A VERSION MODEL FOR AN ONLINE SOFTWARE REPOSITORY

Chi Pio Tou and Robert P. Biuk-Aghai

Faculty of Science and Technology, University of Macau, Macau

ABSTRACT

Recent years have seen an increased interest in distributed collaboration in the area of software engineering, brought about to a major extent by the widespread use of the Internet, and consequently the development of a suitable support infrastructure has in itself become a focus of attention. To enable a distributed development mode, the sharing of software artefacts such as documentation and program code needs to be supported. Online software repositories accessible through the Internet can provide such support. The paper presents a version model for an online software repository that has been developed as part of the ICCS project, a project to provide support for computer-aided software engineering and configuration management over the Internet. The version model supports total versioning of all artefacts in the repository, including both leaf nodes as well as collection nodes. Version histories are maintained as directed deltas relative to the current version. Moreover, both individual file versioning as well as release versioning are supported, where a release history is represented as a delta over the collection objects involved. Objects in the repository are characterized by a set of attributes, which support the querying of objects. An implementation of the ICCS Internet-based repository is also presented.

KEYWORDS

Configuration management, version model, software repository, Internet, cooperative work.

INTRODUCTION

Over the past few years, the rapid development of the Internet has enabled new modes of work, study, and interaction in general. Many buzzwords such as virtual classroom, cyberspace, distributed collaboration, virtual organizations and others are widely in use. All of these are
indicators of an increasing interest in the way through which computer communication technology can impact and improve human activities.

Within the computing field itself, much research attention is being directed towards how communications technology can improve the development and maintenance of computer and software systems; e.g. Biuk-Aghai (1996), Dominigue & Mulholland (1997), Heimbigner & Wolf (1997), Perpich et al. (1997). The development of software by distributed groups of developers, in *virtual teams*, is appealing because of its advantages over conventional, same-time, same-place development: elimination of overheads associated with relocating team members to a central development site, staffing of projects regardless of location of required staff, sharing of experts on concurrent projects, accelerated development through shift work in different time zones (Gorton & Hawryszkiewycz, 1996), and others. Such mode of work has shown to be successful even in large-scale distributed collaboration in software development, as evidenced for instance in the Linux project and various GNU projects.

However, as pointed out elsewhere (Biuk-Aghai, 1998a), such mode of work requires a suitable support infrastructure, allowing effective and efficient exchange of both messages and objects between members of a distributed team. This requires in turn that the combined aspects of Computer Supported Cooperative Work and Software Engineering be addressed. Much research has already been focused on aspects of the availability of shared objects, such as Biuk-Aghai (1996), Eslick *et al.* (1995), van der Hoek *et al.* (1996), and Keller and Schauer (1998). These address problems of *object management*, generally by providing global repositories and providing some mechanisms to control access to objects within the repository. Recently, some research has focused on the composition of the development environment from tool components, thereby increasing the flexibility and tailorability of the support infrastructure, see e.g. Biuk-Aghai (1998b). Other issues that are related in this context are distributed computing: Brown & Kindel (1998), Orfali and Harkey (1998); Shah (1999); component based software engineering: Farrar (1998), Nierstrasz *et al.* (1992), Nierstrasz and Meijler (1995), Oreizy *et al.* (1998), Pree (1997); groupware: Farschchian and Divitini (1997); and others.

The University of Macau launched the ICCS project in 1996 to investigate requirements and to implement prototypes of support infrastructures for remote collaboration in software engineering. One of the focal areas of the project was to define and implement an online software repository for the sharing of software artefacts among members of a distributed software development team. Other focal areas include the definition and implementation of composable software tool architectures and components. The following section will briefly introduce the main components of the project and outline a set of requirements for the software repository. Next, the version model underlying the software repository will be presented, followed by an outline of the current status of the repository’s implementation. Lastly a set of conclusions will be presented.

**ICCS PROJECT OVERVIEW**

As mentioned above, the ICCS project is concerned with support infrastructures for remote collaboration in software engineering. Within this context, the project has been focusing on two main areas:
1. Investigating and specifying the requirements for software repositories supporting distributed software teams.

2. Defining a component model to allow composition of tool components in a software development environment.

In addition, for each of these areas prototypes have been and are still being developed.

**Repository Requirements**

Software repositories have existed for some time, however the majority of them are placed on a central machine and used for co-located development teams. Distributed teams, on the other hand, have different requirements which the repository has to cater for; specifically:

- **Network accessibility.** As team members may be distributed around the globe, it is necessary for the repository to be accessible from any location through an appropriate network connection. We decided to specifically cater for access through the Internet.

- **Security.** Access through the Internet is inherently insecure. To provide adequate security to protect the repository content, only authorized access from registered users should be permitted. In addition, data transferred between users and the repository should be encrypted for increased security.

- **Performance.** Remote access to the repository through a (possibly) slow network such as the Internet requires a repository design that minimizes the number of messages and the message size in order to attain acceptable performance. This can be achieved through the use of data compression, as well as the provision of a repertoire of complex server-side operations.

- **Scalability.** Because of the nature of distributed collaboration, which can be global in nature and can involve very large numbers of participants, the repository should have the ability to grow and still perform well.

Besides these specific requirements, the repository also needs to provide general features that are common among conventional, i.e. non-networked, repositories:

- **Version management.** Software artefacts evolve over time, and all versions during an artefact’s evolution need to be maintained.

- **Release management.** Certain states of a collection of software artefacts that collectively make up a software release need to be captured and maintained.

- **Meta-data.** Information about the artefacts in the repository needs to be maintained, in order to facilitate searching for artefacts by their meta-data.

- **Policy-neutral.** The repository should not favour any particular configuration management policy, but should leave decisions on policy issues to higher level components.
VERSION MODEL

Based on the specific and general requirements outlined in the preceding section, we have defined a version model for an online software repository. The terminology used here is based on that found in Conradi & Westfechtel (1996).

Object and version types

From the developer’s point of view, two general kinds of objects can be distinguished and need to be stored in a software repository: on the one hand files, containing the actual software artefacts, and on the other hand folders (or directories), grouping files and folders of artefacts together. From the repository’s point of view, however, there is only one kind of thing: the repository object, hereafter referred to as object for short. Files and folders merely represent different types of objects, namely atomic and composite objects, respectively. This implies that all repository operations are equally applicable to objects that represent files and folders.

Objects in the repository evolve over time, and this evolution is represented by a set of versions. Two types of evolution are distinguished: sequential evolution, through new revisions of an object, and parallel evolution, through variants of an object. Revisions are evolutions of an object on a branch originating at the object, while variants are evolutions on a separate branch from the original one. Typically, variants are created whenever a version of a software for a different operating environment is required (such as a different CPU, OS, windowing system, etc.). As file and folder objects are treated equally in this model, this represents a total versioning scheme, since not only the evolution of files but also the evolution of folders over time is traced. In addition, objects in the repository can be made either changeable (i.e. changing the object does not create a new revision) or immutable (every change results in a new revision). While immutability is commonly the default or only possible behaviour of a configuration management repository, our version model does not predetermine this but leaves this policy question for higher level components to decide on.

To conserve space, not all revisions need to be kept in their entirety. Only the current, i.e. last revision of an object is kept in this way, while all preceding revisions are kept as a delta to their immediate successor revision. When an earlier revision is required, the repository reconstructs it from the current version and all deltas preceding the current revision up to and including the delta of the desired revision.

Contents of the repository form a directed graph, starting from a given repository root object. Non-leaf nodes of the graph correspond to folder objects, while leaf nodes correspond to file objects. Edges of the graph represent containment relationships between folder objects and other objects. This is illustrated in Figure 1 (a), where composite object x is the root of the graph, which contains composite objects y and z, which in turn contain atomic objects a and b, respectively. In addition, for each object in the repository, there is a separate directed graph representing the version history of that object. Nodes in this graph correspond to revisions, branches to variants, and edges to predecessor-successor relationships. Figure 1 (b) shows the version history for a single atomic object a, which has gone through three revisions from its original revision 1 (shown as a 1 suffix after its name), to revision 4. Object a has also a variant branch, originating from its second revision. Revisions of objects in the variant branch
are numbered starting from 1. Objects in the two branches are distinguished by being marked as belonging to two separate variants, var1 and var2 in this case.

![Diagram](image)

**Figure 1:** (a) Object containment relationship; (b) Version history

To illustrate the relationship of objects and versions in the repository, consider a practical example as shown in Figure 2. As before, rectangles represent atomic objects, ovals represent composite objects, and edges represent containment relationships. Objects are labeled with their file and folder names, respectively. It can be seen that composite object src contains three objects, r.java, o.ico, and z in turn. The first two objects are atomic while the last one is a composite one. Below the composite object z, there is an atomic object p.zip.

![Diagram](image)

**Figure 2:** Object graph

With the total versioning feature, each object is given a revision number. Figure 3 shows the revision history overlaid on the object graph of Figure 2, where the object name is now suffixed with a revision number, and where later revisions are shown in a darker colour. Now there are two revisions of objects src, r.java and z, and three revisions of object o.ico. Objects p.zip and n.zip have only one (the original) revision. The revision history of the composite objects here indicates that composite object src contained three objects r.java, o.ico, and z in its first revision, while the second revision contains only two objects, o.ico, and z. Thus, a deletion of atomic object r.java has taken place on composite object src. On the other hand, composite object z has had one atomic object, n.zip, added.
**Meta-data**

For internal identification, each repository object carries a unique identifier, its *object identifier*. Users, however, typically refer to objects by using their *remote object name* together with their revision number. These identifiers are part of the object’s set of meta-data. Other attributes contained in the meta-data include object type (atomic or composite), variant attributes (more than one is possible when multi-dimensional variants exist), owner ID, creation and last modification date, access permissions, description and comment attributes, etc. These attributes may be used for indexing and searching purposes, and can assist status tracking and auditing of the repository.

In addition to the basic version management discussed so far, which operates on single objects, the version model also provides for *release management*. In this context, a release is a collection of repository objects, both atomic and composite, which collectively represent a certain state of the software system which they are a part of. For example, when a certain version of the software is made commercially available, the state of the released system at that moment in time needs to be captured in order to enable later fault tracking on the same system state. Releases in the version model are created by cloning the hierarchy of composite objects involved in the release, and then creating composition relationships from the composite objects to just those revisions of the atomic objects that make up the release. All atomic objects referenced in such way must be in an immutable state, otherwise a release cannot be created. All objects which are part of the release are marked as such by a special release attribute.

**Linking and replication**

The version model, as explained so far, applies only to single repositories. However, to make the repository scalable, it is necessary to consider fragmentation of repositories over multiple locations. In such a case, multiple repositories may exist side-by-side and it is necessary to integrate their resources. For this purpose, the version model has two additional features, *linking* and *replication*.

Linking allows the composition of objects across repositories. Suppose that in Figure 2 above the composite object *z* and its contained atomic object *p.zip* are located in one repository and the remaining objects in another repository. Then the edge between composite objects *src* and *z* would span the boundaries of the two repositories containing these objects. This is
referred to as a *link* between repositories. Linking is made possible through the use of the internal object identifier, which identifies both the object itself, and the repository containing it. Thus any containment relationship can refer to objects in the same or another repository in the same way, providing a degree of location transparency. Linking is illustrated in Figure 4, showing the repository boundaries, and the link spanning these boundaries.

The second feature to support fragmentation of repositories is replication. With replication, a local copy of a remote object is placed in a repository. The replica’s meta-data contains a reference to the remote object identifier. In this way, whenever an access to a remote object is requested, and the local repository contains a replica of it, the access can be redirected to the local copy. The advantages are faster access and increased availability. Replication is illustrated in Figure 5, where now composite object *z* and its contained atomic object *p.zip* are both replicated in the local repository. The replication relationship is indicated by the dotted line between the original objects and their replicas. Thus, while *src* is directly linked to the remote object *z*, it is also transitively linked to the local replica of *z* (shown by the dotted line between *src* and the local replica of *z*), allowing local retrieval.
REPOSITORY IMPLEMENTATION

The previous section has described the repository’s conceptual model. Based on this model, a prototype repository was implemented. To support the diversity of platforms to be expected in a distributed development team, it was decided to use Java as the implementation language. Furthermore, since the repository can be implemented on a variety of underlying database or file systems, a three-tier implementation was adopted: repository clients operating at the top tier; the underlying database at the bottom tier; and a repository server receiving client requests and executing these on the specific database used at the middle tier. This approach allows the addition or removal of different kinds of databases without impacting clients, as the repository server always provides a constant interface to clients. Client-server communication takes place using Java RMI (remote method invocation). The choice of possible backend databases is only restricted by requiring that it can be accessed by the repository server in one of three ways: through a socket interface, through JDBC, or through CORBA.

For the prototype implementation, the HyperWave system, an advanced Web server with an object-oriented database at its core, was chosen as the underlying database. HyperWave’s features closely match the repository requirements presented earlier, making it therefore an ideal candidate. It provides its own access control, native support for atomic and composite objects, object meta-data, indexing and searching, and other desirable features. It offers a client-server communication protocol through which the repository server communicates.

To date, most of the features of the conceptual model have been implemented. Features still outstanding are encryption and compression of data transmitted through the network, several complex server-side operations, replication, and a few minor features. These features are expected to be added in the future.

CONCLUSION

Software engineering in the future is likely to be a more distributed activity than it is today, with team members dispersed over multiple locations. A suitable infrastructure is essential in order to adequately support such mode of work. One of the key elements of such an infrastructure is a software repository, to manage artefacts created by distributed team members. This paper has presented a version model which is specifically designed for the requirements of a networked software repository. Besides features of conventional version management, the model provides added features to improve security, performance, and scalability in a networked environment. A prototype repository has been implemented in Java, which currently provides the majority of the features of the conceptual model. This repository allows effective sharing of software artefacts regardless of location. In addition, it can be integrated into a component-based software engineering environment as a pluggable software tool component. In this way, a great degree of flexibility regarding the choice and combination of software tools can be achieved.
REFERENCES


