SPACE-EFFICIENT TECHNIQUES FOR STORING VERSIONS OF SOFTWARE COMPONENTS

Mária Bieliková, Pavol Návrat
Slovak Technical University,
Dept. of Computer Science and Engineering,
Ilkovičova 3, 812 19 Bratislava, Slovakia,

The paper describes space-efficient techniques for storing versions of software components which are represented as files. Techniques are based on the use of differential files (deltas). We proposed, analysed, implemented and compared several techniques for applying deltas to trees of versions. We report on an environment the purpose of which is twofold: it supports effective storing and recovering versions and it allows to experiment with a branching tree of versions.

1. INTRODUCTION

Software systems consist of many components. In the course of system development and maintenance, the components undergo changes. The changes reflect development steps, or improvements and fault correctings, or the changes in the system's environment. An important subclass of transformations that change the components are those which, intuitively speaking, preserve the specification of the component and do not radically change the language [1].

Components resulting from such transformations are called versions. Similarly to other works in this area [4, 10, 14] we distinguish two kinds of versions: variants ('parallel' versions) and revisions ('serial' versions). Variants of software components are alternative implementations of a particular concept. Revisions, on the other hand, are modifications intended to replace the previous version. A family of software components comprises all components which are versions of one another.

Versions of software component are often organized in an ancestral (history) tree [11, 12, 3, 8]. The tree has a root version which represents the first version of a software component. Arcs represent the relation *developed_from*. A young history tree is slender: it consists of only one branch, called trunk. As development proceeds, side branches may rise. A need of branching arises in the following situations: (i) simultaneous development among multiple users; (ii) distributed development at several sites; (iii) exploratory development; (iv) old versions still in use need to be fixed; (v) versions with alternative purposes are created. Let us note that only the last case of branch forming is the case of a variant as it is described above.

One essential task of software configuration management is to store the history tree of versions efficiently. To address this problem, several techniques have been devised. The key space-saving idea is that if one version has been developed from another (i.e., a successor in the history tree) then the two versions probably have a great deal of commonality and a small number of differences. To save space only one version is stored in full and the other in form of so called *delta scripts* [9].

When studying the problem of delta storing, two aspects are important: (1) how to generate a delta between two files (and also to recreate the version from delta and version stored intact) and (2) how to apply delta to the version history tree, i.e. which version to store in full and to which version to calculate a delta.

Our goal has been to summarize known approaches of delta application, to propose new techniques and to experiment with their properties. Our concern for software version storing is part of a larger project aimed at developing a method for building a software configuration [2].

The paper is organized as follows. In Section 2 brief overview of the delta generation technique is given. Our proposed techniques for delta application together with known approaches are presented in Section 3. Next, we turn our attention to experiments with described techniques. The paper closes with our conclusions.

2. CONSTRUCTION OF DIFFERENTIAL FILES

Differential file or delta is formed from two original files (versions) by computing the difference between them. Let us assume that $F1$ and $F2$ are files and D is the delta to be generated. D must contain all the information to reconstruct $F2$ from $F1$ (see Figure 1). If $F1$ is older version than $F2$ (in respect to time of creation) D is called a *forward delta*. In the opposite case, when $F1$ is newer version than $F2$, D is called a *reverse delta*.

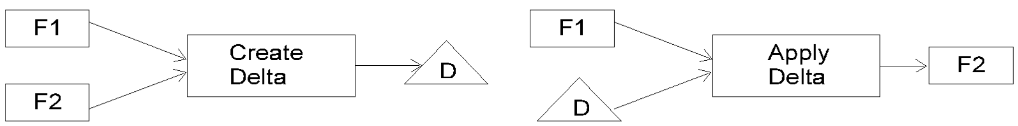

Figure 1: Creation and recovering delta.

Delta D consists of a sequence of edit commands which describe actions to be performed in order to reconstruct the file $F2$. Two edit commands are sufficient: "copy n units from $F1$ to $F2$ starting at s" and "add the unit u to $F2$".

The grain of difference can be a line [11, 12, 7], a byte [9], a database record [6]. The advantage of line oriented deltas is the speed of delta creating. However, the size of delta is bigger and a line oriented delta cannot be used with non-text files. We extended these approaches by a delta based on lexical units. It can be used for text files e.g., program source code. In this case deltas are smaller than the line oriented ones and calculation is easier than with byte oriented approaches.

To generate a delta between two files, an algorithm for isolation of the so called longest common subsequence of two versions is used. Tichy's algorithm [13] generates delta script in linear time and space. It is guaranteed to produce a minimal delta by so-called block moves.

3. DELTA TECHNIQUES

Using deltas is a classical space-time trade-off: deltas reduce the space consumed, but increase access time. In dependence of the application purpose one will stress either space economy or access speed.

Several delta techniques are known. Often one technique is used by one specific tool, e.g. forward delta by SCCS [11], CCC/Manager (Platinum technology), NUCM [5], reverse delta with only the latest revision on trunk stored in full by RCS [12], reverse delta with latest revision on each branch stored intact [8], PVCS Version Manager (Intersolv), Source Integrity (Mortice Kern Systems), Visual SourceSafe (Microsoft), Gypsy [3].

The simplest delta technique is known as the *forward delta*, i.e. differences between two files are created in such a way that delta is applied to transform the old version into the new one. The forward deltas are calculated between the first and second versions, the second and third versions, etc. Only the first version is stored intact. To restore nth version the first, second,$\ldots$,$(n-1)$th deltas are applied to the first version.

We proposed slight modification of the above technique - *forward delta related to the first version*. In this case the forward deltas are calculated between the first and second versions, first and third versions, etc. This technique is much faster than previous. Creating and restoring version requires only one delta calculation. On the other hand, this approach usually requires more space. The first version is in most cases the smallest one, so differences between first and other versions tend to grow very quickly. The technique can be improved by storing version in full when differences surpass a specified limit. Forward delta related to the first version is advantageous for applications with many parallel branches where fast access to any version is required.

Another proposed modification of forward delta technique takes into consideration the different nature of revisions and variants. Usually, differences between variants are much bigger than between revisions. If a side branch represents a variant, it is stored intact. Versions on main trunk (on each branch) are stored as forward deltas related to the first version in the specified branch. Figure 2 illustrates a history tree with one side branch representing a variant (version 6) and one branch created due to different purpose (e.g. parallel development of versions 8 and 5). This approach seems to be suitable for applications where many variants exist that differ significantly from each other.

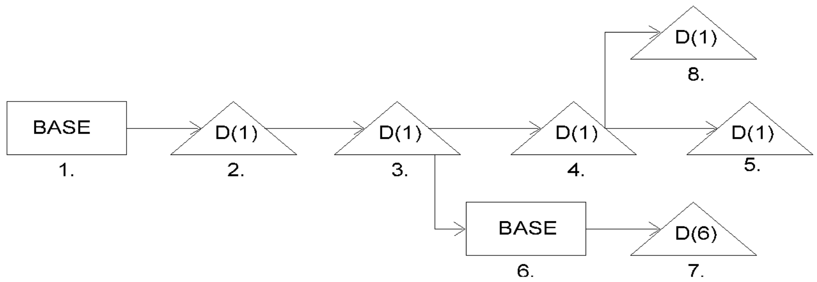

Figure 2: Forward delta related to variant.

The most recent version is accessed much more frequently than earlier versions. This is the main reason for using the *reverse delta technique*. It was first proposed by Tichy. In his tool RCS [12] deltas are arranged in such a way that the most recent revision on the trunk is stored intact. All other revisions on the trunk are stored as reverse deltas. Extraction of the latest version together with adding a new version to the trunk is fast. Branches however need special treatment. RCS uses forward deltas for them. This approach has an disadvantage that restoring versions in side branches is slow. It requires application of reverse deltas from the latest version in trunk up to fork version and then application of forward deltas until the desired branch version is reached. Problem is that it is not clear which branch of the history tree will become the main line of development (trunk) as the time passes.

Modification of this technique relies on an idea to store intact latest versions on all branches. Reverse deltas are smaller when compared to forward deltas because usually the latest version is the biggest. This technique is fast for all leaves of the history tree but is much more space consuming than Tichy's reverse delta technique. In both techniques recovering of versions that are not leaves is difficult i.e., time consuming.

The last proposed technique is also based on the reverse deltas. It improves access to versions that are not leaves. Delta is created always from the latest version on a specified branch. The recovering is fast, as it requires at most one delta application. Adding the new version to the tree takes time proportional to the number of versions in the branch. This technique is suitable for software components archivation where the operation of adding a version is less frequent.

4. EXPERIMENTS AND CONCLUSIONS

The proposed delta techniques have been experimentally verified. We implemented an environment in C++ language with a double purpose: it supports effective storing and recovering of versions and it allows to experiment with a branching tree of versions.

We performed several experiments aiming at experimental evaluation of properties of the described techniques. In particular, we concentrated our attention to the following criteria:

- mean time required to add any version to the history tree (T_IN)
- mean time required to recover any version (T_OUT)
- mean time required to recover last version on any branch (TL_OUT)
- amount of space saved (S).

In order to achieve unbiased independent results, we tested delta techniques with randomly generated problems (i.e., history trees of versions). In the first series of experiments, we worked with regular trees. In the second series of experiments, we worked with randomly generated trees with the probability of forming a branch set to 0.1. According to our opinion this mimicks the actual development process. Nodes of history trees were created on the base of random modification from 5% to 10%.

We summarize results of our experiments in one graph (see Figure 3) which serves for comparing the techniques according to both access time and space compression.

Figure 3: Comparison of delta techniques

We have presented six delta techniques which enable space-efficient storing of versions. The techniques have been implemented and experimentally verified. Their properties (access time and space consumed) are different and create a good basis for the use by various applications. It is desirable to have a possiblity to use different delta techniques within a configuration management tool and to make conversion between them as the software system evolves. Next step in our research is to verify our experiments by daily use of the environment in the software development process.

REFERENCES

[1] M. Bieliková and P. Návrat. Modelling software systems in configuration management. *Applied Mathematics and Computer Science*, 5(4):751–764, 1995.

[2] M. Bieliková and P. Návrat. A knowledge based method for bulding a software system configuration. *Knowledge Based Systems*, 9(1):61–65, 1996.

[3] E.S. Cohen *et al.* Version management in Gypsy. In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, pp.201–215, Boston, 1988. ACM Press.

[4] J. Estublier. The Adele configuration manager. Technical report, L.G.I., Grenoble, 1992.

[5] D. Heimbigner A. van der Hoek and A.L. Wolf. A generic, peer-to-peer repository for distributed configuration management. In *Proc. of the 18th Inter. Conf. on Software Engineering*, pp.308–317, Berlin, Germany, March 1996. IEEE.

[6] R.H Katz and T.J. Lehman. Database support for versions and alternatives of large design files. *IEEE Transactions on Software Engineering*, SE-10(2):191–200, March 1984.

[7] D.B. Leblang and Chase R.P. Computer-aided software engineering in distributed workstation environment. In *SIGPLAN/SIFSOFT Symposium on Practical Software Development Environments*. ACM, 1984.

[8] B. Magnusson, U. Asklund and S. Minör. Fine-grained revision control for collaborative software development. *Software Engineering Notes*, 18(5):33–41, December 1993.

[9] Ch. Reichenberger. Delta storage for arbitrary non-text files. In P.H. Feiler, editor, *3rd Int. Workshop on Software Conf. Management*, pages 144–152. ACM Sigsoft, 1991.

[10] Ch. Reichenberger. Concepts and techniques for software version control. *Software - Concepts and Tools*, 15(3):97–104, 1994.

[11] M.J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[12] W.F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[13] W.F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.

[14] A. Zeller and G. Snelting. Handling versions sets through feature logic. In W. Schäfer and P. Botella, eds, *Proc. 5th European Softwre Eng. Conf.*, pp.191–204. Springer-Verlag, 1995.