Software Configuration Management and Engineering Data Management: Differences and Similarities

Bernhard Westfechtel[1] and Reidar Conradi[2]

[1] Lehrstuhl für Informatik III, RWTH Aachen
Ahornstrasse 55, D-52074 Aachen
bernhard@i3.informatik.rwth-aachen.de
[2] Norwegian University of Science and Technology (NTNU)
N-7034 Trondheim, Norway
conradi@idi.ntnu.no

Abstract. Engineering data management and software configuration management have been evolving fairly independently. On the other hand, it has been observed earlier that many parallels exist [10]. In this paper, we examine the similarities of and differences between EDM and SCM. Many concepts are similar, but there are some differences concerning the objects to be managed. As a consequence, some sophisticated features of modern SCM systems are not applicable in the EDM domain because they are based on assumptions which do not hold there (objects represented as text files, tools operating towards the file system). Some suggestions for further work on version models and on the architecture of version support systems are outlined.

1 Introduction

Engineering data management [29] — frequently also called product data management [19] — is concerned with the management of design data in engineering disciplines such as mechanical (CAE/CIM), electrical (VLSI), or chemical engineering. That is, it is concerned with managing machine-readable data about physical objects, not the objects itself. Since the term "software engineering" was coined in the late 60s [34], many people believe that software construction is — or should be — an engineering discipline as well. As a consequence, we would expect software configuration management (SCM) to be a part of engineering data management (EDM).

Indeed, software engineering support tools and EDM tools have much in common: design philosophies and methods (e.g., object-orientation), complex and versioned data structures, storage and exchange of parts of such data, an associated tool architecture, flexible user interfaces, etc. However, the EDM and SCM community have been evolving fairly independently. On the other hand, the parallels were recognized some time ago. In 1990, Randy Katz published a paper on version models for engineering data [23]. Only recently, we surveyed the state-of-the-art in SCM in the same journal [9]. Although both papers introduce different terminologies and taxonomies, the similarities are still striking.

The ongoing series of SCM workshops provides an important forum for presenting theoretical and practical work in this field [7]. This year, "S" stands for "System" rather than "Software", encouraging a more global perspective concerning the objects to be managed. What does this mean to SCM? This paper contributes a partial answer by comparing SCM and EDM, mainly focusing on version models, or how to structure the version space.

Our comparison is mainly based on an analysis of approaches presented in the scientific literature. We cover both research prototypes and commercial systems. Our expertise refers primarily to the SCM domain, but we have also gathered some experience in EDM [43].

Section 2 illustrates the parallels between EDM and SCM by a couple of examples. Section 3 compares both domains more systematically with respect to the product space (the objects to be managed) and the version space (the concepts for maintaining versions of evolving objects). Looking at EDM and SCM more closely, we identify some important differences which have had a major impact on the development of EDM and SCM systems. Section 4 summarizes our findings.

2 Similarities Between EDM and SCM

Comparing EDM and SCM, we observed that virtually every concept occurring in one domain is also present in the other. Moreover, there are several systems with striking similarities which their authors were not aware of. Below, we give some examples to support this claim.

Object and version plane. In several systems, objects and relationships are arranged into planes which are related by version refinement. For example, this holds for OVM [22], which has been developed for VLSI design, and CoMa [42], which has been applied to both software engineering and mechanical engineering. In both systems, the object plane contains versioned objects (with object identifiers) and their relationships. The version plane refines the object plane, so that each versioned object is refined into a set of versions (with version identifiers) standing in a version graph. Similarly, relationships between versioned objects are refined into more detailed relationships between their versions. We may try to split the version plane further in an "upper" variant layer, and a "lower" revision layer, but these layers are intertwined multi-layers – corresponding to branching in the version graph. The object plane may be used in the following ways: As a view, it provides an abstraction of the bulk of information in the version plane. As a set of constraints, it describes which elements may or must be created in the version plane.

Design and module hierarchies. EDM systems support hierarchies of versioned design objects. This can be done in different ways. Here, we discuss only one of these. The Version Server [6] represents versions of VLSI design objects as follows: Each version has both a contents and a (potentially) empty set of composition relationships pointing to versions of nested design objects. Thus, the

version of some composite object uniquely determines the versions of its components ("version first" selection). POEM [28] applies the same modeling approach to software engineering, relying on module hierarchies where modules are connected by (acyclic) dependencies. A module version references versions of those modules on which it depends.

Product versioning. Several SCM systems, including EPOS [27] and Voodoo [35], apply repositories to manage versions of software objects and to combine these into whole configurations. We first specify a version description, serving as a filter against the repository and defining a uni-version "product view" (i.e., version-first selection). Later, during tool-initiated queries against the selected and available product view, we are not concerned with versioning (product-last selection).

Product versioning also appeared "outside" SCM. For example, the database version approach [5] was developed for the object-oriented DBMS O_2 [2]. Here, database versions are organized into a tree, so that versioning of objects is completely transparent once a database version has been selected.

Attribute-based versioning. Attribute-based versioning is a technique to integrate versioning into the data model. When we define a version model on top of some data model (e.g., OO or ER), we must declare pairs of types: one type for a versioned object and one type for its versions, respectively. In the SCM system Adele [13], we can define one versioned object type instead. The type definition distinguishes between versioned attributes (e.g., textual contents) whose values are specific to each version, and unversioned attributes (e.g., change logs) whose values are shared by all versions and are thus properties of the versioned object as a whole. Again, this approach is not constrained to SCM. For example, attribute-based versioning was also proposed for the object-oriented DBMS EXTRA [37].

Interfaces and realizations. Several SCM systems have been designed for configuring modular programs. For example, in Gandalf [18] a module has a unique interface and potentially multiple realization variants. For each variant, there is a sequence of revisions. Adele generalizes this approach by its family concept: each family may have multiple variants or views of an interface, e.g., subsets of signatures of exported functions or header-files expressed in different programming languages.

Similar models were developed for EDM. For example, the CADLAB system [3] configures versions of VLSI design objects and makes a distinction between interfaces and realizations as well. In contrast to Gandalf and Adele, CADLAB permits different versions of one design object to co-exist in one configuration. However, such co-existence is intentionally supported by a few SCM systems as well. For example, the Cedar System Modeler [24] allows that different clients select different implementations of the same interface. This is particularly important in distributed systems, where heterogeneous processing nodes often run slightly different variants of the "same" software, e.g., Unix vs. NT, or Intel-80x86 vs. SPARC.

3 Differences Between EDM and SCM

Despite these similarities, some crucial differences exist between EDM and SCM systems. These concern both the product space, i.e., the objects to be managed and their relationships, and the version space, i.e., the way how versions of these objects and relationships are organized.

3.1 Product Space

Objects. SCM systems deal with general software objects in machine-readable form, so-called software configuration items. These objects comprise requirements specifications, software architecture designs, module interfaces, module bodies, test cases, test logs, various documentation, project plans, etc. Much of SCM functionality concerns management of human-produced source programs, represented as text files and manipulated with development tools such as editors, compilers, linkers, and debuggers. A classical SCM system therefore provides a central repository (database) for efficient storing of versioned text files. Further, it offers merge tools for combining versions based on an analysis of textual differences. It usually includes a Make tool [14] to reliably and efficiently regenerate an executable software system, based on compile and link steps.

Several software objects encompass "structured" (non-textual) information, such as diagrams and tables in e.g., requirements specifications, architectural designs or project plans (as Gannt diagrams or spread sheets). Indeed, CASE tools for software specification and design may employ very complex data structures [16]. (Compilers and debuggers do likewise, but their data structures are considered intermediate and not permanent, or can be regenerated automatically from source texts.) However, instead of storing data structures in a DBMS, most CASE tools use textual representations to store and exchange data – cf. the discussion on object-oriented databases [1]. Typical textual formats are the ones used e.g., by SDL editors. Of more standard exchange formats, we can mention CDIF [12] for software engineering and IDL [38, 11] for compiling interchange. Since these data representations are tool-produced and tool-consumed, they may not be easily readable or maintainable by human developers.

As an assessment of the software tool situation, we can say that modern CASE tools are not much in use. Text-based software development, mainly centered around programming-related activities, is still dominant. However, this situation is expected to change, especially in the light of the upcoming portfolio of UML tools. So, in the long term, the Integrated Programming Environments [33] from the 80s may become a future reality. We then need to permanently store and interchange more complex data structures, such as abstract syntax trees or graphs.

So, for the time being, we can conclude that SCM is considered a more important step towards process maturity than tool support for requirements engineering and design. This is reflected in the recommendations from ISO-9001 and CMM-level2, both emphasizing product management. So far, many SCM systems mainly focus on the management of text files. Although the designers

of SCM systems have recognized the needs for managing structured objects, they have not gone far in this respect for several reasons. First, the marketplace demands SCM support for programming with classical tools such as text editors, compilers, and debuggers. Second, management of structured objects is hard to support and requires sophisticated object management techniques. Third, there is little agreement on common data models for structured objects, and implementing special solutions for specific customers does not pay off.

In EDM, today's situation is different. In typical engineering applications, text files play only a minor role. For example, this applies to the VLSI domain, where we have e.g. functional, logic and layout descriptions of designs. In CAE/CIM, there are also a vast number of types of design objects, since the corresponding physical product has a complex and diverse breakdown, e.g. subobjects such as cranes, walls, floors, windows, pipes, valves, pumps, wires etc. Furthermore, PPC (production planning and control) systems frequently store their data in relational databases. Thus, the domain-specific tools which are integrated with an EDM system, often store their data in (potentially home-grown) databases and frequently use their private data representations.

The services provided by SCM systems may be inadequate or insufficient in EDM applications. Let us give two examples:

- In SCM, a *Check-out command* extracts a version from a central repository into a local, file-based workspace. Internally, this may involve a conversion from the internal data format (for efficiently storing multiple versions using deltas) into a plain text file. However, conceptually no conversion takes place: we simple retrieve the file which we stored into the repository earlier. This may not be sufficient in EDM, where we have heterogeneous data representations. Thus, a more sophisticated conversion may have to be carried out between the data format used in the repository and the data format requested by the tool in its workspace, and vice versa.
- Recently, *virtual file systems* have become popular in SCM. Clearcase [25], ICE [44], and n-DFS [15] are some examples of SCM systems which provide the tools with a virtual workspace. E.g., Clearcase allows Unix and NT users to transparently access the repository through their native file systems. In many EDM settings, however, such facilities are not offered. For instance, the informatics and mechanical engineering department in Aachen jointly performed the SUKITS project [43], where we had to integrate tools running under MS-DOS, VMS, and different Unix versions. Many of these tools were also based on proprietary data formats and home-grown database systems. As a consequence, a virtual file system was simply not viable, at least on present operating systems.

Since integration of heterogeneous tools is difficult to perform, several efforts have been launched to standardize the data representations. Most notably, the STEP initiative (Standard for the Exchange of Product Model Data [21]) defines a data model (EXPRESS) for product data and so-called partial models for certain subdomains (e.g., CAD or manufacturing planning), i.e., schemas described

in EXPRESS. These schemas are remarkably rich and cover several thousands of pages, defining e.g. the representation of geometric data at a very fine-grained level. There is no counterpart to STEP in software engineering since there is little agreement on software life cycle models and notations for software objects (note the upcoming UML, however). On the other hand, STEP addresses the management of engineering data throughout the whole product life cycle, i.e., it does not prescribe the way an EDM system organizes design objects. Part 41 of the STEP standard (Product Structure Configuration Management) covers only the composition of the final product (usually maintained by PPC systems), but not the organization of all the design objects which describe it.

STEP also defines a textual data format for the exchange of product data between heterogeneous tools. STEP-compatible tools are required to import and export these textual STEP files without loss of information. In the case of STEP-compatible tools, the EDM system can provide a repository of versioned text files in a quite similar way as in the SCM domain. In the long run, this solution is not satisfactory because design objects stored in STEP are still considered individually, ignoring coarse- and fine-grained relationships.

Relationships. SCM often utilizes "horizontal" dependencies between software objects, usually traceability dependencies or import/include dependencies. These dependencies are often used for change propagation, e.g., for build processes. Many of these are inferred from the source code by parsers, e.g., the MakeDepend utility in the Make tool, which have limited knowledge of the syntax of the underlying programming language.

At first glance, EDM differs from SCM in that hierarchies, or "vertical" composition relationships (Part-Of relation), are considered the most important relationships in EDM. However, a closer look reveals that certain kinds of composition relationships in EDM correspond to dependency relationships in SCM. For example, the Version Server manages a hierarchy of VLSI design objects. But the design hierarchy is different from the part hierarchy. The latter refers to the physical parts the final product is composed of. For example, a processor consists of a set of registers, an arithmetic-logic unit (ALU), etc. The part hierarchy models physical composition[3]. In contrast, the design hierarchy models relationships between design objects. The design of a processor does not physically contain the design of the ALU (which can be used for many processors). Rather, the processor design *references* the ALU design (relationship between applied occurrence and definition). Thus, in the case of design hierarchies we actually deal with dependencies (see also the discussion of design and module hierarchies in Section 2) as they appear in software engineering (e.g., include dependencies between modules in C)[4].

[3] In fact, there are different kinds of part hierarchies, e.g., folded and unfolded ones, but this is irrelevant to the point we want to make here.

[4] This is still a bit simplified because design hierarchies are always acyclic, while dependencies may sometimes contain cycles (e.g., class references in Eiffel).

3.2 Version Space

Version space representations. Virtually all SCM systems can be traced back to either SCCS [36] / RCS [39] and conditional compilation. In the former case, the version space of a software object is represented by a version graph. In the latter case, the version space is defined by attributes and their combination constraints (so-called version rules). Thus, an n-dimensional grid is appropriate to represent the version space (with each axis corresponding to one attribute). Examples of SCM systems based on SCCS/RCS are DSEE [26] and its successor Clearcase [25], while EPOS [32] and ICE [44] have their roots in conditional compilation.

Virtually all EDM systems are based on version graphs. Remarkably, alternatives to version graphs are not mentioned in Katz's paper at all. Both revisions and variants of individual components can be represented in version graphs. However, multi-dimensional variation does cause problems because of the combinatorial explosion of the required number of branches.

State-based and change-based versioning. The mainstream of SCM systems relies on state-based versioning, i.e., a version is described by the properties of the evolution state it represents. For example, this can be done by identifying some revision (by a revision-attribute holding a time stamp or revision number) and some variant (by variant-attributes holding enumerated or string values). However, a small number of SCM systems are founded on change-based versioning, i.e., a version is described by the functional changes performed relative to some baseline. Recently, state-based and change-based approaches have started to converge [8]. For example, Asgard [30] implements change-based on top of state-based versioning. The opposite solution (state-based on top of change-based) is also viable, in e.g., EPOS [31] and ICE.

However, change-based versioning until now seems to have been ignored by the EDM community. At least, we are not aware of any change-based EDM system. The only exception is the Version Server [6] which adopts the PIE [17] approach by collecting logically related changes in layers. However, even in this case it seems as if the concept of change-based versioning has not been fully recognized. We do believe that there is no fundamental reason for this. Indeed, change-based versioning can be applied to EDM as well. The concept of change is equally useful for organizing work in this domain as in SCM.

Extensional and intensional versioning. Extensional versioning means that the members of the version set (i.e., all versions) of a versioned object are defined by explicit enumeration. Each of these members has been created previously by Check-in. Thus, extensional versioning is concerned with the reconstruction of existing versions.

In the case of intensional versioning, the version set is defined implicitly by the properties of its members. Any version may be constructed which satisfies these properties — regardless of whether it has ever been constructed before. Intensional versioning implies flexibility: any requested version can be constructed on demand. On the other hand, we have to take care of consistency: the outcome

must be correct with respect to version constraints (e.g., consistent selection of the same variant) and product constraints (e.g., syntactic and semantic consistency with respect to the rules of a programming language). Thus, inconsistent combinations have to be detected and excluded.

Note, however, that a configuration, being a selected set of individual versions, seldom has been "Checked-in" as a whole, even if each component individually was explicitly Checked-in. Indeed, we can compose previously unseen and thus "new" configurations, even if their components are not new. Here, we may consider each component as a "delta" of the encompassing configuration.

All SCM systems support extensional versioning, most of them provide support for intensional versioning as well. Different architectures are employed for the combination. The classical solution is to realize intensional versioning on top of extensional versioning, as in Adele or Clearcase. To this end, a configuration is described by an expression referring to versions of components which are selected from version graphs. The inverse approach takes intensional versioning as the base on top of which we can implement extensional versioning. EPOS and ICE demonstrate that version graphs can be simulated on top of (a derivative of) conditional compilation. Please note that both EPOS and ICE offer a basic, uniform version model, i.e., a framework on top of which different version models can be implemented. Compared to conditional compilation, the strength of these systems lies in their deductive database capabilities, in particular concerning consistency constraints.

By and large, intensional versioning has attracted little attention in EDM systems. Again, there is no fundamental reason for this. As in SCM, design objects may evolve into a large number of revisions and variants, and many changes are applied over their lifetime. Versions of design objects have to be combined in a consistent way in EDM as well. However, support is limited e.g. to SQL-like database queries.

Could SCM systems such as EPOS and ICE be applied successfully to EDM? The answer is probably "no". The point is that intensional versioning is deeply built into the SCM system (extensional versioning being realized on top rather than below intensional versioning). EPOS and ICE perform intensional versioning at the fine-grained level ("fine-grained merges"). Both systems handle software objects represented as text files. In addition, EPOS supports arbitrary versioned attributes in a crude way – but the OID and TYPEID attributes are not versionable! In both systems, fragments (attributes) are tagged with visibility expressions, being boolean expression over a set of global versioning-attributes, and updated upon Check-in.

This approach does not carry well over to design objects. In this case, we have to cope with a variety of native data formats. If the contents of design objects cannot be interpreted by the EDM system, we cannot do more than just store the contents "as-is". Perhaps, we can do this efficiently by a built-in Diff on binary files, but certainly we could do better if we knew the logical structure. Even if we can store the data in textual form (e.g., as a STEP file), a textual

merge[5] is usually not sufficient. In particular, textual merging of STEP files may produce syntactically inconsistent results. This applies to textual merging of programs as well. But the difference is that programs are edited by the user. Therefore, we may expect that the user is capable of fixing an erroneous merge. We cannot make this assumption when merging STEP files, since these are not meant to be human-readable and -editable.

These difficulties may explain that conditional compilation has not played any significant role in EDM. Indeed, conditional compilation requires adequate support for fine-grained configuration of the contents of design objects. This requirement is rarely met, see next section.

Different versioning characteristics. We might expect different versioning characteristics in the SCM and EDM domain. For example, a classical argument is given by Tichy in [40]. Since software may be changed more easily than hardware, SCM has to cope with significantly more revisions and variants than CM for hardware. Even if we accept that claim to hold, we have to keep in mind that EDM does not deal with physical parts. Rather, it is concerned with the descriptions of these. Thus, we have to compare the product development process in engineering disciplines against the software development process. Prior to production, design objects may change very rapidly and frequently. Development methods such as simultaneous and concurrent engineering [4] take this into account (e.g., by encouraging pre-releases of intermediate versions of design objects so that design errors can be detected much earlier). It is by no means evident that software objects evolve more dynamically than design objects in EDM. So far, there is very little published data from either of these domains. We must only conclude that all this needs further exploration.

4 Conclusion

From our investigations, we draw the following conclusions:

- The concepts underlying SCM and EDM systems are very similar. For virtually every concept in one of these domains, we can also find an occurrence in the other.
- There are some differences which are merely terminological. For example, we have shown that the so-called design hierarchies are actually built from dependencies.
- Other differences are accidental. For example, change-based versioning could be applied in EDM as well. The idea has simply not been taken up so far.
- A fundamental difference exists with respect to the kinds of objects to be managed. Up to now, SCM mainly focuses on the management of software objects (mainly programs) represented as text files (as mentioned earlier, this will change when CASE tools are going to be used more widely). Tool

[5] Below, we use the term "merge" loosely to denote any kind of intensional versioning, including both conditional compilation and 3-way merging.

integration primarily refers to compilers, editors and debuggers. Virtual file systems are seen as the ideal workspace support.
In contrast, the most important objects in engineering application are non-textual (e.g., designs or production plans). Applications store their data in a variety of formats, and they frequently do not execute in the file system.

- Many SCM systems support intensional versioning, while only a few EDM systems address this to a limited extent. In particular, systems which provide fine-grained, built-in support for intensional versioning are expected to fail in EDM because of the large variety of data formats involved.
- The ability to allow multiple variants or representations of the "same" object in a configuration becomes increasingly more relevant in federated and/or distributed systems.

By and large, SCM and EDM are related fairly closely. As a consequence, experts from both domains should cooperate more intensively than they used to do. In particular, cooperation is required to support the development of hybrid products consisting of both hardware and software components in a uniform way.

Acknowledgements

Many thanks go to the EPOS group in Trondheim and the IPSEN group in Aachen. The comments of the unknown reviewers are also gratefully acknowledged.

References

1. M. Atkinson, F. Bançilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. DOOD'89, Kyoto, Japan*, pages 40–57, Dec. 1989.
2. F. Bançilhon, C. Delobel, and P. Kannelakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.
3. M. Brielmann, E. Kupitz, D. Mallon, et al. A Common Data Schema for Tool Integration. In *Proceedings CAD'92*, pages 127–140. Springer Verlag, 1992.
4. H.-J. Bullinger and J. Warschat, editors. *Concurrent Simultaneous Engineering Systems*. Springer-Verlag, 1996.
5. W. Cellary and G. Jomier. Consistency of Versions in Object-Oriented Databases. In Bançilhon et al. [2], pages 447–462.
6. E. E. Chang, D. Gedye, and R. H. Katz. The Design and Implementation of a Version Server for Computer-Aided Design Data. *Software — Practice and Experience*, 19(3):199–222, Mar. 1989.
7. R. Conradi, editor. *Software Configuration Management: Proceedings from SCM7 Workshop*, Boston, USA, 18–19 May 1997. Springer Verlag LNCS 1235, 234 p.
8. R. Conradi and B. Westfechtel. Towards a Uniform Version Model for Software Configuration Management. In Conradi [7], pages 1–17.
9. R. Conradi and B. Westfechtel. Version Models for Software Configuration Management, 59 p. *ACM Computing Surveys*, 1998. (Accepted August 1997).

10. S. A. Dart. Parallels in Computer-Aided Design Frameworks and Software Development Environments Efforts. *IFIP Transactions A*, 16:175–189, 1992.
11. T. Didriksen, A. Lie, and R. Conradi. IDL as a Data Description Language for a Programming Environment Database. *ACM SIGPLAN Notices*, pages 71–78, Nov. 1987. (Special issue on IDL, ed. C. Robert Morgan).
12. Electronic Industries Associates, Engineering Department, Arlington, VA. *CDIF CASE Data Interchange Format EIA/IS-106/107.*
13. J. Estublier and R. Casallas. The Adele Software Configuration Manager. In Tichy [41], pages 2–11.
14. S. I. Feldman. Make — a Program for Maintaining Computer Programs. *Software — Practice and Experience*, 9(3):255–265, Mar. 1979.
15. G. Fowler, D. Korn, and H. Rao. n-DFS: The Multiple Dimensional File System. In Tichy [41], pages 135–154.
16. A. Fuggetta. A Classification of CASE Technology. *IEEE Computer*, pages 25–38, Dec. 1993.
17. I. P. Goldstein and D. G. Bobrow. A Layered Approach to Software Design. Technical Report CSL-80-5, XEROX PARC, Palo Alto, California, 1980.
18. A. N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Trans. on Software Engineering*, SE-12(12):1117–1127, Dec. 1986. (Special issue on GANDALF).
19. S. B. Harris. Business Strategy and the Role of Engineering Product Data Management: A Literature Review and Summary of the Emerging Research Questions. *Proceedings of the Institution of Mechanical Engineers, Part B (Journal of Engineering Manufacture)*, 210(B3):207–220, 1996.
20. P. B. Henderson, editor. *Proc. 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh), Apr. 1984. In ACM SIGPLAN Notices 19(5), May 1984.
21. ISO. *ISO 10303: Product Data Representation and Exchange (the STEP Standard).*
22. W. Käfer and H. Schöning. Mapping a Version Model to a Complex-Object Data Model. In *Proceedings 8th International Conference on Data Engineering*, pages 348–357, Tempe, Arizona, 1992. IEEE Computer Society Press.
23. R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, Dec. 1990.
24. B. W. Lampson and E. E. Schmidt. Practical Use of a Polymorphic Applicative Language. In *Proc. 10th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 237–255, Austin, Texas, USA, Jan. 1983.
25. D. Leblang. The CM Challenge: Configuration Management that Works. In Tichy [41], pages 1–38.
26. D. B. Leblang and R. P. Chase, Jr. Computer-aided Software Engineering in a Distributed Workstation Environment. In Henderson [20], pages 104–112. In ACM SIGPLAN Notices 19(5), May 1984.
27. A. Lie, R. Conradi, T. Didriksen, E. Karlsson, S. O. Hallsteinsen, and P. Holager. Change Oriented Versioning. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, LNCS 387, pages 191–202, Coventry, UK, Sept. 1989. Springer Verlag.
28. Y.-J. Lin and S. P. Reiss. Configuration Management with Logical Structures. In *Proc. of the 18th International Conference on Software Engineering*, pages 298–307, Berlin, Mar. 1996. IEEE Computer Society Press.

29. K. G. McIntosh. *Engineering Data Management — A Guide to Successful Implementation*. McGraw-Hill, Maidenhead, England, 1995.
30. J. Micallef and G. Clemm. The Asgard System: Activity-Based Configuration Management. In I. Sommerville, editor, *Software Configuration Management: ICSE'96 SCM-6 Workshop*, LNCS 1167, pages 175–186, Berlin, Germany, Mar. 1996. Springer-Verlag.
31. B. Munch. HiCOV: Managing the Version Space. In I. Sommerville, editor, *Software Configuration Management: ICSE'96 SCM-6 Workshop*, LNCS 1167, pages 110–126, Berlin, Germany, Mar. 1996. Springer-Verlag.
32. B. P. Munch, R. Conradi, J.-O. Larsen, M. N. Nguyen, and P. H. Westby. Integrated Product and Process Management in EPOS. *Journal of Integrated CAE*, 1995. (Special issue on Integrated Product and Process Modeling), 30 p.
33. M. Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*. LNCS 1170, 709 p. Springer Verlag, Berlin, 1996.
34. P. Naur and B. Randell, editors. *Software Engineering – Proc. NATO Conference in* Garmisch-Partenkirchen*, 1968*. NATO Science Committee, Scientific Affairs Division, NATO, Brussels, Jan. 1969.
35. C. Reichenberger. Concepts and Techniques for Software Version Control. *Software — Concepts and Tools*, 15(3):97–104, July 1994.
36. M. J. Rochkind. The Source Code Control System. *IEEE Trans. on Software Engineering*, SE-1(4):364–370, 1975.
37. E. Sciore. Version and Configuration Management in an Object-Oriented Data Model. *VLDB Journal*, 3(1):77–106, Jan. 1994.
38. R. Snodgrass. *The IDL Description Language: Definition and Use*. Computer Science Press, Rockville, MD 20850, USA, 1989.
39. W. F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience*, 15(7):637–654, 1985.
40. W. F. Tichy. Tools for Software Configuration Management. In J. F. H. Winkler, editor, *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 1–20, Grassau, Germany, 1988. Teubner Verlag.
41. W. F. Tichy, editor. *Configuration Management*, volume 2 of *Trends in Software*. John Wiley and Sons, New York, 1994.
42. B. Westfechtel. A Graph-Based System for Managing Configurations of Engineering Design Documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):549–583, Dec. 1996.
43. B. Westfechtel. Integrated Product and Process Management for Engineering Design Applications. *Integrated Computer-Aided Engineering*, 3(1):20–35, Jan. 1996.
44. A. Zeller and G. Snelting. Unified Versioning through Feature Logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):397–440, Oct. 1997.

This article was processed using the LaTeX macro package with LLNCS style