The evolution of configuration management and version control

by Vincenzo Ambriola, Lars Bendix and Paolo Ciancarini

The activities of configuration management and version control are common to a number of engineering tasks. These activities are particularly important for software engineers, since during most of a system lifecycle they have to deal with a growing number of versions of a single component, and to rebuild the complete system in different ways using different components. These tasks are repetitive and trivial, and they require a lot of manual work and accuracy. In this paper, we show how the problem of automating these activities has been solved in a number of software development environments. We describe the evolution of systems for configuration management and version control from simple stand-alone tools, such as *make* and SCCS (based on an underlying file system), towards more integrated systems based on a project database.

1 Introduction

During the development of a project for building a (software) system there are many activities that must be carried out consistently. At the top level, the lifecycle must follow a number of development phases according to a particular method. Each phase produces a plethora of documents that must be co-ordinated. Each document can pass through a number of different versions that reflect the evolution of the project. Increasingly, a programmer can rebuild the same system in different ways, using different modules or versions.

The aim of this paper is to show the evolution of systems and tools used in software development environments (SDEs) for two particular but very common activities: configuration management (CM) and version control (VC).

This work is a careful examination of the evolution of these two concepts, rather than a complete comparison of some well chosen development systems; we are mainly interested in revealing the trends and the different stages in the history of systems for handling the activities of CM and VC. For this reason, the paper is neither intended to be a survey on the subject nor an in-depth description of the systems. Instead we will concentrate on the problems and the solutions that were given in a small number of tools and environments, in order to show the abstract ideas underlying the different implementations.

The exposition of tools and methods is centred on two basic tools and three technological generations. The tools are *make* and SCCS, chosen for their spread use in the system programming world based on the UNIX operating system. The three generations are exemplified by some fairly well known software engineering environments: RCS for the first generation; Cedar and Gandalf for the second one; and Adele for the third one. As we will show, these generations correspond to the use of different basic methods and technologies for the support of the activities of configuration management and version control.

2 Basic concepts

Software engineering is still a young discipline and, as such, it has not yet developed a consolidated terminology. Consequently, comparisons are difficult, because different people often use different terms to cover basically the same ideas and maybe the same names for different ideas. In this Section we give a number of definitions for the most important terms used in this paper. The definitions are informal, since we do not pretend to establish any standard terminology; we just want to state some concepts which will make the presentation of the evaluation and the different systems more coherent and possibly less confusing.

- **Component:** a component is the basic unit from which a system is constructed. It can be either atomic or composed from other components. Samples of components are, among others, the blueprints of a machine, the chapters of a book, the parts (modules) of a program.
- **Configuring:** components are put together to form a system. Configuring of systems can be separated into two related parts: the generic *description* of those parts from which the system is composed, and the actual *instantiation* of the system generated from this description.
- **Versioning:** a component may evolve while time

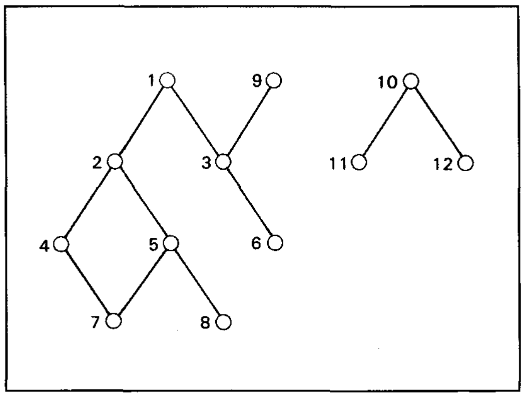

Fig. 1 The description of a configuration

passes and the project goes on. These changes occur because of bug fixes, maintenance, further development etc., and the resulting components are called versions, revisions, variants or deviations. For the sake of simplicity, all these synonyms are unified in the term *version.*

- **Configuration management:** this is the art of controlling the configuring of systems. The goals of configuration management are to facilitate the fast instantiation of a system and to enforce restrictions on the possible ways to describe a system.
- **Version control:** this is the art of controlling the versioning of components. The goals of version control are to facilitate the efficient retrieval and storing of many versions of the same component, and to enforce restrictions on the evolution of a component so that such an evolution is observable and controllable.
- **Software development environment:** the environment in which CM and VC are performed and integrated with each other, and with other tools and resources, to support the creation of software by many programmers.

3 Configuration management

During the development and maintenance of a large and complex system, it is often necessary to instantiate it again and again, both because its components evolve and because there may be a need of configuring the system in slightly different ways.

Every system has an internal structure, i.e. it has parts, and relations among parts. A very common (abstract) relationship is 'A depends on B', meaning that a component A is hierarchically subordinated in some way with respect to a component B. This relation can be represented by a directed acyclic graph, as shown in Fig. 1, in which it is implicit that lower nodes depend on higher nodes, if there is a path that joins them.

The tools that support configuration management of software systems have two main objectives: saving time in building the configurations and taking gain from the underlying structure of a configuration. Saving time concerns minimising the cost of instantiating a software system in terms of CPU (compilation) time as well as in terms of human and elapsed time. Taking gain from the underlying structure means introducing explicit constraints and abstractions, which help in making system description conceivable and controllable.

There are several reasons for addressing these objectives. Even if a system can always be instantiated (and implicitly described) by putting the proper command sequence in a batch file, this approach is not flexible enough. Using a batch file, the system is built from scratch each and every time, a rather time-consuming process for a large system. Furthermore, we will have to manually customise the batch file to instantiate the system in (slightly) different ways. Moreover, worst of all, the description of the system will be unstructured and drowned by irrelevant details and, as such, incomprehensible and vulnerable to errors.

3.1 make

make [1] was built in the mid-1970s as a tool for CM. Its main feature is the ability to automatically issue the shortest command sequence needed to instantiate a given system, based on a simple user-supplied description of its dependencies.

The description of the component dependencies of a system is explicitly given by the programmer, and it is contained in a normal text file, given in a fixed format. *make* reads this file and then constructs a (directed acyclic) dependency graph, from which it checks the consistency of the system, based on temporal attributes of the components.

The basic way to describe a dependency is to state a target file and the list of all the files on which it directly depends, as in

```
module5 : module 7 module8
     commandsequence
```

where **module5** depends directly on **module7** and **module8.** This can be considered as a rule, stating that if **module5** is out of date with respect to its dependencies, the associated **commandsequence** has to be issued to bring **module5** up to date and make the system consistent again. No action is taken if **module5** is not out of date, as the system is already consistent.

make thus satisfies the requirement of building only the necessary parts. However, the construction of the description is not optimal, as it just consists of the rules, whereas the graph is internal to *make.* Furthermore, parameterisation has to be done manually, by directly and completely describing different systems.

4 Version control

During the project development and maintenance a component will undergo changes. The set of changes (operations on a text, usually done by an editor, cut, paste, insert, delete etc.) that transform a version of a component into a (successive) version will be termed *history step.*

In Fig. 2 a history step transforming a version into a new one is depicted; the transforming operations are enclosed in a triangular box to suggest the form of a *delta,* a common name for history steps in many systems. A sequence of history steps forms the development *history* of a component.

For many reasons it is convenient to keep, or to be able to recreate, a component in its previous forms and not just in the current one. Version control is the activity of managing the history of a component addressing two main prob-

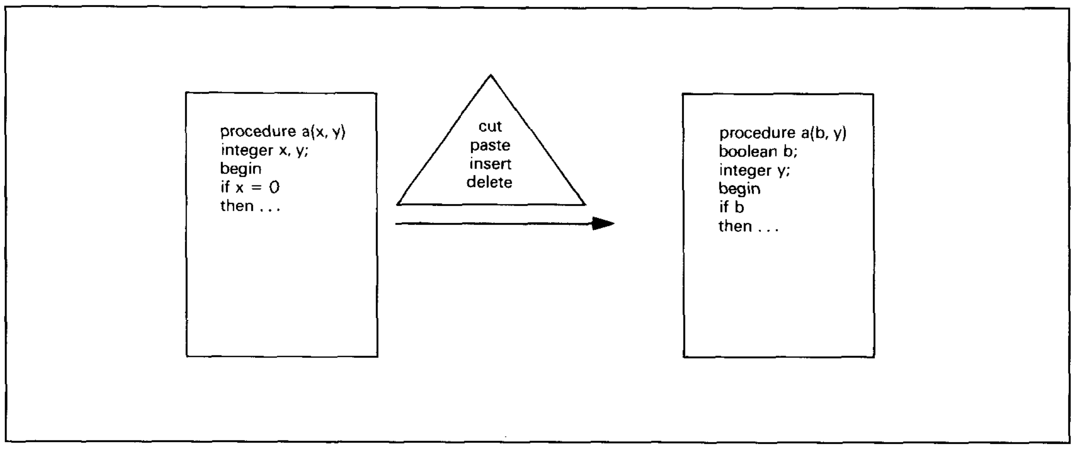

Fig. 2 A history step

lems: saving secondary storage space, and imposing structure on the development history itself. Saving space is mostly a technical matter, in which all the information is compressed in as little space as possible. Imposing structure on the development history introduces a means of restricting the way in which a component can evolve (history steps cannot be arbitrarily complex) and facilitates the retrieval of any wanted version of a component.

There are several reasons for these objectives. Although it is always possible to manually obtain the *back-up* effect, it is usually a very slow and cumbersome process, both space-inefficient and error-prone. Furthermore, as the number of different versions of a module begins to grow beyond certain limits, it becomes increasingly hard to grasp (and remember) how newer versions have evolved from older ones.

4.1 SCCS

SCCS [2] was built in the early 1970s as a tool for version control. Its main features are to manage memory space efficiently, when keeping several versions of the same module, and to facilitate the retrieval of any specific version.

The basic idea behind SCCS is that the history of a component has a linear evolution and always evolves from the previously newest version. This linear evolution is a long line of history steps, shown in bold in Fig. 3, where new versions are always added to the end of the bold line.

The different versions of a module are represented as boxes, whereas the modifications that have to be applied to one version to get the next one are represented by an arc between the two versions, labelled by a *delta*. A delta is a function that can be applied to a module obtaining a new version of that module. A delta is represented by a sequence of commands of two kinds only: insertions and deletions of entire lines. This means that changing a few characters of a line implies the deletion of all the line itself and the insertion of a new line.

SCCS save space by keeping only the first version and all the deltas. It is able to recreate any version by applying the necessary deltas to version 1.0.

In SCCS the modules that come out from major/important changes (the modules are called *releases*) are distinct from modules coming out from minor/less important changes (these modules are called *versions*). This distinction induces a double-level method of grouping together versions into more convenient classes (reflected by the version numbering).

However, this simple way of organising releases and versions is inadequate in many cases, e.g. for bug-fixes. In fact, bug-fixes cannot just be appended to the end of the main line if the module continues its evolution into a new release. This is an unfortunate situation, as maintenance of older releases happens quite often in real life. Therefore, in SCCS new versions can be appended to any release and not just to the latest. This results in a tree-like model, as can be seen in Fig. 4.

A new version of a component under SCCS control is created in three steps: first, a copy of the desired version of a module is retrieved; then changes are made to the copy using a normal editor; and finally, the changed copy (or rather the delta) is stored.

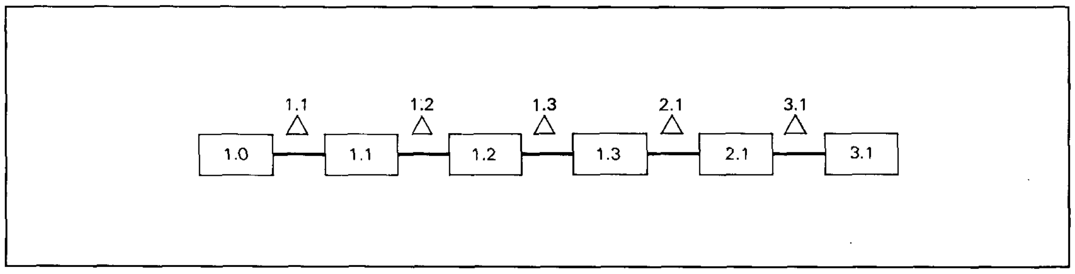

Fig. 3 Simple line model

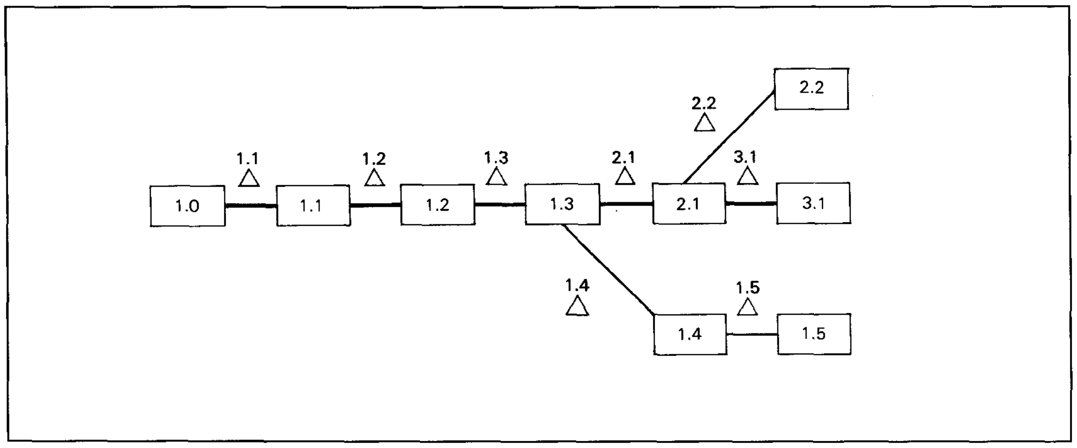

Fig. 4 The SCCS model

In SCCS a delta is treated as a function that can be applied to any version, even out of the original context. Any desired function can be obtained by functional composition of deltas. They can also be used in a rudimentary way to *merge* versions, i.e. to obtain a new version joining the modifications to two older versions.

SCCS provides a rudimentary way of storing additional information about the modules. It is possible to place some identification keywords within the source module, so that the object module refers back to the right version of the corresponding source module. In all, there are about a dozen keywords, such as **release**, **level**, **date**, and so on.

For maintaining the history of a module evolution, SCCS automatically keeps information of who made a change, when it was made and what the change was (by the delta). The reasons for the changes are asked from whoever has made them.

It is interesting to note that, in order to mask some deficiencies of UNIX, SCCS provides some mechanisms for protecting the files put under its control. Moreover, it implements access control to releases by means of locks and passwords.

5 Integration of configuration management and version control

In this Section we present a historical framework for the evolution of tools for CM and VC and their integration, both with respect to each other and with respect to the other tools available in a software development environment. In this evolution we have identified three different stages or generations.

In the following, these three generations will be described and exemplified by some well known systems. These systems have been chosen because they make it possible for us to stress the evolution of some major features, and because they are fairly widespread and additional literature for a more detailed study is easily obtainable. We thus do not pretend to give a thorough description of the systems, nor to have made a representative choice of systems.

Talking about the integration of CM and VC, we also want to shift the attention from the single-user perspective of *make*, towards the problems which arise when many people have to work together on developing software products. This introduces new topics like the ability of avoiding conflicting/unauthorised updates, which was partially handled in SCCS, or like the distribution of the components in different directories or on different file servers, which was not addressed at all in *make*. (It has, however, been handled by Fowler [3].)

5.1 The first generation (RCS)

In the first generation environments the integration of the two activities was achieved by glueing together two independent tools, without any real attempts either to enhance the underlying models or to have a uniform model for the whole SDE. In fact, the integration simply creates a more powerful tool, able to perform both tasks.

The RCS project [4] started in the early 1980s as a further development of SCCS, with the purpose of solving some of the drawbacks of the fuzzy model behind that system. Furthermore, RCS provides a simple interface to *make*, in the naïve attempt of integrating the activities of CM and VC. Note that RCS is primarily a tool for version control. The integration of the two tools is entirely based on the underlying file system and operating system.

RCS enhances the SCCS model by making explicit the tree structure and increasing control by the system. The set of versions of a component (called a *revision group*) is regarded as a tree structure, consisting of a trunk and (possibly) some branches. In RCS every node on the trunk can have several branches. This feature adds one more level to the grouping (numbering) of the versions, as can be seen in Fig. 5. The trunk is the bold line.

Fig. 5 also shows that, as branches are allowed to evolve too, a version identifier is obtained as the concatenation of four numbers: the release, the version in the release, the offspring of the version and the version of the offspring.

RCS is used in the same way as SCCS by checking out a copy, making changes to it and checking back to the changed copy. Moreover, RCS has a facility for locking the files, which ensures protection against several programmers changing the same version at the same time. Checking out using the 'l' switch will lock the file from being checked out until the locking programmer has checked in his new version.

The RCS attribute scheme is very similar to the SCCS

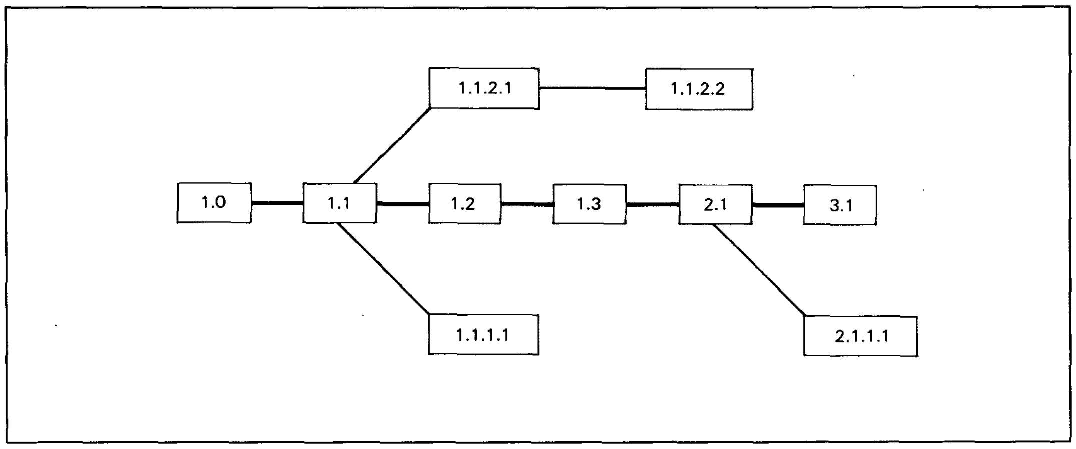

Fig. 5 The RCS model

one. There are keywords for **name**, **revision**, **date**, **time**, **state**, **author** and **locker**. Apart from **state**, which is user-definable, all attributes are fixed.

To handle CM, RCS, provides an interface to *make*. A system description is a set of versions, belonging to different revision groups, together making up a complete system. Each version is selected according to a set of criteria. The supported selection mechanisms are based on the **default**, **release**, **state**, **author**, **date** and **name** attributes. If the instantiation is not consistent, *make* issues the proper command sequence to rebuild it.

With respect to RCS, *make* has been extended with an auto-checkout facility. When a file to be processed is not in the users' working directory, *make* attempts a check-out operation and then performs the required processing. The selection parameters to be used for this check-out operation can be passed to *make* either as parameters or directly embedded in the Makefile. However, if present, a working file will be used. According to this strategy, a programmer knows that RCS/*make* selects an instantiation consisting of the versions he has currently checked out and those checked in by other programmers.

RCS supplies an operation to merge two versions. For example, versions 2.1.1.1 and 3.1 in Fig. 5 can be merged into a new one, say, 3.1.1.1. This is done with respect to a common ancestor, version 2.1. If a fragment of code is found to be different in all three versions (i.e. 2.1.1.1, 3.1 and 2.1), an error is flagged, and user intervention is required to select the correct alternative.

5.2 The second generation (Cedar and Gandalf)

In the second generation tools are integrated with the rest of the software development environment. An important feature of this generation is the attempt of moving away from the file system towards a simple database containing all the components. Being part of an environment in which *software* is constructed, tools essentially deal with components as parts of programs.

However, the most important feature found in the second generation systems is that the components are not black boxes, but software objects with attributes; an external and uniform description of their contents can be given in terms of a module interconnection language (MIL).

5.2.1 Cedar: Cedar is an SDE which has been under development at Xerox PARC since the mid-1970s [5]. Cedar is the name of both the supported programming language and the supporting SDE. With respect to the CM and VC capabilities, it is based on the System Modeller.

The Cedar environment supports the development and use of experimental programs produced by experienced programmers in a distributed computing systems. It is an integrated system that supplies a set of predefined tools and services, but it is also open-ended so it can be easily customised with the addition of new tools. Cedar is a typed language with mechanisms for decomposing the design of a system into modules. There are two kinds of modules: *interfaces* and *implementations*.

Cedar relies on two concepts introduced for CM and VC. One is the *configuration*, which is an explicit description of both components and generic systems. The other is the *description file*, which contains the description of an actual system.

Configurations are an integral part of the Cedar language. They are used to describe components, hence acting as a module interconnection language. A configuration specifies how to resolve import/export between interfaces and provides a way to perform intermodular type-checking.

A Cedar interface can have more implementations. The possible ambiguity can be handled within the language when specifying the desired configuration. When used to describe a system, a configuration contains the logical decomposition of a generic system and is allowed to pick, for each interface, just one implementation among possibly a set of them. Like modules, configurations may import and export interfaces. In this way, configurations can be viewed as abstractions that allow flexibility in creating a system and, at the same time, they can be hierarchically ordered.

Outside the Cedar language, the description files serve two purposes.

- They help to retrieve all the files needed to rebuild a given system with the up-to-date versions.
- A description file serves as a snapshot of a system, being a precise description of how the system was constructed.

The description files can describe how to put together a system, choosing a physical file among the possible versions for each component used in the generic system. As a primary means of guaranteeing consistency of a system, the description file contains the description of the files needed to construct the system (with version numbers and creation dates) and the location of these files, possibly on other file servers.

Cedar supports three operations that operate on a description.

☐ When a user wants to use a system, he first runs a **bringover** command which takes the description files and copies all the referenced files onto his personal machine.

☐ When he has made changes to a system, he **storebacks** the description file, so that all the changed files mentioned in the description files are copied back to their original places, and the version numbers and creation dates of all of the copied files are updated in the description file (so it now describes a consistent version of the new system).

☐ The third one is a **verify** command which can be used to ensure that the contents of a description file in fact cover the files necessary to build the system. The description file must contain references to all the files used in actually building the system and should not contain superfluous references (these might be indications of errors). Like configurations, description files are hierarchical and can contain references to other description files.

In summary, configurations contain the logical description of a generic system, whereas description files serve as a connection between the logical world of modules (interfaces and implementations) and the physical world of files (paths and names). They give a physical description that both mirrors the logical one and is used to instantiate an actual system.

Apart from the description files, the Cedar mechanisms for VC are not very advanced. For each history step, every time a file is changed, the new version is stored and its creation time is appended to the filename. Versions are in fact immutable, since modifying a file that is also used by others could have unpredictable effects. No attempts are made to apply space-saving mechanisms, and there are no restrictions/structure at all on how versions are allowed to evolve.

Although many programmers can work at the same time in a distributed environment, they operate in rather isolated worlds. The description file only reflects one's own changes to the system and not those of others. If a programmer wants to incorporate other people's changes, he has to explicitly change the version information inside his own description file.

5.2.2 Gandalf: Gandalf [6] is an environment for the generation of SDEs. The generated SDE provides an integrated set of supporting tools for system composition and generation, incremental program construction and project management. It has been under development at the Carnegie-Mellon University since the late 1970s. What will be referred to in the following is the Gandalf Prototype (GP) environment.

In Gandalf, system composition and generation are two distinct but related aspects; the functional specification of a system and the control of system versions. The generated environment checks the consistency of system descriptions, provides automatic generation of system versions, maintains the consistency of the project state and guarantees appropriate access to the project database.

In the GP a system description embodies both static and dynamic aspects. The static aspects deal with import/export and intermodule type-checking, whereas the dynamic aspects control system composition and the strategy for checking the constituent components. System generation is kept distinct from system description and is performed only by request from the user.

The GP maintains a structured database of program fragments. All the user activity has to be done within this database. These is no notion of checking files in and out of the database, as in RCS. The *user* enters the database and performs all operations while navigating around *inside* the database.

In the GP the activities of CM and VC have been mixed together, so that the related tasks while building a system could be depicted by *and/or graphs*.

The *box* construct is used as a structuring mechanism for the modules database. It serves to group together related components and can contain other boxes and modules. Using the box concept, it is possible to create a hierarchical grouping of (*AND*) related components, much the same way as it is possible with UNIX directories. Navigation inside the database starts from the top-level box and continues entering one of its boxes or working on one of its modules.

A *module* is an object in the database. A module has two functions: one, very similar to the interface part, for example, Ada, is to state the terms to be imported/exported; the other is to give an implementation of an interface. It differs from the implementation part of Ada by allowing more implementations of the same interface to co-exist. Each implementation can exist in several (*OR*) versions, and when a system has to be generated, exactly one implementation and one version has to be chosen for each interface in the system.

In Fig. 6 the system **Test** is shown in the form of AND/OR graph. It is composed from both Type T1 *and* ProcP1 *and* VarV1. For ProcP1 either ImplA *or* ImplB can be chosen, and for ImplA the choice can again be among different versions.

The *AND* view of a module cannot be changed. Changes to the import/export items will result in a completely new module, with a new name for which new implementation(s) must be given. Moreover, the calling code that uses the new module must be consistently changed.

The user can generate a complete system or just a subsystem. To help choosing among different implementations and versions, exactly one of each has to be designated the *standard* one that will be chosen if not stated otherwise. This can be done by explicitly stating which implementation/version has to be used or by using the *default* clause of the implementation.

A preference hierarchy is introduced when composing a system, so that the versions reserved by the user will always be used. If no version of a component has been reserved, the default version will be used, and finally if a default version is not given, the standard version will be used.

To impose consistency on version control, the GP uses access lists to boxes for protection on a large scale. For

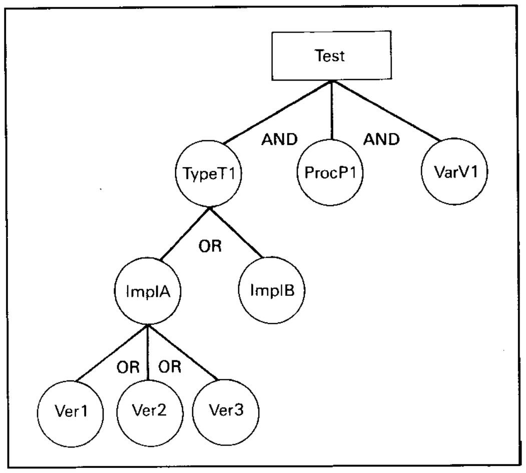

Fig. 6 AND/OR graph

protection on a small scale, there are reserve/deposit locks on boxes, modules and implementations. Versions are immutable, and to save space common code among different versions is shared. The system automatically keeps a history in the form of a log, recording by whom, when and why a change was made. The history can be queried at any time.

5.3 The third generation (Adele)

In the third generation the evolution of the database is almost complete. Systems of this generation are oriented towards high-level languages, where intermodular descriptions are available, and module composition information can be automatically extracted; there is no longer a need for a MIL.

The emphasis is on flexible and simple system composition, usually driven by an elaborated attribute scheme that controls the selection of components from a project database containing user modules and libraries.

Adele [7] was initially designed as a specialised database of programs for an experimental Pascal environment, with the focus on CM. Later on, it was redesigned as a general database management system for handling large-scale programs written in any language. Now it has evolved into a kernel for generating programming environments and has been developed as an industrial product. It has been under development in France since the early 1980s.

Currently, Adele is a database management system for program modules and supports multiversion software, relationships among software components, configuration management, and access control and protection.

In Adele the only requirement for the supported programming language is that components (called *objects*) have an interface and a body. Interfaces can exist in more versions, and bodies for each version of an interface can be in different versions (different implementations of the same interface) and revisions (evolution of the same implementation).

Each object in the database has attributes associated with it. Some of the attributes (such as **date**, **status**) are automatically maintained by the system, and others are set by the user. By using the keyword **attribute**, the user can define new attributes that are collected in a so-called *manual*. The manual is the part of an object where all the information about the object resides in form of the associated attributes.

For the design of attributes, Adele has adopted three principles:

- *uniformity*, i.e. the structure must be uniform for all types of objects;
- *locality*, i.e. the information relevant to the use of an object is attached to that object;
- *automatic derivation*, i.e. the system automatically uses all information that can be obtained or derived from the database.

The information that has to be supplied by the user is kept to a minimum.

The concept of *family* is used to group objects together. A family is a set of alike objects. For example, all the interfaces related to file handling make up a family. Families have a hierarchic structure that induces dependency relations among them. These dependencies can be represented as acyclic graphs. To denote a specific object in the hierarchy, it is necessary to specify the domain (family or subtree of families), the type (specification or implementation) and a set of (predicate, value) tuples that must be met by the objects attributes.

In Adele, a description is treated as an abstraction that hides from the user whether an interface is implemented by one body or by a (sub)family. Thus, instantiation is performed by selecting an implementation for an interface, giving actual meaning to an arbitrary node of the dependency graph.

The instantiation of a configuration is done by computing the composition list, selecting the right implementation from a user-defined description. This description can be either a full or an empty description. In the first case, the object to be chosen is described completely and no choice is left to the system; whereas, in the latter, no restrictions are given and the system is left to choose the object which is specified to be the default object.

Descriptions can also be given in terms of attributes that select an implementation. This selection is either *imperative*, *conditional* or *default*.

Imperative selection is the strongest way to select an object, as it says that the object *must* meet the description to be chosen. The selection can be given in an *inclusive* way (it is chosen if the constraints are met) or in an *exclusive* way (it cannot be chosen if the constraints are met).

Conditional selection only applies if the attributes exist; otherwise it has no effect. This allows us to use an attribute even if not all members of a family actually have this attribute.

The weakest selection method is by default, and it is used only if it does not conflict with other selection methods. This methods naturally expresses a preference among several otherwise equivalent choices.

The selection strategy starts with a set of possible implementations and successively applies the different selection mechanisms until this set is restricted to exactly one choice. If the set becomes empty or at the end it contains more than one possibility, the user is notified that the given description is inconsistent or ambiguous.

The activity of version control is performed on atomic objects: versions, histories, binary, manuals, documentations, and so on. Adele uses an RCS-like scheme with a double-level numbering of implementations, a space-saving delta mechanism and attributes like date and author.

6 Discussion

Finally, we can draw our conclusions about the generations that we have distinguished on the evolution of tools supporting CM and VC. We will summarise the problems that each generation has attempted to solve and give an analysis of the problems they have left often or have introduced.

The first generation tackled the problem of having two separate tools for solving two related tasks. Although the two activities are integrated, they are not interconnected with the rest of the SDE. This means that the user is free to invoke his favourite editor but that he has to manually interact with the tool/system for CM and VC. Another weak point is the generality of the tools with respect to the contents of components regarded as unstructured text files. The drawback of placing no restriction on their contents is that the system cannot take advantage in situations where their contents are indeed restricted. The very limited attribute schema is another weakness. Attributes like time and numbering are not enough to capture all the information contained in a module. This simple form for attributes derives from having just the file system as the 'glue' that connects different tools.

The second generation systems moved away from the file system point of view, introducing the notion of a database. However, the step is very small, as it is limited to considering components as black boxes rather than files. The advantages of having an embedded attribute schema in the database are not exploited, and the selection mechanisms are still rather primitive. Another weakness derives from the assumption that the programming languages used have no constructs for intermodular type-checking, and that they have no means for expressing module interconnection. The consequence of this assumption is that these systems require from the user an external description of each component in terms of its import/export features and of the other components it uses. This approach is convenient when different languages are used for different components, but it becomes an overhead in Cedar and in the GP because they normally support languages that actually contain such constructs. This overhead consists of a lot of redundant information that has to be manually provided by the users.

The third generation systems assumes that software is written in languages that have modularisation constructs. These systems can be customised with respect to a specific language. The result is a language-specific system that has knowledge of the semantics of the components. This knowledge is exploited, for instance, to automatically construct dependency graphs. Finally, they take advantage of the underlying database which provides a rich set of attributes that can be used for choosing the components needed to instantiate a system. Another advantage of having a database as central repository is that the integration among different tools can be made very strong.

7 Conclusions

In this paper we have analysed the problem of configuration management and version control and their integration, using six systems as examples to stress the different aspects of the problems and their possible solutions. Since it is beyond the purpose of this paper to give an evaluation of the systems described, the level of detail is kept low and the reader is referred to previously published literature.

Nevertheless, we have some ideas about how these activities and tools can be improved. First of all, configuration management and version control must be performed inside an environment in order to be integrated with all the other development activities. A fully fledged database should have a key position in this integration as the central repository of knowledge, so that CM and VC can be performed with respect to the conceptual schema underlying the environment. A database is necessary because

- ☐ data are persistent;
- ☐ data are shared;
- ☐ data are structured and related;
- ☐ data should be able to be queried.

A sophisticated query system is needed to provide a more natural and friendly access to the environment at different levels of abstraction.

Finally, there is a need for more intelligent and automated tools which, exploiting the knowledge embedded in the environment, can help the user more effectively in his/her software development activities. A number of proposals on this theme are contained in Reference 8.

8 Acknowledgment

This work has been partially supported by Ministero della Pubblica Istnizione, by Olivetti D.O.R. and by the Danish Ministry of Education.

9 References

[1] FELDMAN, S.I.: 'Make — a program for maintaining computer programs', *Softw. — Pract. Exp.*, 1979, **9**, pp 255–265

[2] ROCHKIND, M.J.: 'The source control system', *IEEE Trans.*, 1975, **SE-1**, (4), pp. 364–370

[3] FOWLER, G.: 'The fourth generation Make'. Proc. Summer 1985 USENIX Conference, 1985

[4] TICHY, W.F.: 'RCS — a system for version control', *Softw. — Pract. Exp.*, 1985, **15**, 637–654

[5] SWINEHART, P.T., ZELLWEGER, R., BEACH, J., and HAGMANN, R.B.: 'A structural view of the Cedar programming environment', *ACM TOPLAS*, 1986, **8**, (4), pp. 419–490

[6] HABERMANN, A.N., and NOTKIN, D.: 'Gandalf' software development environments', *IEEE Trans.*, 1986, **SE-12**, (12), pp. 1117–1127

[7] BELKHATIR, N., and ESTUBLIER, J.: 'Experiences with a database of programs', *ACM SIGPLAN Not.*, 1987, **22**, (1), pp. 84–91

[8] AMBRIOLA, V., CIANCARINI, P., CORRADINI, A., and DEFRANCESCO, N.: 'Towards innovative software development environments'. TR6-88, Dipartimento di Informatica, Università di Pisa, Italy

The paper was first received on 7th August 1989 and in revised form on 29th May 1990.

V. Ambriola is with the Dipartimento di Matematica e Informatica, Universitá di Voline, Via Zanon 6, 33100 Voline, Italy; P. Ciancarini and L. Bendix are with the Dipartimento di Informatica, Universita di Pisa, Corso Italia 40, 56100 Pisa, Italy