A Fine-grained Version and Configuration Model in Analysis and Design

Dirk Ohst, Udo Kelter
Praktische Informatik, Fachbereich Elektrotechnik und Informatik,
Universität Siegen, D-57076 Siegen,
{ohst|kelter}@informatik.uni-siegen.de

Abstract

In this paper we present a model of version and configuration management in the early phases of software development and an implementation of this model. We assume that software documents are modeled in a fine-grained way, that they are stored as syntax trees in XML files or a repository system, and that tools directly operate on these syntax trees. In contrast to file-based systems, structural changes in the document, e.g. the shifting of a method between two classes, can be identified in our model. Configurations allow us to manage groups of single modifications; such a group will mostly correspond to a specific design task or a similar activity. Configurations are thus a means to establish a connection to a change management system.

Index Terms – fine-grained data model, versions, configuration, design transaction, software engineering environments

1 Introduction

Software configuration management (SCM) is an indispensable part of high-quality software development processes. SCM is well established and common practice in the late phases of software development, notably during programming and integration. It is less commonly practiced in the early phases, i.e. analysis and design, for a number of reasons.

One of the reasons is that there are usually not many versions (and hardly any configurations) of analysis and design documents. Thus, occasionally making backup copies is often sufficient. However, object-oriented development methods (e.g. the Unified Process) lead to a significant increase in the complexity and the number of versions of the documents in early phases.

A second reason is that usual SCM systems are not well adapted to the needs and circumstances of document management in the early phases. A large number of SCM systems and concepts is available [2]. However, virtually all of them (incl. systems such as RCS, CVS and SCCS) work only on files, to be specific, files containing lines of text in pretty-printed format. These systems are not aware of the logical structure of the document contained in a file. Their usefulness heavily depends on the assumption that a modification of the document, e.g. the insertion of a statement in a program, typically has the net effect that one or a few adjacent lines of the text are inserted, deleted or modified. They fail to work reasonably under three conditions which occur often in document management in the early development phases and/or with object-oriented development methods:

1. Documents are not text, but diagrams (e.g. the different types of UML diagrams). Diagrams are often stored in files, either in proprietary (printable) formats or in XML formats. In the case of a class diagram, each class might be represented by a few lines of text in the file. The order of these sections of text is irrelevant! The position where a class symbol appears in the diagram is explicitly stored in layout data. Therefore, diagram drawing tools can store the sections representing classes or other diagram elements in arbitrary order. As a consequence of this, even small changes in the diagram can lead to a complete reshuffling of the file contents and a large number of significant textual differences.

2. In object-oriented development methods, analysis, design and implementation are considered as parallel activities (rather than sequential ones as in the waterfall model). As a result, even simple modifications can affect several files, or parts of files, belonging to different development phases. Conventional SCM systems have substantial problems to correctly represent such complex changes.

3. Some tools which support the development of information systems use a database (e.g. a relational one or a specific repository system) to store entity relationship diagrams, database schemata and similar documents. The documents are stored as a set of tuples or objects

and references connecting them; basically, a syntax tree is represented by these data. This approach of modeling and storing documents is commonly called *fine-grained data modeling*. Conventional SCM systems cannot handle this situation at all.

In some cases changes of documents are very complex due to several reasons, e.g. extending the functionality or restructuring of components. This leads to the creation of a large number of versions. Some of them are only of temporary interest because a developer wants to store some consistent and intermediate versions of a document while working on a task or phase of a project. Conventional version management systems do not support temporary versions or relations between versions and tasks or phases of projects. When using these systems the developer can only create successive or parallel versions which can also be accessed by other developers even if this is not desired.

This paper presents concepts and an SCM system which are applicable under the above conditions. One basic assumption is that tools use an internal storage system which follows the approach of fine-grained data modeling. The concepts presented here are applicable independently of whether XML files, a proprietary file format, a relational or object-oriented database is used. The SCM system presented here is integrated in a repository system known as H-PCTE [4], which is a structurally object-oriented DBMS.

The rest of this paper is organized as follows. Section 2 presents our versioning concept, starting with a discussion of the limitations of file-based versioning of structured documents (section 2.1). An introduction to the used data model is given in section 2.2. The versioning model itself is presented in section 2.3. Section 3 describes the limitations of file-based versioning in structuring the version tree (section 3.1) and presents a solution (section 3.2) which is integrated into the version model. In section 4 we present versioning concepts which are related to our approach. Concluding remarks are given in section 5.

2 Version and Document Model

2.1 Limitations of File-Based Version Management

During its development a software product passes through several development cycles. During the analysis one creates a model which consists of the conceptual classes. They are changed later in the design phase and extended with further classes. Due to later extensions with new functions or due to the correction of errors, the software architecture, i.e. the class structure, is usually modified again and again. As a simple example, we consider the class `Panel` shown in figure 1(a). This class represents a container for graphical elements. Our system is later extended by a class `ScrollPane`, which offers a comparable functionality, however extended by scrollbars.

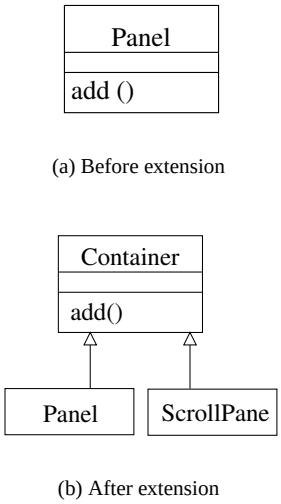


Figure 1. Example of a class structure

Both classes contain operations with the same functionality, e.g. the method `add`, which adds a new graphical element to the container. Therefore the developers extend the model with a common super class `Container` which realizes this and similar methods (s. figure 1(b)); the common methods are shifted from the class `Panel` into the class `Container`.

The shifting of methods from one class into another corresponds to the shifting of a block of text between two files or within one file. In either case, conventional SCM systems based on state comparison cannot identify this shift correctly. These systems interpret this as the deletion of one block of text at the first location and the insertion of a new block of text at the second location.

Fine-grained data modeling in combination with version management make it possible to detect the shift of a method (and similar modifications in the structure of documents) because a method is represented by a unique object and because objects have an identity.

2.2 Document model

In a fine-grained data model all elements of a UML diagram are modeled as separate objects, for example the document itself, all classes, methods, attributes and parameters of methods. An example of a meta-model of a fine-grained model for UML class diagrams is shown in figure 2. Between the object types there are component relationships, e.g. a document contains classes, a class contains methods

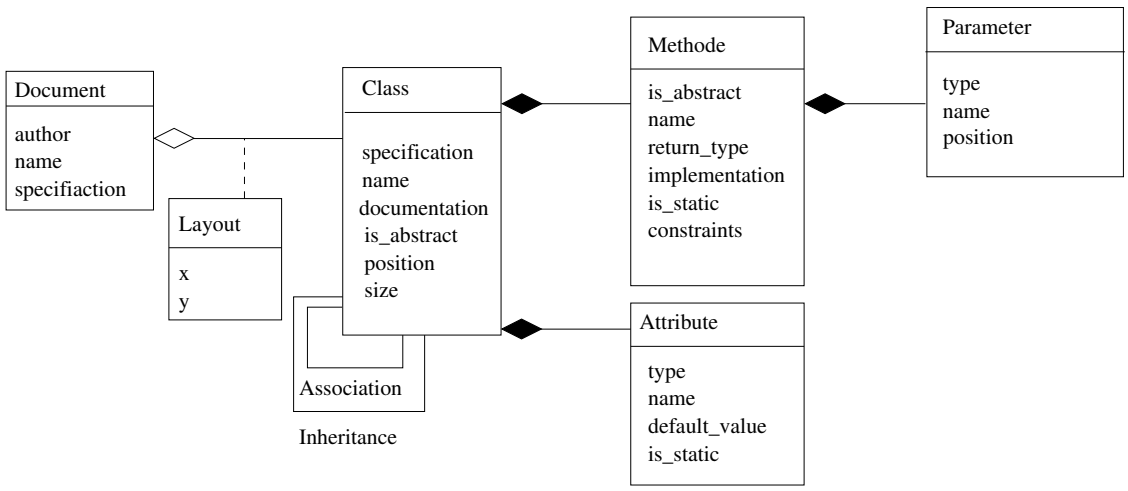


Figure 2. Meta model of a fine-grained data model

and attributes and so on. Further kinds of relationships between object types which represent classes can express inheritance, aggregation or association relationships between classes. All relationships are assumed to be bi-directional, so that the object structure can be traversed by the software engineering tools from the root to the leaves and vice versa. The object attributes contain data which describes the classes, methods, etc. Examples are the name or the type of an element, or layout information belonging to one diagram. Let us look how to store the class structure given in figure 1. The object structure shown in figure 3 uses the meta model from figure 2. Every component of the classes is represented by an object of the appropriate type.

There are two requirements on software engineering tools when using a version model for a fine-grained data model. Firstly, the tools have to modify the syntax tree of the documents, so that all modifications of the documents are represented as operations upon the syntax tree. The second requirement is that structural changes are done by shifting the objects across the document and not by deleting and recreating them. The latter would result in new objects with new identities, which makes it nearly impossible to find the new position inside the syntax tree because the whole tree must be searched.

2.3 Versions and configurations of documents

The tools use *tool transactions* (TTAs) [5] offered by the repository to operate on the data. The TTAs are long running transactions, they should not be misunderstood as transactions in conventional data bases. The TTAs have a longer run time and slightly modified characteristics [6]. All modifications which are made in context of a TTA lead to an automatic versioning of the modified objects and relationships with the restriction that each object and each relationship is versioned only once in a TTA. Each object and each relationship is versioned independently, thus it has its own version tree. The automatic creation of versions increases the probability that the version needed by a user at a later access actually exists. But this depends on the number of the executed TTAs and the number of operations executed within the TTA.

Beside simple modifications the deletion of the objects or relationships creates new versions but marked as deleted. When shifting an object from one location to another two versions of the relationship between the shifted object and the two origin objects are created. At first a new relationship between the shifted object and the new origin object is created which is marked as ”created because of a shift”. Secondly a new version of the relationship between the old origin object and the shifted one is created which is marked as ”deleted because of a shift”. Beside this information references to the old and the new origin object are stored. Without such information the shift could only be recognized by comparing the old and the new origin object. This is possible because one can walk from the shifted object to the origin object in both versions.

The automatic creation of versions leads to a large number of object versions. If these versions are recombined arbitrarily by a user, inconsistencies could result. A user cannot manually select a version of all objects or relationships because there are too many. It would mean that the user has to choose the desired version for every element of a UML diagram, namely all classes, methods, attributes or parameters. So it becomes necessary to store consistent versions together. This is achieved by configurations. A configuration combines all versions of objects and relationships created in context of a TTA. We call the configuration which is currently modified by a TTA the *working configuration.*

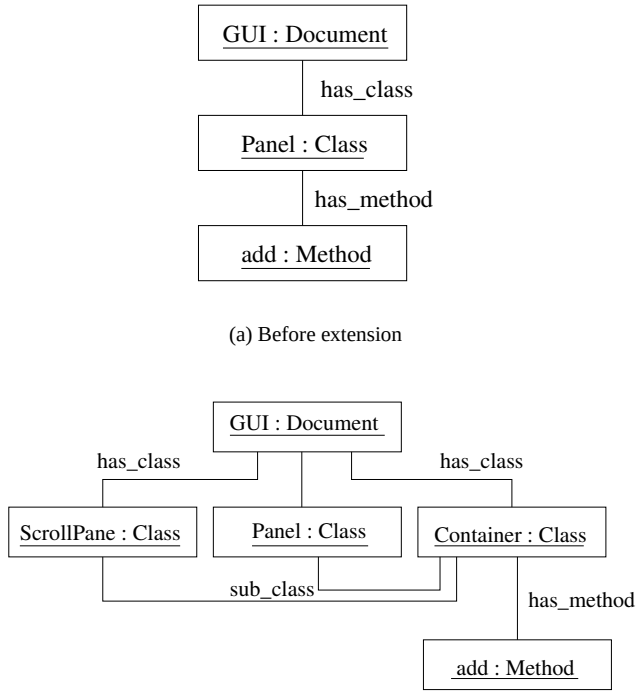


Figure 3. Example of an object structure

Sometimes it is necessary to mark a state of a document as consistent or as a release version. This is achieved by creating a configuration manually which is the new working configuration. The old one is frozen. All further created versions will be stored in this new working configuration.

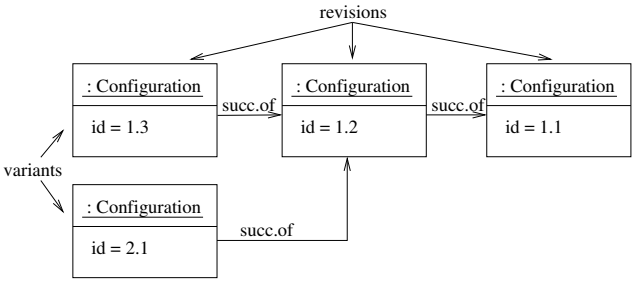


Figure 4. Structure of configurations

The basis for accessing versions desired by the user are the configurations. Therefore a *base configuration* has to be chosen for a TTA which determines the accessible versions. The working configuration of this TTA is then a direct successor of the base configuration. If one chooses a base configuration with one or more direct successors a variant will be created (s. figure 4). Only the current versions of objects and relationships are accessible within the TTA. The latest version could be either currently modified in the working configuration or it could be a frozen version from the base configuration or from one of its predecessors. If there are several versions of an object or a relationship in succeeding configurations, only the most recent version is accessible.

The configuration without any predecessors is called the *initial configuration*; it is created automatically at the beginning of the project and does not contain any data. The first started TTA uses this configuration as its base configuration.

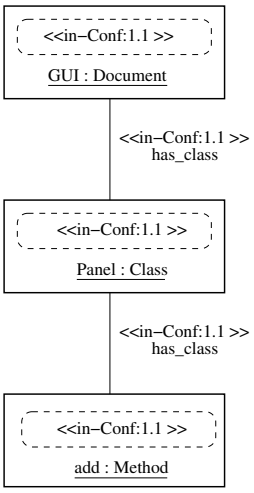
All configurations having a successor cannot be changed any more. One can therefore understand the entire configuration structure as a persistent coarse-grained undo-log.

All configurations are stored as persistent objects in the repository, in order to allow the users simple access, especially for specifying a base configuration. Furthermore, the configuration objects can be extended by information about the executed changes, for example a modification comment.

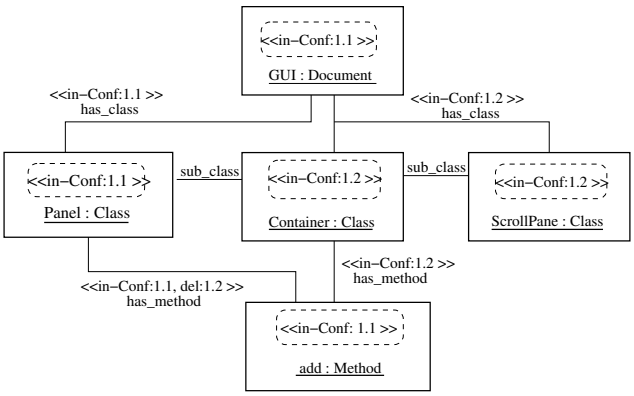
In contrast to file-based SCM systems there are no explicit workspaces because all data i.e. the versions are managed by the repository. The repository is the workspace [3]. Because all tools operate in context of a TTA and all changes are stored within a configuration, one can imagine a configuration as a virtual workspace, which is altered by a TTA. Thus the configurations form a tree of workspaces. Modifications are only possible at the leaves of the configuration graph but not at internal workspaces.

We consider again the example from section 2.1. The object structure before and after the extension is presented in figure 5 with a slightly modified UML object diagram notation. The bordered stereotypes indicate the configuration in which the versions of objects and relationships are available and thus accessible. Figure 5(a) shows the object structure created in configuration 1.1 before the extension. Each object exists only in one version. All relationships are bi-directional, so that one can walk through the object structure in any direction, and determine for example the class in which a method resides by walking from the object representing the method to the object representing the class.

Figure 5(b) shows the object structure in configuration 1.2 after creating the classes `Container` and `ScrollPane` as well as shifting the method `add` into the class `Container`. The two new classes are modeled by new objects. The objects representing the method `add` and the class `Panel` are not modified by shifting the method `add`. Only the relation between these objects is modified. Therefore a new version of this relation is created, it belongs to the new configuration 1.2. The old version of this relation belongs to configuration 1.1 and is not accessible any more in a TTA based on configuration 1.2 because the relation between the class `Container` and the method `add` is a direct predecessor of the relation between the class `Panel`



(a) Before change



(b) After change

Figure 5. Object structure in repository

and the method `add`.

3 Logical changes of documents

3.1 Limitations of File-Based Version Management

Conventional file-based version management systems only deal with versions of documents. All versions of a document are stored in one big version tree. Single branches of such a tree represent alternative realizations or they result from concurrent work. Versions of different documents which belong together can be labeled with a symbolic name. It can be used for labeling consistent versions, release versions or versions created within a task or phase of a project. One disadvantage is that no additional information can be stored in the symbolic name, e.g. the task specification. A possible solution could be to store additional information as comments on every created version. But this is a bad solution because of several reasons. Firstly, the comment is stored at every created version and therefore several times. Secondly, the versions which belong together are not easily identifiable. And thirdly, it is impossible to specify the task or phase of a project in advance in which the modification has to be done. This has to be done afterwards taking the risk not to include all versions into the result of a project phase. The problem increases with the duration of the changes to complete a task or phase of a project because a lot of versions of one document are likely to be created. One symbolic name cannot label all created versions of this document. Thus different names have to be used, but this is too complex and error-prone. Another possible solution is to create a branch and store all versions created in context of a task in this new branch. But the branches are only numbered and the relation between branch number and task had to be managed externally.

3.2 Design Transactions

One can understand all modifications done to complete a task or phase of a project as the result of a long transaction. We call this type of long transactions *design transactions* (DTAs). The main difference between DTAs and tool transactions (TTAs) is that a DTA is a logical frame within which the TTAs are executed. The DTA does not change the documents directly; all changes are done inside the TTAs which are executed inside an operating system process. Terminating an operating system process results in terminating a TTA. In contrast to that a DTA is not bound to an operating system process and has a longer run time. Because a DTA is a logical frame for the TTAs all configurations are created inside it.

Tasks must often be divided into sub-tasks. Therefore DTAs can have a hierarchical structure. The root DTA can be mapped to the entire project, which is divided into sub-projects or sub-tasks. Every sub-project or sub-task is mapped to a corresponding sub DTA of the root DTA. We store DTAs as persistent objects in the repository. An example of a DTA structure is presented in figure 6. Each DTA references one or more configuration branches. Only the latest configuration of a branch is directly referenced. Every DTA has a base configuration which determines the versions of objects and relationships that can be modified. This configuration must be declared at the time the new DTA is created. The base configuration references an existing configuration of another DTA. If the new DTA is the root DTA an empty configuration will be created.

Data about a task such as its specification, involved de-

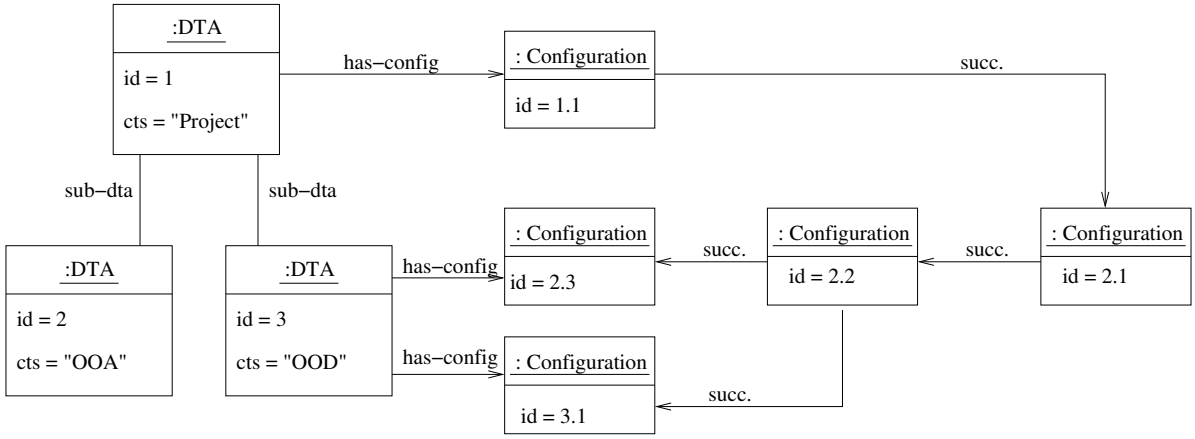


Figure 6. Structure of design transactions with configurations

signers etc. must be managed. Every software engineering process may have additional data. Therefore users can adapt the DTAs to their needs by declaring subtypes which add further attributes to the DTA objects.

In summary the DTAs can be used to partition the structure of configurations and to relate successive configurations to tasks or phases of projects. Furthermore the DTA concept is not rigidly bound to a software development process. In fact the tool developer has the freedom to use the proposed versioning concept in a wide variety of software development processes. One option is the use of a process machine which controls the creation and use of DTAs. Another option is to let the developers create DTAs manually. Besides this it is possible to use one DTA for the entire development or for every small task of a project. This flexibility allows the proposed versioning concept to be adapted to different process models, e.g. the waterfall model, the spiral model, or the Unified Process.

4 Related work

Almost all version management systems are file-based, only a small number of such systems deal with the versioning of structured documents. One example is IPSEN [12]: all versioned documents are modeled as abstract syntax graphs. A modification within a document results in a new version of the whole document; this leads to a very large number of object versions. One improvement on this is the Unified Model [1] which is based on *change propagation*. The method creates new versions of all objects located at the path from the modified object up to the root-object of the document, whenever an object is changed. This leads to a large number of unused object versions [7] and does not work with fine-grained data model [8]. Besides the large number of unnecessarily created versions both systems have some common limitations. The first concerns the identification of structural modifications in documents. The problem is the absence of a version tree for each object or node in the syntax tree. Without such a version tree it is not easily possible to identify structural changes because the entire document has to be searched for the shifted objects. The second problem is that all created versions resides in one big version tree; therefore one cannot relate tasks to versions of objects created due to finishing this task.

Ensemble [9] [10] uses another approach of fine-grained versioning of structured documents. Every object has its own version tree called the local version tree. The consistency between versions of different objects is ensured by a global version tree which references the object versions for each document. If there are references between two documents, the global version trees of these documents contain references to one another. The main focus of Ensemble is the improvement of incremental algorithms. This model does not deal with the determination of differences, merging of versions, or relating modifications to tasks.

The systems presented so far deal with arbitrarily structured documents. The version model DIVERS presented by Rho and Wu [8] is oriented towards the versioning of software diagrams. Fine-grained modeling is assumed, thus a node of a diagram is represented by an object. Instead of change propagation the authors suggest using a log of the performed editing operations. However they do not offer a method to create configurations of object versions belonging together and there is no method of relating changes and phases of a project. Furthermore the access to old diagram versions is not efficient because old versions have to be calculated using the log of editing operations. If the version tree of a class or a method shall be shown, the entire log must be searched.

Most SCM systems mentioned so far are state-based, i.e. the objects or files are versioned as a whole. In contrast to this change-based versioning is based on the idea of com-

bining a group of modifications in different files into one logical change. This model has some advantages [11] over state-based versioning. It corresponds better to the way how developers think. Another advantage is that it groups modifications in several components. This avoids errors when resetting modifications because all concerned components are known. However, the authors are not aware of a change-based version management system which is oriented towards fine-grained documents.

5 Conclusion and future work

The model presented here has the advantage that the modifications made for the solution of a given task or in context of a project phase can be related to configurations. The structure of configurations can be partitioned by design transactions (DTA). These can be used to connect the SCM system to a change management system. Because one can create the DTAs hierarchically, it is flexible and not bound to a specific software development process. The documents are fully under version control for the entire development cycle of a product.

Because new configurations are automatically created, even in case of minor modifications of objects or relationships, the entire history of documents' modifications is documented. One example is the shifting of a method between two classes.

Further work addresses the copying of objects. A copy of an object is a new part of a document with its own version tree which is unrelated to the version tree of the original object. Therefore a relationship must be created between these two version trees to express the copy-of relation.

The fundamental functionality to create versions is implemented and working. It has been integrated into the tool environment PISET [5]. Because a kind of tool transactions is already used inside PISET, the versioning functionality could be integrated without considerable efforts. The tools had to be extended by functions which create and select configurations. The next step is to realize a mechanism to compute differences between versions and to merge different versions to a new one.

Acknowledgments

We would like to thank our colleagues with whom we discussed the ideas that led to the present paper and, in particular, M. Monecke and M. Welle.

References

[1] U. Asklund, L. Bendix, H. B. Christensen, and B. Magnusson. The unified extensional versioning model. In J. Estublier, editor, *Proceedings of the 9th International Symposium on Software Configuration Management (SCM-9), Toulouse, France, September 5-7, 1999*, volume 1675 of *Lecture Notes in Computer Science (LNCS)*, pages 100–122, Berlin - Heidelberg - New York, 1999. Springer–Verlag.

[2] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[3] J. Estublier and R. Casallas. The Adele configuration manager. In W. Tichy, editor, *Configuration Management*, pages 99–133. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994.

[4] U. Kelter. H-PCTE – a high-performance object management system for system development environments. In *Proceedings COMPSAC Illinois, September 23-25*, pages 45–50. IEEE Press, 1992.

[5] U. Kelter, M. Monecke, and D. Platz. Constructing distributed SDEs using an active repository. In *Proc. 1st Intl. Symposium on Constructing Software Engineering Tools (COSET '99); 17.-18.05.1999, Los Angeles, CA*, pages 149–158, 1999.

[6] D. Platz. *Ein Werkzeugtransaktionskonzept für Objekt-Managementsysteme als Basis von Software-Entwicklungsumgebungen*. Shaker Verlag, Juni 1999. Dissertationsschrift, Praktische Informatik, Universität-Gesamthochschule Siegen.

[7] R. Ramakrishnan and D. J. Ram. Modeling design versions. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 556–566. Morgan Kaufmann, 1996.

[8] J. Rho and C. Wu. An efficient version model of software diagrams. In *Proc. 5th Asia-Pacific Software Engineering Conf., 2-4 December 1998 in Taipei, Taiwan, ROC*. IEEE Computer Society, Dec. 1998.

[9] T. A. Wagner and S. L. Graham. Integrating incremental analysis with version management. In *Proceedings of ESEC '95 - 5th European Software Engineering Conference. Sitges, Spain. 25-28 Sept. 1995*, pages 205–18, Berlin - Heidelberg - New York, 1995. Springer–Verlag.

[10] T. A. Wagner and S. L. Graham. Efficient self-versioning documents. In *Proceedings IEEE COMPCON 97. Digest of Papers. San Jose, CA, USA. IEEE Comput. Soc. 23-26 Feb. 1997.*, pages 62–67. EEE Comput. Soc. Press, Los Alamitos, CA, USA, 1997.

[11] D. W. Weber. Change sets versus change packages: Comparing implementations of change-based SCM. In R. Conradi, editor, *Proceedings of the 7th Workshop on System Configuration Management (SCM-7), at ICSE'97 Boston, MA, USA, May 18-19, 1997*, volume 1235 of *Lecture Notes in Computer Science (LNCS)*, pages 25–35, Berlin - Heidelberg - New York, 1997. Springer–Verlag.

[12] B. Westfechtel. *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*, volume 280 of *Informatik-Fachberichte*. Springer–Verlag, Berlin - Heidelberg - New York, 1991.