Serdica J. Computing **7** (2013), No 2, 179–198

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

VERSIONING OF GRANULATED DATA IN HIERARCHICALLY COMPOSED WORKSPACES*

Vladimir Jotov

Abstract. For the last 30 years there has been a lot of research of versioned software products, but challenges remain nevertheless. This article focuses on a 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, which complete the first model.

1. Introduction. Classic version control systems allow users to apply version control over files [2, 5, 6, 8, 9]. The domain of software development abstractions (classes, interfaces, objects, etc.) is distinguished from the file abstraction domain. These systems are very fast and widely used. Nevertheless, the following disadvantages can be stated:

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

Key words: version control, data granulation, workspaces, models.

*This article presents the principal results of the Ph.D. thesis Models in software version management, based on hierarchical compositions of workspaces by Vladimir Jotov, successfully defended at Veliko Tarnovo University "St. St. Cyril and Methodius", Faculty of Mathematics and Informatics, Department of Computer Systems and Technologies, on 16 March, 2013.

- Access to the objects (files) is vouched by a file system. We must note that very few file systems support the required level of security when a user works with versioned objects.
- Files as versioned objects have a large data granulation level. They don't allow the user to specify the relations between separate objects (files).

In order to solve these issues, Nguyen [8] introduces an object-oriented approach to versioning. In his models, data is presented as a table of slots made from attributes and nodes. As a disadvantage of the model, it could be pointed out that many data slots could remain empty. Nevertheless the model follows the need of multilevel data granulation.

In software development industry the term workspace is understood as an isolated space (environment) where a certain work is done. The first commercial solution with hierarchical structure of workspaces [12] was introduced in 1989. Nevertheless, Estublier [4 – page 406] emphasizes 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 by other users.
2. Option of building a specific version/configuration of the software system.
3. Separation of changes, tests and other pursued activities without a direct effect over the product or other users' work.

Thus we can formulate the following problems in the current article:

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

2. Object model. Leading authors in the version control domain [3, 10] define the version object as two-part entity–object states and an object version graph. An object version graph is a graph where the nodes represent object states and arcs represent version transitions.

We can identify the main feature of a versioned object as the possibility to define data granulation in a free way. This feature needs to be supported by the model of the versioned object. This leads us to the need of defining object compositions as part of the model. The formal definition of the composed object could be formulated as follows:

Definition 1. *A composed object is an object that is built from other objects using a composition entity.*

Definition 2. *As composition we will comprise the entity that defines the relation between super-object and sub-object. One composed object could be a super-object for one or more composition instances, i.e., be built by one or more sub-objects.*

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

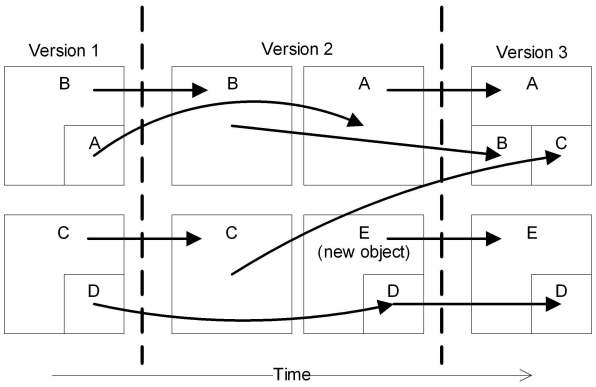

Fig. 1. Example of change in objects' content

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

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

- Versioned object id.

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

A versioned primitive is determined in a unique way using the pair **versioned object id** and **version number**. In spite of the possibility of using this unique pair and a compound key, good practices in ER model design [1] recommend that each entity should possess its own non-compound key. In our case we will introduce an additional field as a primary key – a global version number.

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

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

Here despite the fact that the attribute combination is always unique, we will also use an additional attribute for primary key – composition id.

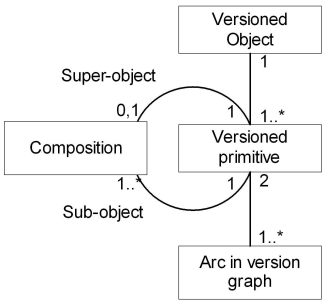

Fig. 2. ER model of versioned object

Due to needs of accounting and traceability of change, 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 – one entity for nodes and another one for arcs [11]. Looking at a versioned object definition, we could conclude that graph nodes correspond to a versioned primitive entity. The missing part will be implemented as a 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 who made the change.
- Date and time of change.

Versioning of composed object. In this paragraph we will present 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.

Definition 3. *A composed object with rank zero, i.e., a simple object is an object which is not associated with its own sub-objects. A composed object with rank N is an object for which the largest rank of associated sub-object is equal to* $N-1$.

$$R_{Object} = \begin{cases} 0, & \sum Subobjects = 0 \\ N, & \max(R_{Subobjects} = N-1. \end{cases}$$

The granularity degree of an object is its rank.

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

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

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

Rule 1. *In a given composition of composed objects, a certain object can occur at most once.*

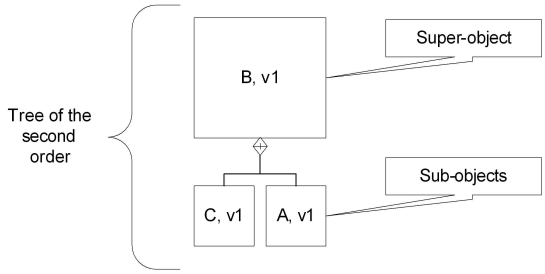

Fig. 3. Tree of objects

Rule 2. *One object can be part of at most one object composition.*

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

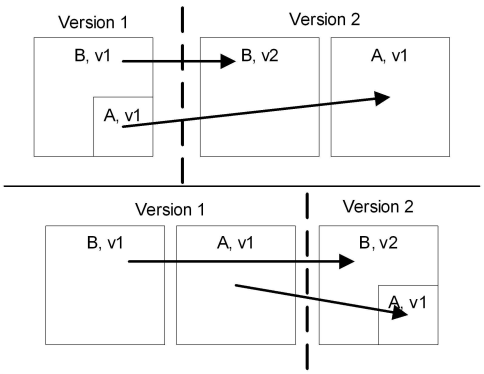

Fig. 4. Change in objects' composition through change of version

Another feature of composed objects is the case of sub-object change when we get an indirect change of the composed object (Figure 5). Let us look at the example: Let us have a change of a 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. The association of an object as a sub-object and removing of association

with a sub-object and its transformation to a simple object could be regarded as a special case of sub-object change.

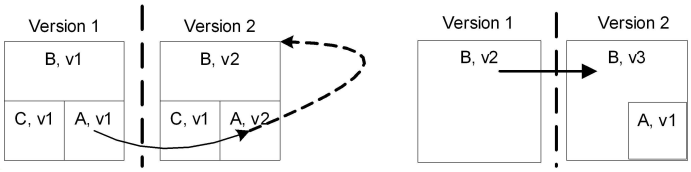

Fig. 5. Indirect change of version of composed object in case of its sub-object change

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

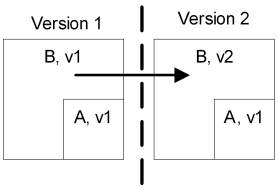

Fig. 6. Change in super-object doesn't affect the version of sub-object

A consequence can be deduced from the last two rules:

Consequence 2. *Changing the version of a sub-object for a super-object does not affect the versions of its sibling sub-objects* (*Figure* 5).

3. Model of Hierarchical composed workspaces. The first model we are going to examine is the model of hierarchical composed workspaces. Within the model the following definitions will be used:

Definition 4. *A product is the object of material or immaterial manufacturing, which after its creation can be reproduced and distributed to customers.*

Definition 5. *A product release is a fixed version that has passed a certain number of controls and meets the criteria of quality, safety and security. Only product releases are distributed to customers. Versions that are not released are called practice working versions.*

Definition 6. *A workspace is a location where certain activities are carried out on the development of a version of a product.*

Definition 7. *A main workspace is a workspace where ??the final preparation of equipment and product release is made.*

The arrangement (composition) of the workspace is taken to provide opportunity for each participant in the product development process and its releases to carry out activities both individually and in cooperation with other participants. It is the workspace that provides the opportunity for independent work which does not affect nor is affected by the work of other participants. On the other hand, composing workspaces in a hierarchy is claimed to be a mechanism which ensures cooperative work. Figure 8 presents a diagram of hierarchical composition of workspaces.

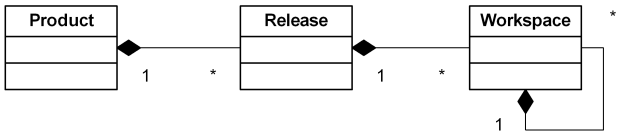

Fig. 7. Class diagram of Product-Release-Workspace model

Versioned object visibility model in environment with hierarchical composition of workspaces. As in any hierarchical structure, here, too, a parent-child relation will be considered. We will focus on the versioned object visibility defined by the following principles of visibility.

Definition 8. *The local version of the versioned object is the version that is associated with a specific workspace.*

Definition 9. *The visible version of versioned object for a given workspace is a version of the object with which the user can work.*

Visibility principles:

Principle 1. *The local version of the versioned object for a given workspace is the visible version of the object in the same workspace, despite other local versions in the parent workspaces.*

Principle 2. *The local version of an object in a workspace can be seen recursively in all subspaces, unless another local version is defined therein.*

From the above principles we can deduce some consequences:

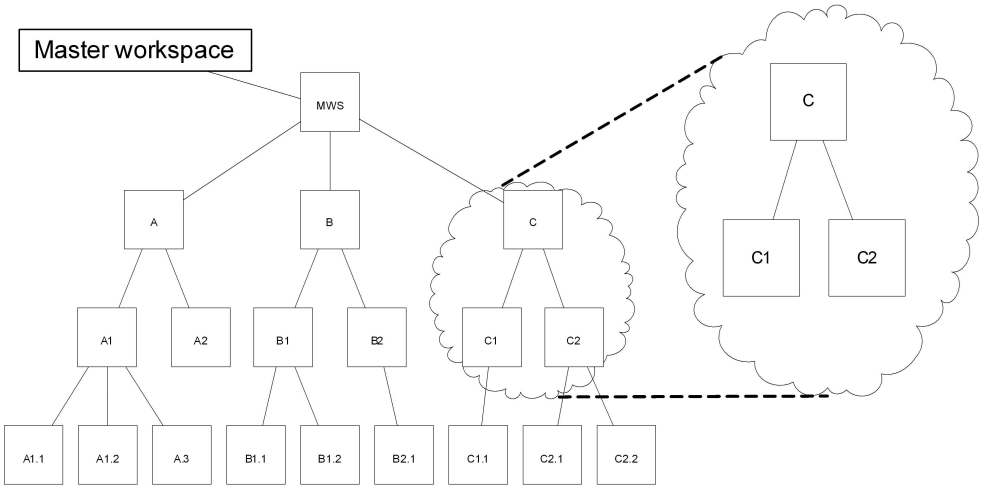

Fig. 8. Example of hierarchical composition of workspaces

Consequence 3. *In any workspace, where objects do not have a local version, they are presented with their version found in the nearest parent workspace.*

Consequence 4. *If for a given workspace the object has no version in either parent workspace, it is not visible in the selected workspace.*

In Figure 9 we present the way the two principles of visibility influence object version distributions, for example hierarchical composition of workspaces.

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

The composed object model and visibility principles lead us to the need to address the problem of composed object visibility.

Consequence 5. *A certain composed object version is visible in a certain workspace only and solely when all its sub-objects are visible in that workspace.*

4. Transactions over versioned objects. We introduced transactions over versioned objects in [7]. Here we will make a short presentation and classification of them.

bigskip

Transaction within a single workspace. Creation is the first transaction for each versioned object. After the completion of the transaction, the object

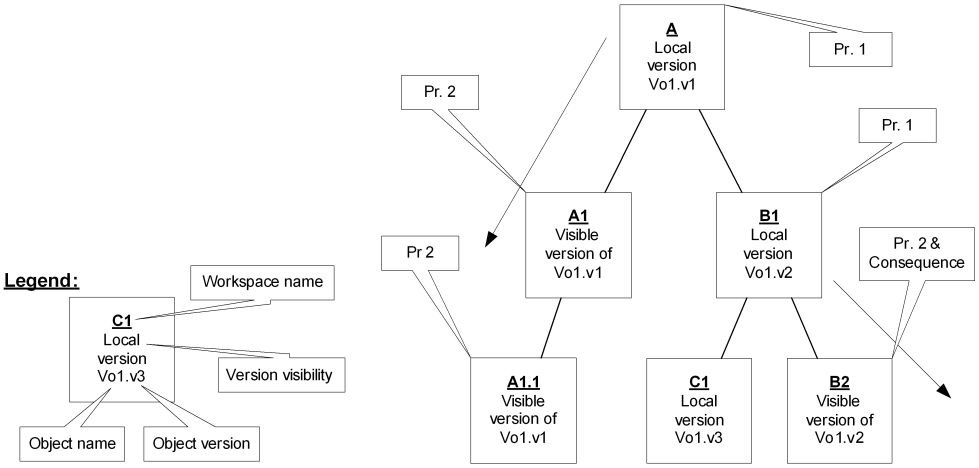

Fig. 9. Distribution of object versions according to the visibility principles

has an initial (zero) version, in which it is "empty", i.e., contains no information.

The transaction of state marker creation by 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 the model, the last state was released, and the current local version of the object is the version prior to refusal.

The creation of long sequences of unbranched versions, especially by 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 make analyses of the work faster and easier.

Updating a versioned non-local object, i.e., an object that does not have any local version in the current workspace, can be considered the most important part of the current section. This transaction is not entirely limited to a single workspace, because it is composed of the following steps:

- Retrieve the previous version of the site visible for the workspace.
- Create a local version of the object in the current workspace.
- Create an arc in version graph – linking the previous visible version with the new local version of the object.

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

respectively to make it invisible in its sub-workspaces. It should be noted that all transactions over the object described in this section are 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 an 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.

Definition 10. *Let us look at one versioned object and two versions of it, X and Y. If there is a path in the versioned graph of the object from version X to version Y, then version Y is called a derivative version of X, and version X is called a previous version of Y.*

Definition 11. *Let us look at one versioned object and two versions of it, X and Y. If there is no path in 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 of 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 10.

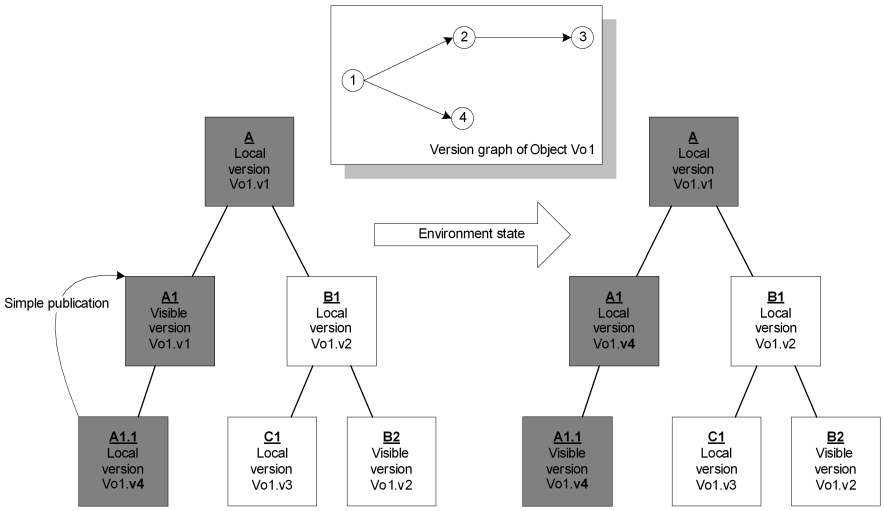

Fig. 10. Simple publishing

Following the transaction which needs to be addressed, is that of updating publication (Figure 11). It is typical of it that it is possible when it simultaneously satisfies two conditions:

- In the parent workspace there exists a local version of the object to be published.
- The version of the object to be published is a derivative of the version in the parent workspace.

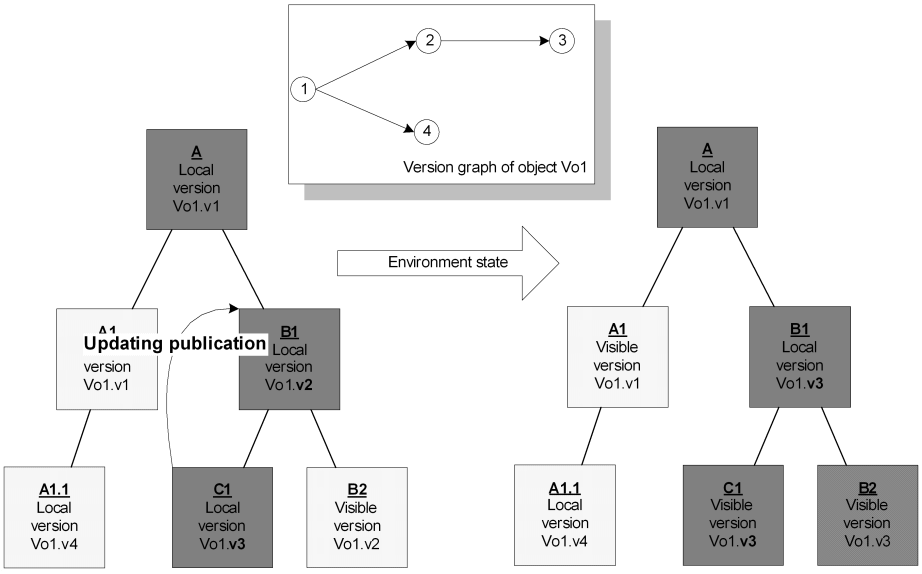

Fig. 11. Updating publication

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

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

The transaction of giving up of a local version is the reverse of the transaction of publishing a version. The only step in this transaction is the removal of the local version of the object on the workspace. It is important to note that

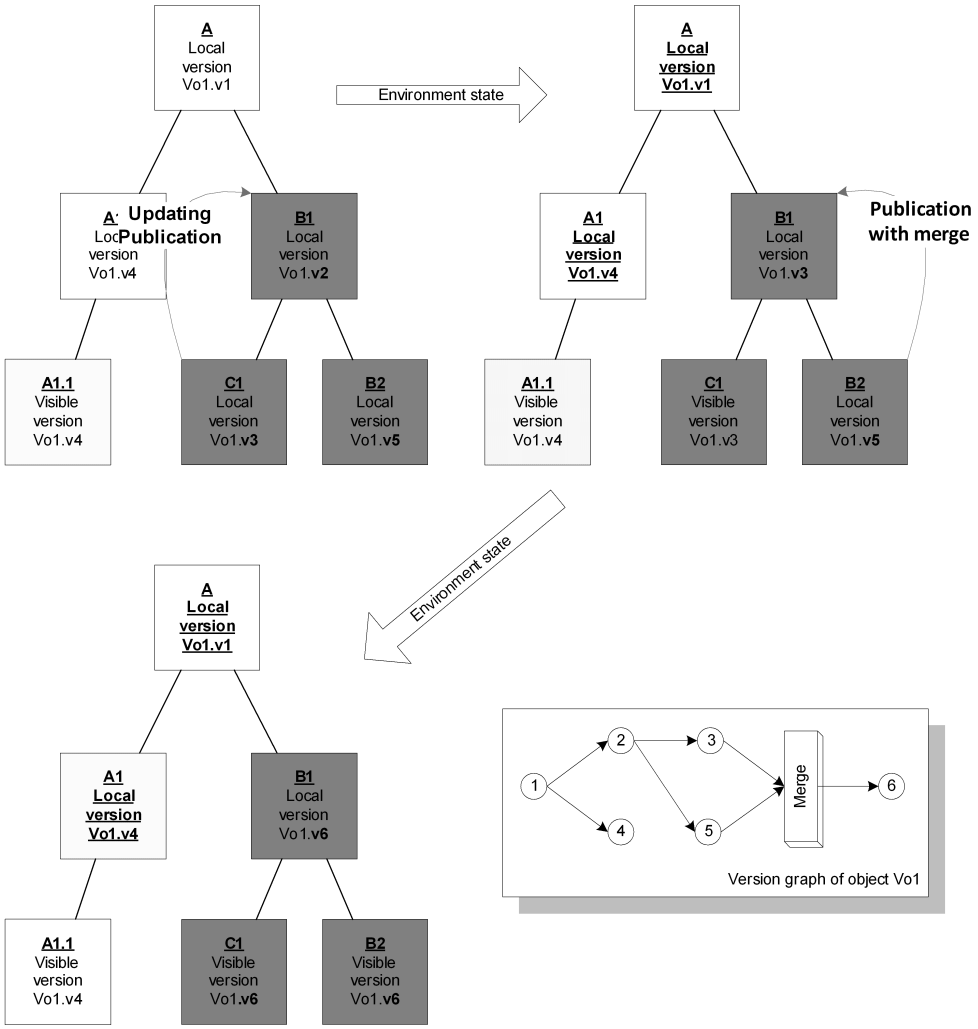

Fig. 12. Publication with merge

if any of the parent workspaces has no version of the selected object, the object becomes unavailable for subsequent use. This fact should be taken into account when the transaction takes place in the main workspace of the product's release.

Transactions over composed objects. Let us have the following situation: a local version of the object **B** in the parent workspace and its visible version in current workspace. Let us make in **A** a sub-object of object **B** in the

current workspace (Figure 13). After publication, the version of sub-object **A** may not lead to a change in the version of object **B** in the parent workspace. However, in a subsequent publication, the version of the object **B** in conjunction with its compositions in the parent workspace will lead to an automatic update of the compositional scheme of objects (in Figure 13 with a 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.

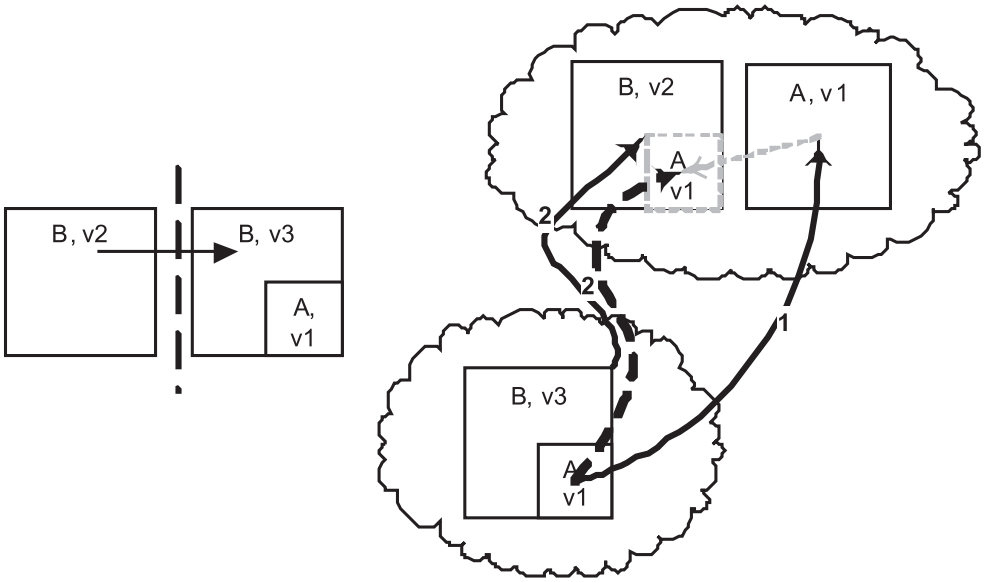

Fig. 13. Newly created sub-object to a super-object

The 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 13 – arrows with number 2).

Let us consider the situation where there is a local version of the object in the parent workspace that is visible in the current workspace (Figure 14). 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 of the new version of the sub-objects in the parent workspace should not be allowed. This restriction follows from the fact that a new version of a sub-object assumes a new version of the super-object (Figure 14 – arrow with number 1). In addition, we introduced the restriction that an object may be present in only one version in a workspace. In conclusion

to the situation we can formulate the following rule:

Rule 3. *The publication of a version of a 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* 14 – *the arrows with number* 2).

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

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

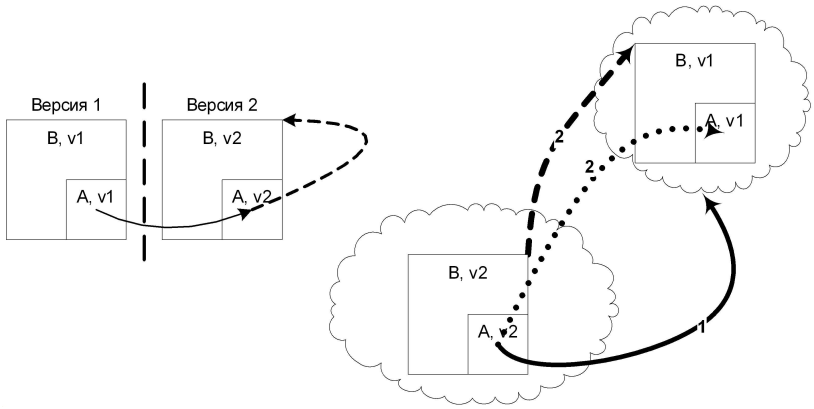

Fig. 14. Indirect change inversion of a super-object, caused by a new version of a sub-object

Let us have a visible composed object **A** with a sub-object **B**, object **A** and sub-object **B** being 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** (**??**). 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 object **A** should be with its version **v1** (visible or local version).

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

Rule 4. *Let us have an object's version that has a previous version, and that is a sub-object of a composed object in the parent workspace. The publica-*

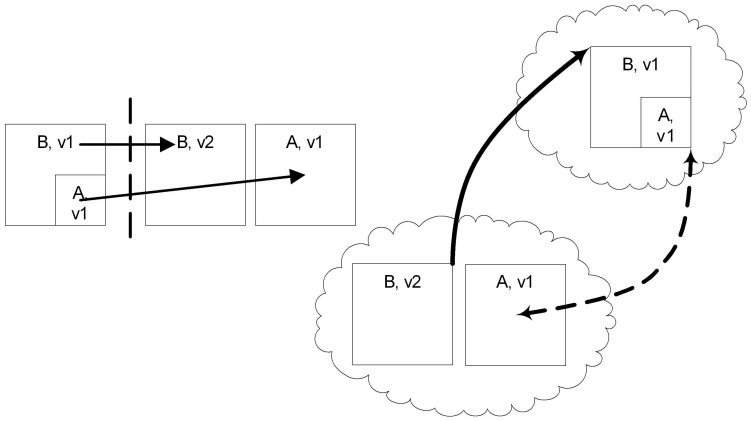

Fig. 15. Absence of change in the version of object A, i.e., there is no need of its publication

tion of that object has to be performed simultaneously with the publication of the composed object's version.

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

Rule 5. *The give-up of a local version of a composed object has to be performed in conjunction with a recursive give-up of all its sub-objects.*

5. Workspace environment configurations. The model of workspace composition presented above allows us to specify its appearance in the form of workspace environment configuration. By workspace environment configuration we will understand the process of determining the hierarchical architecture of workspaces. In Figure 17 and Figure 18 we present two examples of workspace configurations. These diagrams display the freedom of workspace arrangement in the most appropriate manner according to company architecture, project specifics, methodology characteristics, or other needs.

Figure 17 presents workspace composition where all mainstreams are divided into separate sub-trees—requirements, architecture, development and QA. Only requirements that meet the requestor's business needs are supposed to be published to the master workspace of the project. And only after that do they become visible to other project participants. The same scheme should be used for

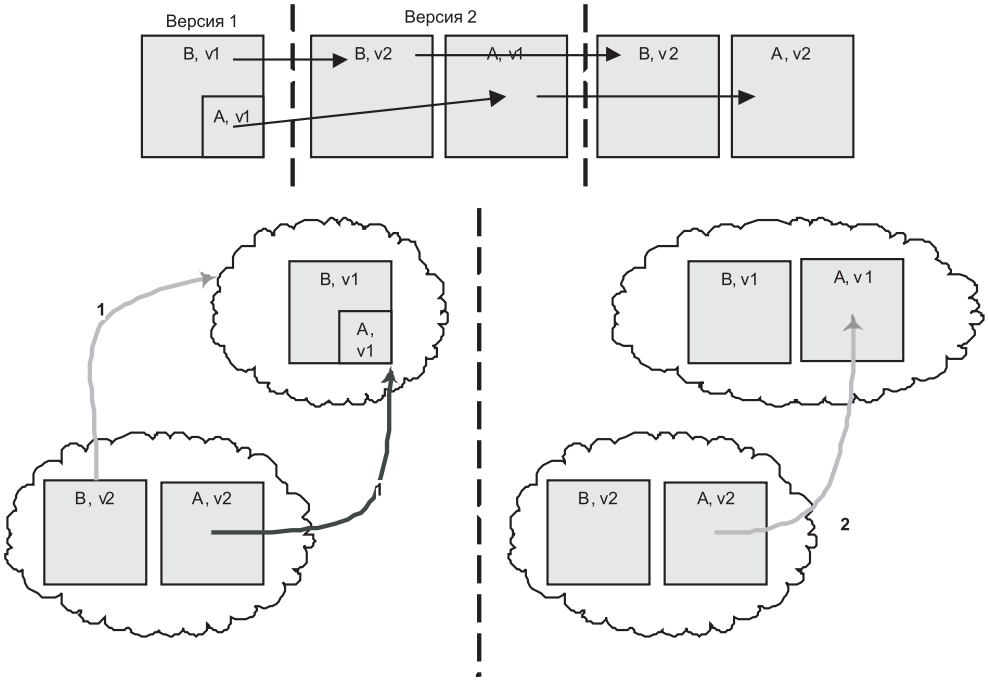

Fig. 16. Publication of a former sub-object is not possible before publication of the super-objects new version

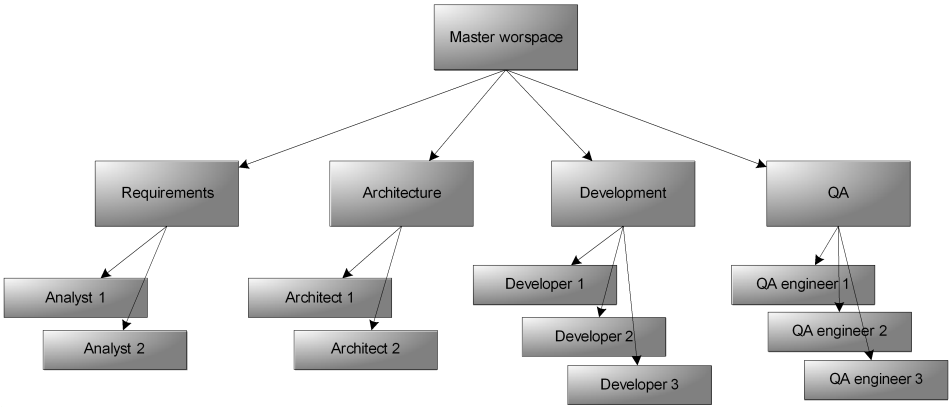

Fig. 17. Model of organization-driven workspace configuration

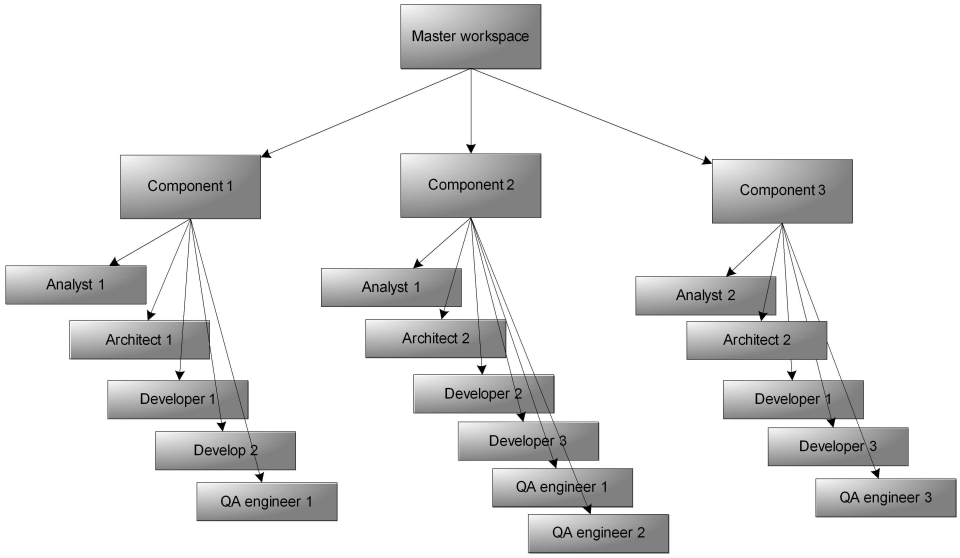

Fig. 18. Model of component-driven workspace configuration

the 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 a very topical issue in large projects.

In order to solve that issue on Figure 18, we present component-oriented architecture of workspaces. Here we have dedicated workspaces for each project participant and software components.

6. Conclusion and future work. The object-oriented model of a composed versioned object presented in this paper extends the potentials of versioning systems, allowing different level of data granularity. The model is a complement of the model of hierarchically composed workspaces including a set of rules for version control.

Versioning of complicated objects in complex environments becomes necessary in complex prolonged software projects. Modern software life-cycle methodologies put more emphasis on linking artifacts between different stages—requirements, test plans, project plans, etc. The existing version control systems do not provide a sufficient service level of change traceability among objects. We could formulate, as a future research direction, the investigation of traceability meta-data to support the accounting in versioning domain.

The issue of granting the respective data access level to versioned objects

shapes the theme of data security in versioning domain. This theme is a challenge for further development of the presented models.

REFERENCES

[1] Ambler S. W., Pr. J. Sadalage. Refactoring Databases: Evolutionary Database Design. Addison Wesley Professional, 2006.

[2] Collins-Sussman B., B. W. Fitzpatrick, C. M. Pilato. 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., B. Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, **30** (1998), No 2, 232–282. DOI= `http://doi.acm.org/10.1145/280277.280280`

[4] Estublier J., D. Leblang, A. Hoek, R. Conradi, G. Clemm, W. Tichy, D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, **14** (2005), No 4, 383–430. DOI= `http://doi.acm.org/10.1145/1101815.1101817`

[5] Git-Fast Version Control System. `http://git-scm.com`, 2012

[6] Jones M. T. Version control for Linux. 2006. http://www.ibm.com/developerworks/linux/library/l-vercon, 2009

[7] Jotov Vl. Transaction over Versioned Objects in Hierarchical Workspace Environment. ECAI09, Pitesti, Romania, 2009.

[8] Nguyen T. N. Model-based version and configuration management for a web engineering lifecycle. In: Proceedings of the 15th international Conference on World Wide Web (WWW '06), Edinburgh, Scotland, May 23 - 26, 2006, ACM Press, New York, 2006, 437–446. DOI= `http://doi.acm.org/10.1145/1135777.1135842`

[9] Price Derek R. CVS—concurrent versions system v1.11.22, 2006. `http://ximbiot.com/cvs/manual/cvs-1.11.22/cvs.html`, 2009

[10] Slein J. A., F. Vitali, E. J. Whitehead, D. G. Durand. Requirements for distributed authoring and versioning on the World Wide Web. *StandardView*, **5** (1997), No 1, 17–24. DOI= http://doi.acm.org/10.1145/253452.253474, 1997

[11] Stephens S. M., J. Rung , X. Lopez. X.: Graph data representation in oracle database 10g: Case studies in Life science. *IEEE Data Eng. Bull*, **27** (2004), 61–67.

[12] Sun Microsystem. Inc. The network software environment (NSE), Sun Tech. Rep. Sun Msicrosystems, Inc., Mountain View, CA, 104, 1989.

[13] Westfechtel B. Structure-oriented merging of revisions of software documents. In: Proceedings of the 3rd international Workshop on Software Configuration Management (Ed. P. H. Feiler), Trondheim, Norway, June 12–14, 1991, ACM, New York, 1991, 68–79. DOI= http://doi.acm.org/10.1145/111062.111071

Veliko Turnovo University "St Cyril and St Methodius"
Teodosij Turnovski str.
5003 Veliko Turnovo, Bulgaria
e-mail: vjotov@acm.org

Received August 7, 2013
Final Accepted August 26, 2013