Interval Construction within Partially Ordered Object Versions [1]

Georg Heidenreich, Mark Minas

CS IMMD 2, University of Erlangen
Martensstrasse 3
D 91058 Erlangen, Germany
{heidenre,minas}@immd2.
informatik.uni-erlangen.de

Jürgen Landauer

Fraunhofer Institute
for Industrial Engineering
Nobelstrasse 12
D 70569 Stuttgart, Germany
Juergen.Landauer@iao.fhg.de

1 Introduction

Complex technical designs typically consist of many elements which may be managed as versioned objects. When building a composite system version, an appropriate set of object versions (*configuration*) has to be selected. Complicating matters, semantical dependencies between objects restrict the possible combinations of object versions.

The versions of design objects are called *revisions*. In our approach, revisions are immutable for quality assurance reasons, e.g., reconstructability. Changing an object can only be modeled by adding more revisions to that object.

2 Revision Labels

Revisions of objects (or files, respectively) have always been managed by using labels attached to the revisions. Yet, typical version managers assume that the revisions are ordered totally or at least in a tree-like order. The ordering of labels reflects the semantical dependencies of revisions of a single object, therefore the labels should be able to represent a revision *R* inheriting properties from *multiple* predecessor revisions. Multiple preceding revisions typically reflect concurrent development of updates. Total orders and trees are not useful for version labels, because they do not allow for multiple preceding revisions, at least they do not describe multiple predecessors. Thus, it is our belief that there should only be a single restriction to the ordering of versions:

> Revisions should not depend on each other in a cyclic way.

Therefore, the ordering should be a directed acyclic graph (*dag*) for each object. For simplicity, we also assume that each version dag is well-founded, i.e. that there is a single starting version for each object (this is not a necessary assumption to our theory of version classification in this paper).

As an abstract way to characterize system versions, we use absolute version expressions which allow for computing the ordering relation between labels. Version expressions serve for two purposes: Firstly, they label revisions within single objects. These expressions are called *label expressions*. And secondly, as *configuration specifications*, they are used to specify and select configurations: when building a system version, we have to select a set of revisions, one for each object. System versions are automatically built using a configuration specification describing an interval condition which has to be satisfied by every revision selected.

When building configurations, there is not always a revision exactly matching the configuration specification. This is no restriction, instead it is useful when building modular systems, as demonstrated in

[1] This work has been supported in part by BASYS GmbH, Erlangen and the German Federal Department of Research and Education (BMBF) under contract FKP 0004401B4A.

the following. Consider a software system consisting of a revision of a database module with label “DB = DELPHI” and a revision of a graphics module with a label “GUI = MOTIF”. Since database and graphical user interface are independent aspects, it does not make sense to develop a joint version labeled with “DB = DELPHI $\wedge$ GUI = MOTIF”. Instead, Plaice et al. suggest to use the more general revision in terms of the ordering relation if no revision exactly matching the specification is available [PW93]. Though we consider their selection principle to be very powerful, we doubt that their ordering relation is easy to compute. In our paper, we introduce an abstract labeling algebra without these drawbacks and present a practical implementation, the ALEX version manager.

In our work we introduce a version classification scheme for technically designed composite systems. As opposed to typical version management systems, our version classification pattern can also be used as a configuration specification in order to select a complete set of compatible versions, one for each object.

3 Results

We introduced a lattice algebra as a version management scheme, which is more general than tree-like version labels of typical revision management tools. Our implicit version order, where an arbitrary set of version expressions represents a version graph, also allows for the insertion of new versions, which is not easily dealt with when storing the ordering relation explicitly. Using this algebra we demonstrated how system variants can be selected automatically and independently of the number of the program files. The selection uses an interval predicate in order to specify a unique version for each object. Our approach is not only applicable to programming–in–the–large, it equally supports versioned systems within any technical design process.

4 Related Work

Plaice et al. have introduced a very general model for managing versions of versioned elements. They presented a non-lattice version algebra because their versioning operator is non-commutative [PW93]. We found that attribute-valuepairs instead of the single identifiers in their paper allow for the definition of a commutative versioning operator within a version lattice. In this lattice, version expressions can be reduced to a simple normal form, which makes computing equality and the version ordering relation $\sqsubseteq$ very easy. The paper of Plaice et al. only defines a configuration specification using the upper bound of all version labels, which, in our opinion, is not suitable for practical applications.

The idea of using attributes for version classification has been described in various articles by Estublier et al. [Est94]. However, neither the theory of that approach, nor its implementation (ADELE) can handle general semantical dependencies between different development phases. Finally, there is no way to describe joining revisions in ADELE because a tree structure is used for versions. We do not consider the tree structure to be appropriate because of the changes that have to be merged into systems. Instead, a partial order seems to be sufficient to describe a general version model.

References

[Est94] J. Estublier. The ADELE configuration manager. In Tichy, editor, *Configuration management*. John Wiley & Sons, New York, 1994.

[PW93] J. Plaice and W. W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3), pp.268 – 276, 1993.