**Versioning of granulated data in hierarchically composed workspaces**

*Abstract:*

*For last 30 years a lot of researches of versioned software products, but nevertheless there still remain challenges. This article focuses on model of versioned objects and hierarchically composed workspaces. The presented model of versioned object aims to solve the issue of granulation of versioned data. The model of hierarchically composed workspaces provides methods and rules for versioning, completing the first model.*

*Keywords: version control, data granulation, workspaces, models*

*ACM Computing Classification System (1998): F.4.1.*

1. Introduction

Classic version control systems allow the users to apply version control over files [2, 5, 6, 7, 8]. The domain of software development abstractions (classes, interfaces, objects, etc.) are distinguished from file abstraction domain. These systems are very fast and widely used. Nevertheless the following disadvantages could be stated:

* The access to the objects (files) is vouched by file system. We have to notice that very few file systems support the required level of security when user works with versioned objects.
* Files as versioned object have large data granulation level. They don’t allow the user to specify the relations between separate objects (files).

In order to solve these issues researcher like Nguyen [7] makes introduce object-oriented approach of versioning. In his models data is presented as table of slots made from attributes and nodes. Nevertheless this approach allows the user to specify data in single level of granularity.

In software development industry the term workspace comprehends an isolated space (environment) where certain work is done. First commercial solution with hierarchical structure of workspaces [11] is introduced in 1989. Nevertheless Estublier [4 – page 406] emphasize that “**…a modern workspace is created “behind-the-scenes” to perform a particular user-selected task …** ”. The authors label the workspace as a system element that has to provide the following main features:

1. Sandbox – a save space where users have the opportunity to work without being affected from other users.
2. It allows building specific version/configuration of the software system.
3. Separation of changes, tests and other pursue activities without direct affect over the product nor other users’ work.

Thus we can formulate the following problems thin current article:

* To present an object-oriented model of versioned object that allows specifying freely the level of data granularity.
* To provide a model of workspaces with hierarchical composition including set of rules for version control.

2. Object Model

Настоящият параграф има за цел да представи модел на версионизиран обект. Моделът предоставя възможност за гъвкаво комбиниране между версионизирани обекти, като се изграждат композиции от обекти. Построяването и представянето на модел на версионизиран обект е реализирано като класически модел същност-отношение (Entity-Relationship Model). Той ни предоставя добър механизъм за гъвкаво и свободно реализиране на поставените задачи.

Lead author in version control domain [3, 9] determine the version object as two part entity - object states, and object versions' graph. Object version graph is a graph where the nodes represent object states, and arcs represent version transitions.

Main feature that a versioned object supposes to implement is to provide possibility to define data granulation in a free way. This feature supposes to be supported by the model of versioned object. This leads us to the need of defining of object compositions as part of the model. The formal definition of the composed object could be defined as follow:

1. Composed object is an object that is built from other object using composition entity. Съставен обект се нарича обект, който е съставен от други обекти (версии на обекти) посредством композиция.
2. As composition we will comprise the entity that defines the relation between super-object and sub-object. One composed object could be super-object for one or more composition instances, i.e. to be built by one or more sub-objects.

The fact of adding of the composed objects and sub-objects in the domain leads us to the need of redefinition of versioning process over versioned objects. On the following diagram is presented an example for change of version composition. Here arrows represent changes in objects’ content among different versions of end product.



Figure 1 An example for change in objects’ content

On versions binding of certain versioned object it is necessary to use relation of type foreign key. From this pursue that **versioned object entity** has to consist only of unique and immutable number. It is useful to regard that number as primary key for the entity.

Object’s versions could be разглеждат as its primitives (**versioned primitives**) with the following attributes:

* Versioned object id
* Version number – a serial number which specify in unique way the version within the versioned object.
* Object’s name. Setting the name to be on primitive level allows to the user to track among different versions even then the object is renamed. This model becomes more complete, excluding the weakness related with object (file) renaming in systems as Subversion (SVN), Git, and other. [2, 5, 6].
* Object content.

Versioned primitive is determined in unique way using the couple **versioned object id** and **version number**. In spite of the possibility to use that unique pair and compound key, good practice in ER model design [1] recommends that each entity to possess its own not compound key. In our case we will introduce an additional field as a primary key – global version number.

Versioning of composed object requires defining an additional entity – Versioned primitive composition (in short Composition). This entity is a relation entity that binds in a unique way version of a super-object with version of a sub-object. The following attributes of composition could be defined:

* Global number of super-object
* Global number of sub-object.

Despite of the fact that attribute combination is always unique, we will use additional attribute for primary key – composition id.

Due to accounting and change traceability needs, our model will be extended in order to support the feature of versioned graph. In ER models graph structures could be presented using two entities – entity for nodes and entity for arcs [10]. Looking to versioned object definition we could conclude that graph nodes correspond to versioned primitive entity. The missing part will be implemented as new entity which will correspond to graph arcs – arc of version graph. The new entity requires the following attributes:

* Arc id – primary key.
* Global number of source version.
* Global number of target version.
* User that made the change.
* Date and time of change.
* Additional change data.

On Figure 2 is presented the ER model of versioned object.



Figure 2 ER model of versioned object

### Versioning of composed object

In current paragraph we will represent the peculiarities in versioning of composed objects with rank one. Based on that, we will specify the versioning process of composed objects with rank N. The following definition specifies the term rank of composed object.

1. Composed object with rank zero, i.e. simple object is an object which doesn’t have associated with itself sub-objects. Composed object with rank N is an object for which the largest rank of associated sub-object is equal to N-1.

Granularity degree of an object is its rank.

It is important to note that definition of composed object doesn't apply any restrictions on sub-objects. This leads to the following conclusion:

1. One sub-object itself can be part of composed object therefore we can build a composition of composed objects.

One of core tasks ahead of current article is to avoid unnecessary complication of the models. Having that as well as missing necessity, we can specify the following restriction rules during building of composed objects:

1. In given composition of composed objects, certain object can be at most once.
2. One object can be part of at most one object composition.



Figure 3 A tree of objects

In change of composition between two objects, we should regard the objects’ versions as different (Figure 4). Let examine a chair (super-object) with armrests (sub-objects). When we remove the armrests from a chair we get a new version of the chair - a chair without armrests. We have to underline that sub-objects don't change its version. Therefore we get only change in compositions of super-object. We have similar situation in building of super-object, i.e. when we have a simple object that is transformed to composed object. When we add armrests to a chair, we get new version of that chair, without changing the version of armrests.



Figure 4 Change in objects' composition through change of version

Another feature of composed objects is the case of sub-object change we get an indirect change of the composed object (Figure 5). Let look at the example: Let we have change of chair's upholstery from blue to red. In this case not only the version of the upholstery is changed but also the version of the chair. As a special cases of sub-object change could be regarded association of an object as a sub-object and removing of association with sub-object and its transformation to simple object.



Figure 5 Indirect change of version of composed object in case of its sub-object change

The opposite case - when we have a change in super-object - this doesn't mean that version of its sub-objects is changed. So if you have a chair with three legs and red upholstery, the addition of the fourth leg of the chair doesn't change version of red upholstery's sub-object. (Figure 6)



Figure 6 Change in super-object doesn't affect the version of sub-object

The last two rules can be derived consequence:

1. Changing the version of a sub-object for a super-object doesn't affect the versions of its sibling sub-objects ().

The composed object model and visibility principles (from previous section) lead us to the need to address the problem of composed object visibility.



Figure 7 Change in certain sub-object doesn't affect version of its siblings

1. Certain composed object version is visible in certain workspace only and solely when all its sub-objects are visible in that workspace.

3. Model of Hierarchical composed workspaces

First model that we are going to examine is the model of hierarchical composed workspaces. Within the model the following definitions will be used:

1. Product is called the subject of material or immaterial manufacturing, which since its creation can be reproduced and distributed to customers.
2. Product release is called a fixed version that has passed certain quantities controls and meets the criteria of quality, safety and security. Only product releases are distributed to customers. Versions that are not release, called the practice working versions.
3. Workspace is a location where certain activities are carried out on the development of a version of a product.
4. Main workspace called workspace, which made ​​the final preparation of equipment and product release.

Arrangement (composition) of the workspace is taken to provide opportunity for each participant in product development process and its releases to carry out its activities both individually and cooperating with other participants. Namely workspace provides the opportunity for independent work that does not affect or be affected by the work of other participants. On the other hand, composing workspaces in a hierarchy claim to be a mechanism to ensure cooperative work. Diagram (Figure 9) presents a diagram of hierarchical composition of workspaces.



Figure 8 Class diagram of Product-Release-Workspace model

### Versioned object visibility model in environment with hierarchical composition of workspaces

As in any hierarchical structure and here will be considered a parent-child relation. We will focus on the versioned object visibility defined by the following, principles of visibility.



Figure 9 Example of hierarchical composition of workspaces

1. *Local version of the versioned object* is the version that is associated with a specific workspace.
2. *Visible version* *of versioned object* for given workspace means a version of that object with which the user can work.

Visibility principles:

1. *The local version of the versioned object for given* workspace is the visible version of that object in the same workspace, despite other local versions in the parent workspaces.
2. The local version of object in a workspace can be seen recursively in all subspaces, unless another local version defined therein.

From above principles we can deduce consequences:

1. In any workspace, in which objects don’t have a local version, they are presented with their version found in the nearest parent workspace
2. If for given workspace the object has no version in either parent workspace, it is not visible in the selected workspace.

On Figure 10 we present how both principles of visibility influences on object version distributions for example hierarchical composition of workspaces.



Figure 10 Distribution of object versions according to the visibility principles

In order to achieve completeness and correctness of the model can be formulated as following constraint: an object can present only one version in a workspace.

## Transactions over versioned objects

We introduced transactions over versioned objects in [Pitesti]. Here we will make a short presentation and classification of them.

### Transaction within a single workspace

Creation is the first transaction for each versioned object. After completion of the transaction, the object has an initial (zero) version, in which it is "empty", i.e. contains no information.

The transaction of state marker creation which we create a new version of a versioned object can be regarded as the basis of a mechanism for creating safe-points.

As a reverse transaction to create a state may be classified for this waiver of marker status. Through it the model was released last state and current local version of the object is the version prior refusal.

The creation of long sequences of not branched versions, especially from the same user within the same workspace, leads us to the idea of the transaction of combining successive versions, in order to save memory and follow faster and more easier analysis of the work.

Updating versioned nonlocal object, i.e. an object that doesn’t have any local version in the current workspace, can be defined as the most important of the current section. This transaction is not entirely limited only to a single workspace, since it is composed of the following steps:

* Retrieve the previous visible for workspace version of the site.
* Create a local version of the object in the current workspace.
* Създаване на релация на версиите (дъга в графа на версиите), в която предишната видима версия се явява версия-първоизточник за новата локална версия на обекта.

Deleting an object is possible by creating a transaction called “tag for deleted object”. This tag is intended to "hide" the object in its workspace and make it invisible in its sub-workspaces. It should be noted that all described in this section transactions over the object can no longer available, except for transaction rejection of tag status.

### Transaction among two workspaces

Transactions between two workspaces can be divided into two groups - the publication of object’s version and giving up a local version. Before examining, it is necessary to introduce the terms "derivative" and "parallel" version of an object.

1. Let's look at one versioned object and two versions of its X and Y. If there is a road the versioned graph of the object from version X to version Y, then Y version is called a derivative version of X, and version X is called previous version of Y.
2. Let's look at one versioned object and two versions of its X and Y. If there is no road the versioned graph of the object from version X to version Y, then both versions are called *parallel* versions or *not-derived* versions.

Publishing an object’s version means the transformation the local version of the object from the current workspace into a local version of its parent workspace.

The simple version publishing is in a situation where there is no local version of a published object in the parent workspace - Figure 11.



Figure 11 Simple publishing

Following the transaction, which needs to be addressed, is that of updating publication (Figure 12). Typical for it is that it is possible when simultaneously satisfy two conditions:

* In the parent workspace exists a local version of the object be published.
* The version of the object be published, is a derivative of the version in the parent workspace.

When an updating publishing happens merge of two versions is not needed, as a derivative version of evolution up to the previous version.



Figure 12 Updating publication

When the object version to be published in the parent workspace is a parallel to the version in parent workspace (Figure 13), then it should to merge both versions. As a result of the merger produce a new version of the object. We don’t aim to present a new method for merging versions of an object, so we can be used as handheld merge approach or algorithmic approach similar to the algorithm Westfechtel [12].



Figure 13 Publication with merge

Transaction give up of local version is the reverse transaction by publishing a version. The only step of in transaction is the remove the local version of the object on the workspace. It is important to note that if any of the parent workspaces there is no version of the selected object, then the object it becomes unavailable for subsequent use. This situation should be taken into account when the transaction takes place in the main workspace of the product's release.

### Transactions over composed objects

… transaction from ….

Let have the following situation: a local version of the object **B** in the parent workspace and its visible version in current workspace. Let make in current workspace a sub-objects in **A** of object **B**. (Figure 14). After publication of version of **A** sub-object may not lead to a change in the version of object **B** in the parent workspace. However, in a subsequent publication version of the object **B** in conjunction with his compositions in the parent workspace will lead to automatic update of the compositional scheme of objects (on Figure 14 with green dotted arrow). This is dictated by the fact that information regarding the organization of the composite object should be considered as an inseparable part of it.



Figure 14 Newly created sub-object to a super-object

Publication of the new version of the composite object B, v3 leads to the requirement that this be done in a set with the version of the newly created sub-object (Figure 14 – arrows with number 2).

Let us consider the situation where we have a local version of the object in the parent workspace that is visible in the current workspace (Figure 15). In current workspace we change the sub-object A, which leads to a change of object B, i. e. the creation of a new local version of the sub-objects leads to the automatic creation of a new local version of the entire composite object. We should notice that separate publishing the new version of sub-objects in the parent workspace should not be allowed. This restriction follows from the fact that a new version of sub-object assumes a new version of super-object (Figure 15 – red arrow with number 1), in addition we introduced the restriction that an object may be present only one version a workspace. As a conclusion to the situation we can define the following rule:

1. Publication of a version of local composite object should be made bundled with all local versions of its sub-objects that have a different version in the parent workspace  
   (Figure 15 – the green and yellow arrows with number 2).

Local versions of the sub-objects in the parent workspace can be either derivatives as well as parallel. In these cases it is necessary to execute the transactions that are covered above.

Let us examine same situation when we have a local version of the object **B** in the parent workspace that is visible in the current workspace (Figure 16). From the composition of the composed object is excluded a sub-object. Reflecting this change in the parent workspace is achievable only by publishing the composed object. This publication only transfers the composition change in parent workspace without changing the version of sub-object. The former sub-object is no longer part of the new super-object version.



Figure 15 Indirect change inversion of super-object, caused by new version of sub-object



Figure 16 The absence of change in version of object A, i.e. there is no need of its publication

Let us have visible composed object **A** with sub-object **B**, object **A** and sub-object **B** are local versions in the parent space. We remove from the composition of object **A** its sub-object **B**, i.e. we create a new local version of object **A** (Figure 17). In this case a publication of any new version of object **B** would lead to the following conflict: version **v1** of **B** requires that in its workspace the object **A** to be with its version **v1** (visible or local version).

This fact could be regarded as prerequisite for the following rule:

1. Let have an object’s version, that have previous version, and that is a sub-object of a composed object in the parent workspace. Publication of that object has to be performed simultaneous with publication of the composed object’s version.



Figure 17 Publication of former sub-object is not possible before publication of super-object’s new version

As we noted above, the reverse transaction of publication is the give-up of local version. Upon give-up of local version of the composite object we should be taken into consideration the fact that its version could largely dependent on the version of its sub-objects. This leads us to the following rule:

1. The give-up of local version of composed object has to be performed in conjunction with recursive give-up of all its sub-objects.

4. Workspace environment configurations

The presented above model of workspace composition allows us to specify its appearance in the form of workspace environment configuration. Under workspace environment configuration we will understand the process of determining the hierarchical architecture of workspaces. On (Figure 18 and Figure 19) we present two examples of workspace configurations. These diagrams display the freedom of workspaces arrangement in a way that is the most appropriate manner according to company architecture, project specifics, methodology characteristics, or other needs.

On Figure 18 is presented workspace composition where all mainstreams are divided in separate sub-trees – requirements, architecture, development and QA. Only requirements that meet requestor’s business needs suppose to be published to master workspace of the project. And only after this they become visible to other project participants. The same scheme should be used for distribution of other artifacts – architecture, source code, test cases, etc. As a disadvantage of the examined architecture we can emphasize the large amount of information that users have to study. This is very topical issue in large projects.



Figure 18 Model of organizational driven workspace configuration

In order to solve that issue on we present component-oriented architecture of workspaces. Here we have dedicated workspaces for each project participant and software components.



Figure 19 Model of component-driven workspace configuration

5. Conclusion and future work

...

Reference

1. Ambler, S. W., Pr. J. Sadalage,Refactoring Databases: Evolutionary Database Design, Addison Wesley Professional, 2006.
2. Collins-Sussman B., Fitzpatrick, B. W., Pilato C. M., Version Control with Subversion, book compiled from Revision 10945, 2008, http://svnbook.red-bean.com/en/1.0/index.html (visited in march 2009).
3. Conradi, R. and Westfechtel, B. 1998. Version models for software configuration management. ACM Comput. Surv., Vol. 30, no. 2, pp. 232–282, June 1998, DOI= http://doi.acm.org/10.1145/280277.280280
4. Estublier, J., Leblang, D., Hoek, A., Conradi, R., Clemm, G., Tichy, W., and Wiborg-Weber, D. 2005. Impact of software engineering research on the practice of software configuration management. ACM Trans. Softw. Eng. Methodol. 14, 4 (Oct. 2005), 383-430. DOI= http://doi.acm.org/10.1145/1101815.1101817
5. Git - Fast Version Control System, http://git-scm.com/ (visited in January 2012).
6. Jones, M. T., Version control for Linux, 2006, http://www.ibm.com/ developerworks/linux/library/l-vercon/, (visited in February 2009).
7. Nguyen, T. N. 2006. Model-based version and configuration management for a web engineering lifecycle. In Proceedings of the 15th international Conference on World Wide Web (Edinburgh, Scotland, May 23 - 26, 2006). WWW '06. ACM Press, New York, NY, 437-446. DOI= http://doi.acm.org/10.1145/1135777.1135842
8. Price, Derek R., CVS—concurrent versions system v1.11.22, http://ximbiot.com/cvs/manual/cvs-1.11.22/cvs.html, 2006 (посетен през април 2009).
9. Slein, J. A., Vitali, F., Whitehead, E. J., and Durand, D. G. 1997. Requirements for distributed authoring and versioning on the World Wide Web. StandardView, Vol. 5, no. 1, pp. 17-24, March 1997, DOI= http://doi.acm.org/10.1145/253452.253474.
10. Stephens, S. M. , Johan Rung , Xavier Lopez, X.: Graph data representation in oracle database 10g: Case studies in Life science, IEEE Data Eng. Bull, vol. 27, pages 61-67, 2004.
11. Sun Microsystem, Inc. The network software environment (NSE), Sun Tech. Rep. Sun Msicrosystems, Inc., Mountain View, CA, 104, 1989
12. Westfechtel, B., Structure-oriented merging of revisions of software documents. In Proceedings of the 3rd international Workshop on Software Configuration Management (Trondheim, Norway, June 12 - 14, 1991), pp. 68-79, P. H. Feiler, Ed. ACM, New York, NY, 1991, DOI= http://doi.acm.org/10.1145/111062.111071