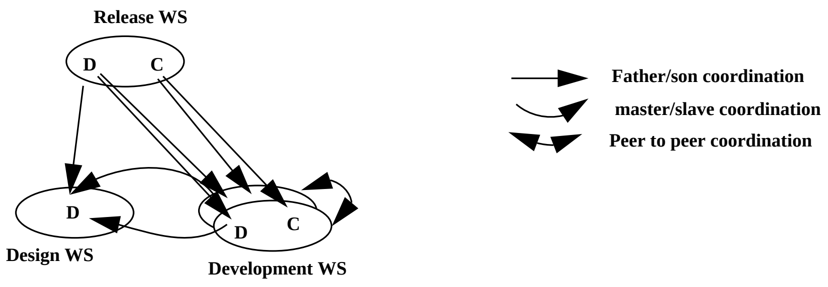
4.4.3.1 COORDINATION CONTROL.

Coordination is related to the management of concurrent changes to shared artifacts. We need the ability to define when a modified component must be propagated (notified, made visible, copied, merged, etc.) to the other WSs containing (copies of) this component.

The fundamental reason for coordination is that object merging is not a perfect mechanism. Inconsistencies may arise from an object merger; the probability of problematic mergers rapidly increases with the number of changes performed in both copies. Were mergers to be performed only at transaction commit, most of them would not be successfully performed. Frequent mergers, at some well defined points, are needed to maintain two cooperating WSs in synch.

Of course, if the coordination mechanism is used, work under way in a WS is made visible to other WSs while they are operating. A WS can then no longer, be considered in any aspect, an ACID transaction. This is why the word transaction in this document only applies to short transactions

Coordination control will be presented using the following example: A *Release* WS contains a Design document (*D*) and a configuration (*C*); it has sub WS *Design*, where the design is developed, and a *development* WS containing both. Development WS exists in two instances.



Classically, the father/son relationship is identified, it links a WS (the son) with the WS wherefrom its objects are issued (the father). If a WS contains more than one object, a potentially different father/son coordination applies to each original/copy pair.

In our example the two design WSs must coordinate their work using a coordination policy different from the release/design coordination (peer to peer coordination). Similarly, design and development may coordinate their work (when a design is changed), with still another coordination policy (master slave coordination).

In actual fact, all these coordinations are identical as far as we are concerned; peer to peer is the same coordination type instantiated in both directions; a father/son is just a special case of master/slave. All coordination operations rely on the following operations:

- **promote**. The modified or created object components in the WS become visible for its parent WS (and eventually merged);
- **resynch**. The components updated in the master WS are copied from the master to the slave WS (and eventually merged).

Coordination is *Asymmetric* since only the slave can execute operation resynch. The different coordination policies are implemented, in our system, depending on the way copies are created and resynchronized, on the moment of synchronization and so on. For purposes of illustration there follow the most popular predefined coordination policies.

Private coordination: This is an explicit coordination; at the user's request coordination (through the resynch operation) is performed. This is the default father / son relationship.

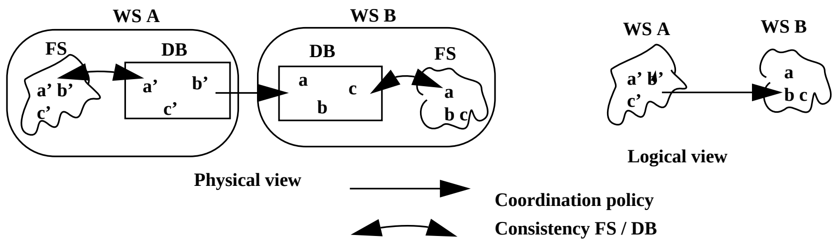
Controlled coordination. The resynch operation is automatically performed when a user defined condition, defined on the slave or the master components, is satisfied. (see an example in 4.5).

4.4.3.2 FILE SYSTEM VS. DATABASE REPRESENTATION

For activities executed by people such as like the project leader or the configuration administrator, there is a need for high level and abstract concepts. The team leader would like to manage users, schedule, teams, etc.; the configuration manager has to deal with complex objects, queries, consistency, transaction and so on. These users need a substantial modelling power, and thus an advanced database representation.

Conversely, most Software Engineering activities manage files and use tools for managing files. this is the case with all production activities such as designing, developing, testing, validating etc., and tools such as editors, compilers, linkers and debuggers. Clearly these activities need a file system representation. It follows that at least two representations are of concern, namely, the database (DB) and the file system (FS) representations.

At first glance, these requirements conflict. How can a WS containing a FS representation (called a File system Work Space: FS_WS) be built which meets all the above requirements? The guiding principle for such architecture is the following:



Managing a File System and a Database view

The characteristics of this solution are that a WS, with an FS representation, is in fact a pair: a WS in the DB (called DB_WS), and a part of an FS system (called FS_WS) containing copies of the files present in the DB_WS. In this solution, the FS is a real operating system FS.

This implementation meets requirements for openness, transparency and efficiency but raises a very difficult problem. The FS supporting the WS is not controlled by the data base. There is an inherent shift between the FS and the DB WS[2]. If the FS_WS is a bare FS,

2. . The DB_WS is designed to support process enaction, and the FS_WS to support (parts of) process performance. Not surprisingly, there is exactly the same difference between DB_WS and FS_WS as between enaction and performance.

requirements for cooperation control, high level modelling, and DB services cannot be met. Such an implementation can work only if the system can enforce consistency between the FS_WS and its corresponding DB_WS.

The *file mapper* is an essential basic component. It is responsible for determining the place and name (directory, files name, links, etc.) where an Adele object is (must be) located in the FS. Conversely, given a file in an FS, the file mapper knows which Adele object (if any) it is the image. As a consequence, and because all Adele command can be called directly from shell level, users can request for attributes and relationships of files, by simply providing file names. Examples are request for which files a given user has created or the dependencies of a file.

In the future, we envisage similar, simultaneous management. WSs supported by different operating system (e.g. Unix, VMS, DOS, etc.), supported by foreign platforms (Other CM tools, other DBM, PCTE), and in general any other object manager. This evolution would appear to be mandatory in future heterogeneous PCEs (Process Centered Environments) where various large-scale tools, each containing its own data management system, will have to collaborate. Our ambition is not to centralize and manage all the data of the environment, but to manage the WSs in which the data is managed, and to define and enforce the coordination between WSs. Great difficulties remain to be overcome. We feel that research direction (through WS support and coordination) is a useful approach to the problem.

4.4.4 WS MANAGEMENT AND CONFIGURATIONS

In the previous chapters we saw which kind of relationships can exist between Work Spaces and the Reference Space. This was a static view. We should like now to discuss the relationship between the activities performed in WSs and the configurations (which are in the RS).

4.4.4.1 PREVIOUS APPROACHES

All CM managers have a concept of WS and configuration, even if ill defined. Following Feiler and Dart [Dar91], [Fei91], the following classes of work space management strategies have been identified.

CI/CO.

There is no real WS management strategy here. The only service provided is a locking mechanism that ensures changes on individual components are sequentialized. Users must define and enforce rules and conventions themselves.

COMPOSITION

In the composition approach, configurations are entities understood by the system (they are part of the product model), and a WS is created from a configuration. The system controls which components are in each WS, such that it is possible to know how derived objects have been built and thus to reuse them later.

CHANGE SET

In the Change Set approach, configurations are also objects which are checked-out in a WS, but here the changes performed in a WS are known (at least their name) by the system as a logical change, and themselves stored as an object. A logical change applies to all modified files of the WS and bears a name. For a given configuration, the system is then able, to compose known logical changes to provide a new configuration incorporating all the changes performed in the various WS wherein this configuration was checked-out. Aide De Camp is the only commercial system that implements this approach; a more academic one is presented in [Mun93].

TRANSACTION

In the transaction approach, the work performed in a WS is seen as a transaction. Sub transactions can be defined. In this approach, relationships between a transaction and its sub-transactions are explicit, and managed by the CM tool. This can be compared with the sub database approach (4.4.1.3).

4.4.4.2 A PRIORI VS. A POSTERIORI CONFIGURATION BUILDING

It is interesting to note that two main approaches are in use for building configurations.

A PRIORI CONFIGURATIONS

Under this philosophy, a WS is built containing (normally) a configuration, with the explicit goal of building another configuration. For instance checking out V2.3 (a WS) configuration to build V2.4. The goal is known from the outset and, i file changes may occur and components may be added or deleted n the WS. At the end of the activity, the fact that version (V2.4) is valid is taken for granted and check-in builds version V2.4.

This philosophy underlies most CM tools: the user is in charge of defining the components and making sure the configuration is consistent; the CM tool does not provide any help in this process.

A POSTERIORI CONFIGURATION

Under this philosophy, the system provides some support to try and work out how to build a configuration, given some of its characteristics. This is not unlike a DBMS approach in which the content of the database and the components that could be used to constitute a configuration are known. It is an approach that privileges the reuse of existing components, whereas a priori configurations lead all too often to duplicate components.

In theory, almost anything can be checked-out in the WS, and components are checked-in along with their semantic properties. Configurations are built using only the CM system.

A posteriori configuration follows the composition approach if component selection is supported (and not just revision selection). Given the properties a new configuration must have, the CM searches for all components that could constitute the configuration. Completeness and consistency control are checked in the process. To a lesser extent the Change Set approach is in this class, since the system is able to “invent” component versions providing an original configuration (a baseline) and a set of change names are known.

Adele, in an earlier version, was a “pure” a posteriori approach: only the Adele configuration manager was allowed to build configurations. We soon realized that it was often too academic. Currently, both approaches are supported and complement each other. For instance, a WS can be built from an existing configuration or from a computed configuration. It is possible to check-in either components with their semantics, or indicate to the system that the WS now IS configuration X. Configuration X is created and contains the WS components. However, it is always possible to ask whether the X configuration is complete and consistent, and provide the semantics of the changed components, to help in computing later configurations reusing these components. The Adele WS Manager naturally supports this feature.

4.4.4.3 EVALUATION

Various strategies have been used for controlling activities and providing users with WSs; they all have their strengths and weaknesses. It is noticeable that all the previous classes of approaches involve an implicitly WS concept. The current state of the art is characterized by the following weaknesses at least:

- Either the RS or the WS has taken precedence; their relationship and consistency is frequently unsatisfactory.
- WS management strategies are predefined by the SCM systems, they are often inflexible.
- Coordination and cooperation are ill supported. Parallel work is either prohibited or unsupported.

We believe our WS manager subsumes all current approaches, in generality and flexibility. It has been designed by successively refining from previous ad-hoc WS management policies implemented on clients' requests. Predefined policies such as private, parallel, shared, exclusive) are provided in a library, in the form of predefined generic relationship types (see example in 4.5.1.3). They can easily be adapted, customized, refined and combined (private, exclusive for example).

4.5 PROCESS SUPPORT

A constraint which customers would often like to integrate is that "a configuration is valid only when all tests have been successfully executed". While this is quite understandable, supporting this type constraint requires in-depth knowledge of the activities to be carried out, the place where they are executed (WS) and the nature of the collaboration between team members (tests may be shared out among various people). Good configuration management needs powerful process support.

However, process support imposes extremely varied requirements on the system. It means that all actors, artifacts, activities, methods tools and procedures used throughout software development must be modelled, from requirement specification up to (and including) maintenance. The way in which activities are defined, enacted and controlled, services for global visualization of all activities and so on all need to be carefully thought through.

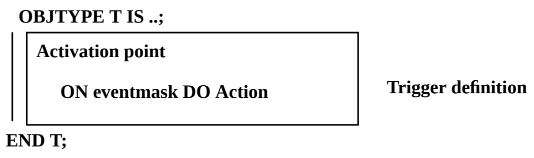
Process support, in Adele, takes place through a layer of services. The basic level is the process engine which is based on triggers. Overlying that comes the Work Space Manager, discussed earlier, and, finally, we have high level process support.

4.5.1 THE PROCESS ENGINE

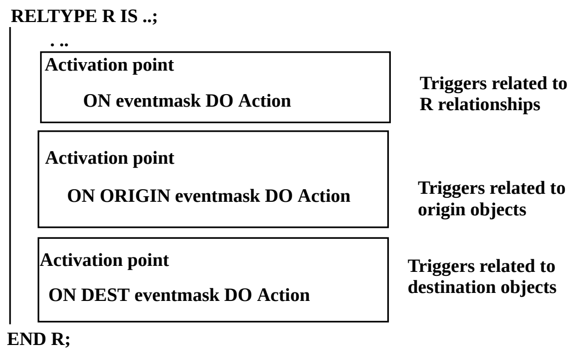
Our Process Engine relies on Event/Condition/Action (ECA) rules or triggers. The general expression for these rules is as follows:

```
"ON Event WHEN Condition DO Action".
```

We have modified these rules slightly. Triggers are defined within types and they are inherited along the type hierarchy.



In relation type definition there are two additional definitions: triggers related to events arising in the relationship origin object and triggers related to events occurring in the relationship destination object.


RELTYPE R IS ..;
. ..
Activation point
ON eventmask DO Action
Triggers related to R relationships
Activation point
ON ORIGIN eventmask DO Action
Triggers related to origin objects
Activation point
ON DEST eventmask DO Action
Triggers related to destination objects
END R;

Activation point. When a method is called, the system opens a transaction to execute it. Method execution always generates events at certain specific points within its associated transaction:
PRE triggers are activated immediately after opening the transaction, before method execution. They enable the system to check objects and the system state before operation execution, to extend the method by prologues and so on.
POST triggers are executed immediately before the commit of the corresponding transaction. Their aim is to analyze the database results, to assert the new state of the data base, to undo (rollback) the modifications performed by the operation, to add more computation, to chain with further actions and so on.
AFTER triggers are executed after the validation of the transaction. They can be used to notify the validated operation and to take historical information, to chain with further actions and so on.
ERROR triggers are executed if the transaction is aborted, as an exception mechanism. They allow alternative action to be taken.
Eventmask. This is a condition defining when events are to be taken into account. Eventmasks are first-order logical expressions. A priority level is an integer which determines the execution order when multiple triggers are simultaneously fired.

```
EVENT eventmask = Condition, PRIORITY nn;
```

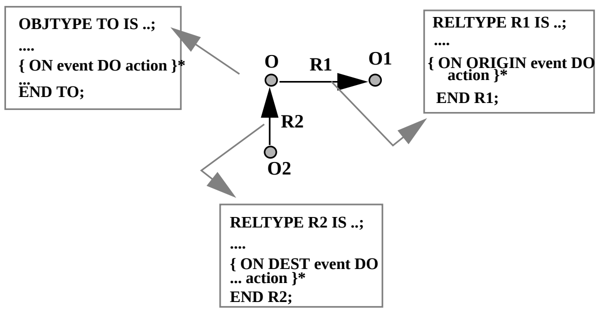
An *Action* is any program in the Adele Language.

4.5.1.1 THE TRIGGER EXECUTION MODEL

When an event occurs, the system executes the triggers whose eventmask is true. The instance on which a method is invoked is called the *reference instance*. This instance plays a central role because the triggers to be activated are found in its type definition. The reference instance can be either a relationship or an object.

If the reference instance is an object, the triggers to be executed are found in that object type definition, and also in the *relation type* definitions associated with that object. If the object instance O is the origin of $O|R_1|O_1$, and the destination of $O_2|R_2|O$. for any method on O the triggers will be found in the:

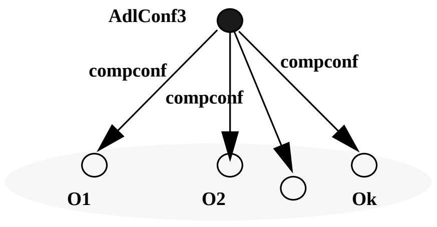
- definition of the O object type (ON clause),
- definition of the relation type R_1 (ORIGIN clause)
- and, definition of the relation type R_2 (DEST clause)



This mechanism is used to propagate events along potentially complex graphs. The execution of a trigger or a method can create new sub-transactions because other triggers can be activated in reaction to new events. These new sub-transactions are nested in the initial one.

4.5.1.2 AN EXAMPLE OF CONSISTENCY CONTROL

Assume that *AdlConf3* is a configuration and that its type is called *TConf*. *TConf* has, among its attributes, a *state* attribute which indicates the configuration state, e.g., stable, in-development, in-test, in-documentation. If the configuration is stable (state = stable), its components cannot be modified. Its components are objects (instances of heterogeneous types, e.g., programs, binaries, documentation, etc.) and are related to *AdlConf3* via associations, instances of *compconf* relation type.

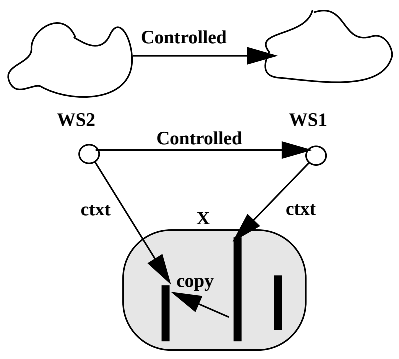


Triggers can be used to prohibit modifications on the configuration components. We can use the *compconf* relation type to define a trigger related to the destination objects, i.e., the configuration components. The transaction will be aborted if there is a modification attempt:

```
RELTYPE compconf IS ...;
      DOMAIN !type = TConf --> !type = Object
      POST ON DEST modified DO {
          IF (~!O%status == stable) THEN ABORT;
      }
END compconf ;
```

4.5.1.3 COORDINATION PROTOCOL

Assume that WS1 and WS2 are coordinated by a *controlled* policy (see 4.3.3.1), with condition “the master object get status = valid”, and that X is a shared object



```
RELTYPE ctxt ;
1     DEST ON modified DO updated (!O -d !D !cmdline) ;
END ctxt ;
RELTYPE controlled ;
2     EVENT to_resynch = updated and ~!d%state = valid ;
3     DEST ON to_resynch DO
          copy (!d, ~!O->ctxt (!name = !d)) ;
END controlled ;
```

The meaning of Line 1 is that each time a component is modified (it is the destination of a *ctxt* relationship), the method *updated* is propagated to the origin (the WS itself); here!D = X and !O = WS1. Thus, the *controlled* relationship is alerted by the *updated* event on its destination; event *to_resynch* (line 2) becomes true if the state of the modified object (now called !d as being option -d of the *updated* method) has also state = valid. If that is the case, line 3, a copy (and possibly a merger) is performed on the WS2 object having the same name as the modified object. ~O->ctxt returns all WS2 components, and (!name = !d) select those (unique here) whose name is X (!d).

The WS manager was implemented using just a few pages of triggers.

4.5.1.4 CM BASIC SERVICES.

It has been repeatedly argued that CM basic services include reporting, auditing, change report, accounting and so on. Our process engine has been designed to implement these services quite easily.

For purposes of illustration, the Work Space Manager was programmed in a few pages of triggers, our generic change control in 2 pages and 3 days' work, and the change set approach (inspired by ADC) in 3 pages. A state transition diagram defining the successive state of an artifact typically corresponds to one page of programming.

4.5.1.5 EVALUATION

The Adele Process Engine has been designed to be able to support almost any specialized process formalisms. For example, simple state machines such as those in CaseWare, petri nets like those in ProcessWeaver [Fer93]; notifications such as those in DSEE [LCS88] and command encapsulation of the ClearCase type are easily supported. Goal-oriented approaches like those in Marvel [KBS90] can also be implemented but more arduously.

The use of our triggering mechanism shows that it is possible to model and automate a wide range of software management strategies. Consequently, the system is highly adaptable and can be customized to specific users' needs.

4.5.2 HIGH LEVEL PROCESS SUPPORT

It has been found that triggers are powerful but that they fragment of the process into a large number of tiny definition types, making it difficult to get a clear view of a complete process description. Trigger are not only, a process modeling formalism, but a process engine defined to support different process formalisms. A process modelling language should allow users to describe different software management policies in terms of concepts more abstract than objects, triggers and associations.
For the design of our high level process formalism, we identified the following requirements:

- coordination and cooperation are ill supported, although they involve the most work; they are also highly uncertain.
- The complete process is very complex; the support should present a simplified view to all human agents.

Consequently, we have defined a process formalism, called *Tempo*, based on the concepts of *role* (viewpoint) and *connection* (coordination control). This language is presented below.

4.5.2.1 THE ROLES AND VIEWPOINT APPROACH

Each agent class have different activities. Examples are a project manager's activities (project status control, planning), a configuration administrator's (selecting components for a given baseline), a software process engineer's (defining data models, process models, enaction control), a development engineer's (editing, compiling) and testers' (running test cases, writing problem reports). All have a very different perception of the same reality. Processes should respect and even reinforce these different perceptions, proposing suitable representations.
It would seem worthwhile to respect and even emphasize this fact; this is the goal of the role concept. A role gathers together all entities with the same static and dynamic definition in a given process. A role looks very much like a type: it is the description of the common characteristics of a set of objects, and is called its extent. The original point is that a role

extent may contain objects of different types, and that objects of the same type may belong to different role extents in the same process.

The viewpoint approach features a number of conceptual properties.

- From the process engineer's standpoint:
  A new process can be defined almost regardless of previously existing processes. The size and complexity of a process definition is substantially reduced.
- From an agent's standpoint:
  A human agent, when engaged in a process, plays a role (human-agent) in this process. The process description is really the agent's view of the process, since it contains everything that is pertinent for executing that activity, and no more.
- From an evolutionary perspective:
  The modification or creation of a role is only propagated in the process definition where it appears. Any new process definition can be defined, modified or removed at any moment.

Interested readers should refer to [BM93].

4.5.2.2 CONNECTIONS

We define a Work Environment (WE) as a process occurrence. It is a tuple:

```
WE =  (WS, PM, Tools, User)
```

where WS is a Work Space, i.e., the set of object instances the process will perform on, *PM* is the process model, *Tools* the tools that will be executed on the WS objects, and *User* the user(s) allowed to work in this Work Environment.

A WS has the properties presented before: by default any change performed in a WS is local to that WS (isolation property). The PM describes only what is done in the WS. This property makes it easier to define a PM.

Our ambition is to do this in such a way that:

- Synchronization is fully transparent (granted by default).
- Coordination between two processes is explicitly defined, at the model level, outside these two process descriptions. It is implemented in such a way that independence of process descriptions is (almost) respected.
- Cooperation is an intrinsic part of the process description. This will include the description of cooperation between sub processes.

Currently, most process formalisms provide support for only some of these dimensions; SCM systems usually only support synchronization and most process managers only propose cooperation. For example, Process Wise Integrator 'interactions' [Rob94], [SW94] implement an explicit client-server interface. This is only one kind of cooperation, Process Weaver only supports message passing which is another kind of cooperation; formalisms relying on a database only have explicit locking of objects, i.e., basic synchronization.

Synchronization being provided by the Adele kernel, we propose the concept of *connection* to implement both coordination and collaboration. A connection is a link between two roles allowing to communicate, either through data flow, status checking, notification or message passing.

To illustrate how role collaboration can be defined, let us imagine the following scenario from the example provided in 4.5.1.2. When a new release of a given software product must be developed, a process, called *release* is created. An arbitrary number of *development* WE and a single *design* WE can collaborate in this release process. Each *development* WE can change configuration components (role *conf*) and have read access to the design objects (role *design_doc*).

The synchronization between *development* WEs is as follows: when a given module M receives the *ready* state in a *conf* role, M copies in each *conf* role of all other *development* WE must be merged, and their owner notified. If a design document D in a *design_doc* role is changed in a *design* WE, the new D version automatically replaces the previous one, and notification is sent to the WE user.

A module M receives the *available* state, in a release WE, only if all its copies have the *ready* state. When all modules have the *available* state, a *validation* WS can be created.

```
TYPEPROCESS release ;
      ROLE implement = development ;  -- Implement sub WEs.
      ROLE design = designing;        -- Design Sub WE
      ROLE valid = validation ;       -- Valid Sub WE .
      ROLE design_doc = document ;{ ..}-- Indicate the process for these documents.
      ROLE components = module ; {    -- Contains all conf modules of that WE.
1         EVENT ready = (implement.conf.*.state := ready) ;
          ON ready DO {
2             IF implement.conf.!name.state == ready THEN
3                 state := available ;
4             IF *.state == available THEN new valid ;}
      };
5     CONNECTION dev_conf IS notify, merge ;
6         CONNECT implement.conf WITH implement.conf ;
7         EVENT   notify_when  = ready ;
                  resynch_when = ready ;
      END ;
      CONNECTION md_design IS notify, resynch ;
8         CONNECT design.design_doc WITH implement.design_doc ;
          EVENT   notify_when  = ready ;
                  merge_when   = ready ;
      END ;
END release ;
```

- Line 1 stipulates that when, in any implementation WE (role conf), any object (*) enters the *ready* state, the event *ready* occurs.
- The line 2 sentence *implement.conf.!name.state* evaluates the set of values of the attribute *state* of the object that produced the *ready* event, as found in the *conf* roles of all impelment processes. Operator “==" means set equality. Line 2 means that each copies of the object *name*, in all development WE have the *ready* state.
- Line 3 says that the M copy, as found in the release WE must take the *available* state.
- The line4 expression **.state* returns all state values of all objects in the component role in the release process. Line 4 means that when all module get the *available* state, a *valid* role (i.e., a validation WE) must be created.

Some basic behavior is provided in a standard library, for example *notify, resynch* and *merge* in this case. Using standard inheritance mechanisms, each connection can reuse these process fragments (line 5), and redefine, for instance, the events for which some behavior must be executed. Line 7means that notification and resynch must happen when the object becomes *ready* . The CONNECT clause expresses which pair of role must be coordinated. Line 6 means that, for a given release process, the *conf* roles between each pair of *development* WEs are connected by a *dev_conf* connection (peer to peer connection). Line 8 stipulates that each *design* WE (role design_doc) are to be coordinated with each *development* WE (role design_doc) by an *md_design* connection (master/slave connection). The father/son connections in example 4.5.1.3 (SELF.component with implement.conf; design with design.design_doc and SELF.design with implement.design_doc) are defined in the same way.

Thus, depending on the connections, the activity performed inside a WE may or may not interfere with other activities carried out in parallel during the software process.

4.5.3 EVALUATION

The translation between Tempo and Adele is quite clear. A process model is associated with a WS type and a specific schema. The PM is defined in this schema as the behavior of the WS. The connection concept is directly mapped to Adele coordination relationships (controlled). Roles are implemented using complex objects, a multiple schema approach, and semantic relationships. However, this does not yet correspond to full implementation of the role concept.

Tempo has not been experimented with yet. More research is needed before an implementation can be made available. Efforts have been made instead to design and implement the process engine and in a commercial level WS manager, since it was felt that these are the major components needed for implementing any process support environment.

4.6 EVALUATION

The Adele project has now been running for ten years. Its earliest commercial versions were distributed, directly by our university, to large companies from 1984 onwards (notably Sextant and Bull, the former for the development of embedded systems, including the Airbus series and the Ariane rockets; the later for the support of its proprietary operating systems). They have been heavily used ever since. Major SEEs have been developed on top of the Adele system. Examples are afforded by Palas (Sextant), Emma (Bull), SDE (Matra), CajMarie (Ericsson). Currently hundred of licences are in daily use. Since 1993, Adele has been distributed and developed by the Logiscope/Verilog company. Sales may therefore be expected to increase in the near future[3].

Throughout this period the practical problems encountered in the setting up and use of SCMs in critical environments have been studied in collaboration with our clients. Roughly speaking, the customer’s requirements may be said to be as follows:

1. Efficiency, efficiency, efficiency,

2. Transparent, open, evolutive, non-intrusive.

3. Better product modelling, versioning facilities.

3. Adele is distributed by Logiscope Technology 3010 LBJ freeway, suite 900 Dallas, Texas U.S.A. Verilog SA 52 av. Aristide. Brian 92220 Bagneux France

4. A custom-built SEE is too expensive. Even its design is beyond our capability. Please help us.

Seen from a technology provider’s standpoint, these requirements are interpreted as follows:

(1) Efficiency becomes truly critical as soon as large-scale development is under way. Selecting the platform to start with is critical.

(2) The SEE to be built must not be constrained by the underlying SCM tool. Naming conventions, data model constraints and so on should be invisible from the outside. WS policy definition and implementation were found to be very costly in most SEEs. Process support, when lacking, means that huge amounts of code have to be written in order to implement the desired process. One ends up with a very expensive and inflexible environment, and one which is difficult to maintain and evolve.

(3) There is a need for a better description of product structures, for controlling them and for defining product evolution constraints. Versioning capabilities are currently too primitive.

(4) The last point is totally at odds with the others. Most users need an immediate start-up solution, with the insurance they will be able to make it evolve as soon as they understand their own process better. In this respect, a good SCM should be seen as a way to gradually understand, define, improve and adopt processes in an evolutive way.

Most of our customers, until recently, have been large companies, with critical software. They have used Adele as a kernel, building dedicated SEEs on top. Generally, customers liked the great flexibility and power of the tool, its capabilites for modelling complex products, integrating and interfacing with tools and methods, and its programming facility for implementing specific interfaces. They disliked the strict product model, the overly constraining automatic configuration selection, and the lack of associated services. We, on the other hand, have been disappointed to note that automatic configuration selection was not used as much as expected.

These remarks, and others, have provided the incentive to make the product evolve as an efficient and highly customizable kernel. Process capabilities are currently the major focus of our research, as well as more general and powerful data models (points 1, 2, 3).

More recently, with wider marketing of the product, emphasis has been placed on predefined high level services (point 4), such as a generic graphical interface, the WS manager, and the design of a few predefined SEEs dedicated to certain types of customers. Indeed, it has been found that the cost lies much more in defining the services to be provided and their interconnection, than in their implementation. In this respect, our process support has proved to be highly effective, with most implementations being a matter of just a few pages of program. The WS manager itself is fully implemented in 4 pages of triggers. This may be compared with the thousands of lines of code customers have had to write to realize a single specific WS implementation.

4.7 CONCLUSION

We have tried to show that an SCM system should rely on three major components: an object manager controlling the Repository Space (RS) where software artifacts and their versions are stored, a Work Space manager controlling the content and activities performed in the Work Spaces (WSs), and finally a Process Manager controlling the tasks and activities performed in both the WSs and the RS.

We think there is a need for:

- a powerful data model including an explicit versioning model, controlling the RS.
- a WS model, which is much more powerful than a bare file system.
- explicit mapping between RS and WS including inter-WS collaboration.
- an explicit Process Model, controlling both the RS, the WS and their relationships.

We have presented an approach based on an advanced database, supporting a specific data model, sub databases, WS control and process control. Commercial SCMs first used the bare file system with no data model at all. Currently, advanced SCMs use databases and offer some data and process modelling capabilities. But much progress needs to be made in both directions. Adele is specifically designed for the support of a specific data model and full process support, but the ideal has not yet been reached.

We claim that future configuration managers will have to start from more specialized databases, providing high-level data modelling, full process support and the specific services needed by configuration management. We expect our work on Adele to provide both a practical database and deeper insight into the class of database we shall need in the future.

REFERENCES

[AS93] Paul Adams and Marvin Solomon. An overview of the CAPITL software development environment. Technical Report TR 1143, Computer Science Department, University of Wisconsin-Madison, April 1993.

[BE86] N. Belkhatir and J. Estublier. Experience with a data base of programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 84–91, Palo Alto, California, December 1986.

[BK93] N.S. Barghouti B. Krishnamurthy. Provence: A process visualization enactement environment. *ESEC93, 4th European Software Engineering Conference.Garmish, Germany. Springer Verlag.,* Septembre 1993.

[BM93] N. Belkhatir and W. Melo. Supporting software maintenance in tempo. In *Proceedings of the Conference on Software Maintenance*, Montreal Canada, 1993.

[CW93] M. Cagan and A. Wright. Untangling configuration management: Mechanism and methodoloy in cm systems. In *Proc, 4th International workshop on Software Configuration Management*, Baltimore, May 1993.

[Dar91] Susan Dart. Concepts in configuration management systems. In Springer Verlag Peter H. Feiler, editor, *International Workshop on Software Configuration Management SCM3*, pages 1–18, Trondheim, Norway, June 1991.

[Dit89] K.R. Dittrich. The Damokles database system for design applications: its past, its present, and its future. In K. H. Bennett, editor, *Software Engineering Environments: Research and Practice*, pages 151–171. Ellis Horwood Books, Durhan, UK, 1989.

[EF89] Jacky Estublier and Jean-Marie Favre. Structuring large versioned software products. In *Proceedings of IEEE COMPSAC 89*, pages 404–411, Orlando, Florida, September 1989.

[Est85] J. Estublier. A configuration manager: The Adele data base of programs. In *Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, Harwichport, Massachusetts, June 1985.

[Est88] Jacky Estublier. Configuration management: The notion and the tools. In Springer Verlag, editor, *Proc. of the Int'l Workshop on Software Version and Configuration Control*, pages 38–61, Grassau, Germany, january 1988.

[Fei91] P. Feiler. Configuration management models in commercial environment. Technical Report CMU/SEI-91-TR-7 ADA235782, Software Engineering Institute (Carnegie Mellon University), 1991.

[Fel79] S. I. Feldman. Make: a program for maintaining computer programs. *Software–Practice and Experience*, 9(4):255–265, April 1979.

[Fer93] C. Fernstrom. Process weaver: adding process support to unix. In *Proceedings of Second Intl. Conference on the Software Process ICSP2*, pages 12–26, Berlin, Germany, 1993.

[HL82] R. L. Haskin and R. A. Lorie. On extending the functions of a relational database system. In M. Schkolnick, editor, *sigmod*, pages 207–212, Orlando, FL, June 1982. acm.

[HN86] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[Kam87] R.F. Kamel. Effect of modularity on system evolution. *IEEE Software*, January 1987.

[Kat90] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, [12] 1990.

[KBGW91] Won Kim, Nat Ballou, Jorge F. Garza, and Darrell Woelk. A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51, 1991.

[KBS90] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Preliminary experience with process modeling in the Marvel software development environment kernel. In *Proc. of the 23th Annual Hawaii Int'l Conf. on System Sciences*, pages 131–140, Kona, HI, January 1990.

[KK90] David G. Korn and Eduardo Krell. A new dimension for the unix file system. *Software—Practice and Experience*, 20(S1):19–34, June 1990.

[LC84] D. B. Leblang and Jr. Chase, R.P. Computer-aided software engineering in a distributed workstation environment. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, Pittsburgh, PA, may 1984. acm, acm.

[LCM85] D. B. Leblang, Jr. Chase, R.P., and Jr. McLean, G.D. The DOMAIN software engineering environment for large scale software development efforts. In *Proceedings of the 1st International Conference on Computer Workstations*, pages 266–280, San Jose, CA, November 1985. IEEE Computer Society.

[LCS88] David B. Leblang, Robert P. Chase, and Howard Spilke. Increasing productivity with a parallel configuration manager. In Springer Verlag, editor, *Proc. of the Int'l Workshop on Software Version and Configuration Control*, pages 21–37, Grassau, Germany, january 1988.

[Leb93] C. Leblang. Clearcase. In *Proc. of the 4th International Workshop on Software Configuration Management*, Baltimore, Maryland, may 1993.

[Mic88] Sun Microsystems, editor. *Introduction to the NSE*. Network Software Environment: Reference Manual. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, USA, part no: 800- 2095, march 1988.

[Mun93] B. P. Munch. Uniform versioning: The change-oriented model. In *Proc. of the 4th International Workshop on Software Configuration Management*, pages 232–240, Baltimore, Maryland, may 1993.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Per87] Dewayne E. Perry. Version control in the Inscape environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, Monterey, CA, March 1987.

[Rob94] I. Robertson. An implementation of the ispw-6 process example. In *Third European Workshop On Software Process Technology EWSPT94*, february 1994.

[Roc75] M. Rockhind. The source code control system. *IEEE Trans on Soft. Eng.*, SE-1(4):364–370, Dec 1975.

[RPD86] J.M. Neighbors R. Prieto-Diaz. Modules interconnection languages. *The journal of systems and software*, pages 307–334, 1986.

[Sci91] E. Sciore. Multidimensional versioning for object-oriented databases. *Proc. Second International Conf. on Deductive and Object-Oriented Databases*, December 1991.

[Sha84] M. Shaw. Abstraction techniques in modern programming languages. *IEEE Software.*, pages 10–26, October 1984.

[Sno93] R. T. Snodgrass. *An Overview of TQuel*, chapter 6, pages 141–182. Benjamin/Cummings, 1993.

[SW94] Jin Sa and B. Warboys. Modelling processes using a stepwise refinement technique. In *Third European Workshop On Software Process Technology EWSPT94*, february 1994.

[Tho89] I. Thomas. Pcte interfaces: Supporting tools in software engineering environments. *IEEE Software*, November 1989.

[Tic82] Walter F Tichy. Design implementation and evaluation of a revision control system. In *Proc.6th Int. Conf. Software Eng., Tokyo*, septembre 1982.

[Tic88] Walter F. Tichy. Tools for software configuration management. In *Proc. of the Int. Workshop on Software Version and Configuration Control*, pages 1–20, Grassau, january 1988.

[Wat89] Richard. C. Water. Automated software management based on structural models. *Software Practice and experience*, 1989.

[Wie93] D. Wiebe. Object-oriented software configuration management. In *Proc 4th Software Configuration Management workshop*, Baltimore, May 1993.