Merging graph-like object structures

Albert Zündorf
Technical University of Braunschweig
Institute for Software
Gaussstr. 11
38023 Braunschweig
Germany
zuendorf@ips.cs.tu-bs.de

Jörg P. Wadsack, Ingo Rockel
Department of Mathematics and Computer Science
University of Paderborn
Warburger Str. 100
33098 Paderborn
Germany
[maroc|inro]@uni-paderborn.de

Keywords
Version Management, Object Orientation, Delta Computation, Optimistic Locking, Merge Algorithms, Long Transactions

1 INTRODUCTION

This paper addresses the problem of coordinating a team of software developers concurrently working on a common software system. The standard approach to coordinate concurrent activities on a common set of data is locking. Any part of data used by one person is locked against concurrent use by another person. The second person has to wait until the first person has finished his or her task and releases the lock. In databases, sophisticated locking and transaction concepts minimize the waiting times for concurrent users by offering different lock granularities and different locking levels (e.g. multiple read locks vs. single write locks). However, these locking strategies assume that locks are hold for relatively short times (some seconds), only.

In software development, version control systems like RCS [Tic85] or SCCS [Roc75] employing a pessimistic locking concept on a per file, per class, or per method basis are used. However, changes to source code may require some days or weeks. We call such changes long-transactions. Such long transactions lock certain source code parts for a long time. For example, one person may want to do a bug fix in a file which is locked by a second person. The first person may have to wait several days until he or she can proceed or he or she may negotiate with the lock owner. This requires extra efforts. In large development teams these extra coordination efforts may become a severe productivity problem.

To overcome these problems, optimistic locking concepts have proven very successful for software development in larger teams. CVS [Ced93, CVS] is a well known configuration and version management system that supports optimistic locking. Optimistic locking concepts allow multiple persons to change the same file, class, or method, concurrently. The management system just keeps track, which person works on which version of a given piece of software. When a concurrent change to the same piece of software happens, the management system computes a delta comprising the changes of one person and this delta is then applied to the version of the other person. If the changes do not actually conflict, the different versions are combined, automatically. In case of actual conflicts, manual interventions become necessary. Based on our experiences with a fairly big software project, the Fujaba (From UML to Java And Back Again) project, and a number of industrial projects, actual conflicts occur seldom (1 out of 1000 changes to a file within a 380 000 LOC project with about 500 classes and a team of about 15 developers results in a merge conflict). Thus, optimistic locking and merging mechanisms seem to provide a solution to the long transaction problem.

In large projects, a software system may not only consist of source code but also of various other documents like class diagrams, use case diagrams, project plans, and other design documents. Most of these design documents are stored in special binary formats. Unfortunately, most version management systems work with text files, only. They are based on diff and patch mechanisms that compute changes between two text files and that are able to apply such changes to a different version of that file, if that version has not changed too much. Thus, optimistic locking is usually not supported for design documents. As design documents become larger and more important, this becomes a severe problem for large projects.

Consequently, it is absolutely critical to have also object structure based delta and merging mechanism to support optimistic locking concepts for design documents. We will first discuss a merging mechanism for object structures. This merging mechanism banks on a textual representation of the object structure. This do not solve all emerging problems thus, second we will outline our advanced mechanism working on graph like object structures.

2 TEXTUAL REPRESENTATION BASED OBJECT STRUCTURE MERGING

A basic requirement for a textual persistence mechanism that shall serve as a basis for a merging mechanism is that reading a textual description and dumping it again without any modifications should result in exactly the same text. If the text differs, the textual merging mechanism will create textual deltas although no modification has happened. This creates a high likelihood that unnecessary merge conflicts are created.

Naive implementation of a textual persistence concept may change the textual representation of an object structure e.g.

due to different visiting sequences of the contained objects. Let us for example assume, that an object stores a set of references to neighbors within a hash table or tree based container that uses the address or hash code of the stored object as access key. Typically, the reading mechanism of a textual persistence concept has no or little influence on the address or hash code of re-created objects. Thus, the addresses or hash codes of re-created objects may differ from the addresses or hash codes that have been used during storing the object structure. Accordingly, the order in which a re-created container retrieves the same set of object differs from the original order used to store the objects the first time. If the order of the objects changes in the textual representation of the object structure dramatically, the textual merging mechanism is screwed.

Similarly, the change of object identities (ids) during the re-creation of an object structure from its textual representation creates a problem for the unchanged representation of references to neighbors. If this representation is based on the object id provided by the runtime system, all object ids may have changed on dumping the object structure, again. This screws the textual merging mechanism, too.

To solve these problems, the textual consistency concept should employ its own persistent object ids that must not change on multiple dumps and readings of an object structure. In addition, these persistent object ids should be used as sorting and hashing criterias for containers storing sets of references to neighbor objects in order to achieve a stable visiting order during the textual dumping.

However, using persistent object ids may not suffice to guarantee a certain visiting order for containers of references. Hash based containers may reorganize themselves on the insertion or deletion of objects. Thus, inserting one new object to such a container may change the textual representation of the whole object structure, dramatically. This may be avoided by sorting the object based on their persistent keys during the dumping process.

Another group of problems is related to independent additions of objects to an object structure by different users. Usually, we would expect that the merging mechanism should merge these independent changes without problems. However, naive text dump mechanisms tend to append new objects at the end of the object structure. This may for example happen if persistent object ids are created in consecutive order and if these ids are used as sorting criterias for the textual dump. If this happens, concurrently created new objects will both be inserted at the end of the corresponding text dump. Thus, a text based merging mechanism will have a merge-conflict due to concurrent changes at the same text position, although the changes do not actually interfere.

Similarly, a severe problem is created if different users create different objects concurrently and if these objects get the same persistent object ids, by accident. If the textual representation of these different objects are placed at different positions in the dumped text, the merging algorithm may merge these changes without any problems. However, this may create a textual representation that uses the same persistent object id, twice. This creates severe problems for the re-creation of the object structure.

To solve these problems, separated name spaces for persistent object ids created in concurrent sessions should be employed. A simple way to achieve this is to employ a unique session id which is used as a basis (prefix) for the persistent object ids created in that session.

If one user has at most one session at a time, we could use permanent user ids as name space prefixes. This would also address the problem of finding independent insertion positions for new objects. If we prepend the user id to the persistent object id and if we use lexical order based on the persistent object id, each user id gets its own insertion compartment. Objects created by different users (in different sessions) would be inserted into the text dump at different positions, thus avoiding unnecessary textual merge conflicts. However, they still have to take care that concurrently created new user compartments are inserted at different text positions.

Dumping and reading the object structure

Due to the discussion above, built-in serialization mechanisms provided by modern programming languages are usually not appropriate as a basis for textual merging mechanisms. Thus, a new textual persistence mechanism has to be build.

A standard way to implement such a textual persistency concept is to employ a base class that provides appropriate read and write methods for all attributes of that class. All classes that shall become persistent inherit from this base class and extend the read and write methods in order to cover the attributes introduced in that class. The persistent base class should introduce an attribute for the persistent object id of the objects and perhaps static attributes to store the session id and a counter used to create new unique persistent object ids. Dumping an object structure just starts with a root object. Reference attributes are dumped using the persistent object id of the target object and later on the target object is visited recursively. Some kind of marker or an object table may be used to avoid multiple dumps of a single object. The read procedure needs the type of the persistent objects in order to create new instances of the correct class. Then, the read method is used to restore the attribute values. In case of reference attributes, a look-up table is employed to turn the persistent object id into a reference to the corresponding object.

Due to our experiences the read/write method approach creates serious maintenance problems. Each time a new attribute is inserted into a persistent class, it has be extended to the corresponding read and write methods, accordingly. This is easily forgotten. Another problem are library classes like *Color* or *Point*. It is not allowed to modify such library classes in order to make them persistent. In Java, the additional problem that multiple inheritance is restricted to interface classes is faced. Thus the inheritance of persistency properties may become complicate.

We overcome these problems by using a generic, table driven persistency mechanism. Implementing in Java allows us to exploit Java runtime type information to inspect instances of persistent classes and to store and recover their attributes.

The classes are marked as persistent either by inheriting from a certain base class or by adding it to a persistent class table. The latter mechanism allows to cover library classes, too. To exclude certain attributes from the persistency mechanism, the Java keyword *transient* may be used. Similar information may be provided by the persistency table. This generic persistency mechanism is easily adapted for new programs. In addition, the mechanism solves the maintenance problem caused by the introduction of new attributes.

Remaining merge conflict problems

The persistency and merging mechanism described so far works already fine for many situations. However, some principle problems remain.

Consider for example a cutout of the abstract syntax graph of an UML CASE tool. The base version may just consist of a an empty class diagram. User `maroc` may now enter a class Customer and user `zuendorf` a class Car. These independent changes should not create a merge conflict. If we have prepared different text compartments for different users, the addition of the new UMLClass objects to the text dump will not create a problem. However, in our meta model the class diagram holds a set of references to all contained classes in its attribute items, cf. Figure 1. The addition of a

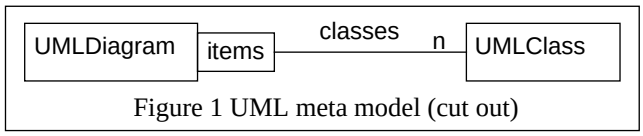

Figure 1 UML meta model (cut out)

class to a class diagram adds a reference to the items attribute, too. Thus, after the addition of one class to the class diagram, the textual dump of the class diagram object will contain one line for the items attribute describing the reference to the added class. Unfortunately, for both users this line is added at exactly the same position in the text dump, cf. lines 3 to 5 in Figure 2. Thus, a textual merging mechanism will report a merge-conflict, which actually does not exist.

```
1) /* text dump from user maroc */
2) ... // in class UMLDiagram
3) <attribute> <id>2.3</id> <type>Attr</type>
4)   <name>items</name> <value>Customer</value>
5) </attribute>
   ...
1) /* text dump from user zuendorf */
2) ... // in class UMLDiagram
3) <attribute> <id>1.3</id> <type>Attr</type>
4)   <name>items</name> <value>Car</value>
5) </attribute>
```
Figure 2 Position conflict

Let us now assume, that we are able to overcome this problem by some reorganization of the text dump. Thus, user `zuendorf` and user `maroc` may have successfully merged their two classes Car and Customer. They both take this common version and concurrently add a class Contract. If we are able to work around the insertion problem to the items attribute of the UMLDiagram object, a textual merging mechanism would probably merge these concurrent changes without any problems. However, this would create the problem that the merged class diagram contains two independently created classes with the same name. In Figure 3 the merged class diagram is shown as (XML-) text dump, in lines 5-7 and 11-13 a class Contract is declared. In our meta model (cf. Figure 1), class UMLDiagram employs a qualified association items to hold the contained classes. The corresponding cardinality constraint guarantees unique qualifiers. This means, our data model does not allow to store two classes with the same name within one class diagram. Thus, in our example the reader would have a problem to restore an object structure where a successful textual merge has added two classes with the same name. This is shown in Figure 4 for classes Contract.

```
1) ... // in class UMLDiagram
2) <attribute> <id>2.3</id> <type>Attr</type>
3)   <name>items</name> <value>Customer</value>
4) </attribute>
5) <attribute> <id>2.14</id> <type>Attr</type>
6)   <name>items</name> <value>Contract</value>
7) </attribute>
8) <attribute> <id>1.3</id> <type>Attr</type>
9)   <name>items</name> <value>Car</value>
10) </attribute>
11) <attribute> <id>1.16</id> <type>Attr</type>
12)  <name>items</name> <value>Contract</value>
13) </attribute>
```
Figure 3 Merged class diagram

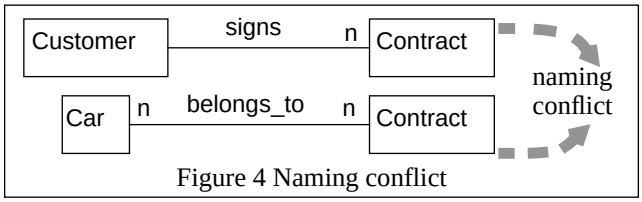

Figure 4 Naming conflict

To summarize, a text based merging mechanism has no knowledge about the semantic constraints introduced by the meta model of our object structure. Thus, the textual merging mechanisms is not able to deal with a number of merge problems related to such constraints. Consequently, we have developed an object structure based merging mechanism, described in the next chapter.

3 OBJECT STRUCTURE BASED MERGING

Our object structure based merging mechanism is based on a simple meta model providing objects, labeled links, and

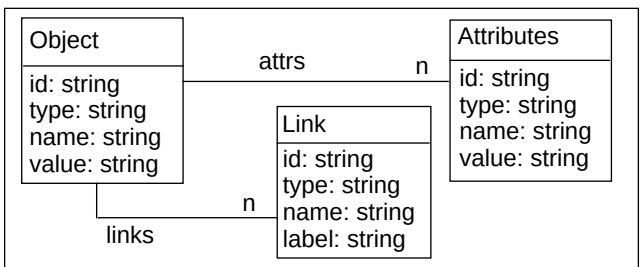

Figure 5 Text dumping meta model

attributes, cf. Figure 5. The elementary change operations of this meta model are

- create / delete an object
- create / delete a link
- change an attribute value

```
algorithm delta(A, B)
1) input A,B : set of object structures
2) output D : delta of the object structures
3) local variables OA, OB, ON, OD: object
4) local variables LA, LB, LN, LD: link
5) begin
6)   for all (OA,OB) = pairWithSameID (A,B)
7)     if OA.attrs ≠ OB.attrs
8)       then D.addChangeAttributeEntries (OA,OB)
9)     fi
10)    for all LB = (OB.link - OA.link)
11)      D.addCreateLinkEntry(LB)
12)    for all LA = (OA.link - OB.link)
13)      D.addRemoveLinkEntry (LA)
14)  for all ON = (B - A)
15)    D.addCreateObjectEntry (ON)
16)    D.addChangeAttributeEntry (ON.attrs)
17)    for all LN = (ON.link)
18)       D.addCreateLinkEntry (LN)
19)  for all OD = (A - B)
20)    D.addDeleteObjectEntry (OD)
21)    D.addChangeAttributeEntry (OD.attrs)
22)    for all LD = (OD.link)
23)       D.addRemoveLinkEntry (LD)
24) end
```

Figure 6 The delta algorithm

Computing Object Structure Deltas

We first describe an algorithm computing a delta between two object structures in terms of these elementary change operations, cf. algorithm *delta* in Figure 6. The delta algorithm employs two sets of known objects for object structures A (old) and B (new) that are to be compared. Our algorithm iterates through the sets of known objects of the two object structures and retrieves pairs of objects with the same persistent object ids. For each such pair of objects OA and OB we compare the set of basic attributes, cf. line 7. If the attribute values differ we add an appropriate changeAttribute (line 8) entry to the computed delta. Then we consider all links attached to the objects (lines 10 to 13). Two links with the same label are considered equal if they target objects with the same persistent object id. If object OB has a link with label L targeting and object OX and OA does not have such a link, then an appropriate createLink entry is added to the delta (line 18). If a link is removed we add a removeLink entry to the delta (line 13).

Once all pairs of objects with the same persistent object id have been considered, our algorithm considers the remaining objects (line 14 to 18). Each object ON existing in object structure B but not in object structure A has been added to object structure B. Thus we create corresponding createObject and changeAttribute entries to the delta (lines 15 & 16). In addition, we add all links attached to ON to our delta (lines 17 to 18).

Objects contained in object structure A but not in object structure B have been removed in object structure B. For such objects OD we add deleteObject entries to the delta. In our approach we add changeAttribute entries for deleted objects to the delta, too (lines 20 & 21). Although, this information may not be necessary for the merge process, it allows to use the delta as a reverse delta, if required. This means, the delta can be used to reconstruct object structure A from its successor object structure B. Accordingly, we add deleted links attached to deleted objects to the delta (line 22 & 23).

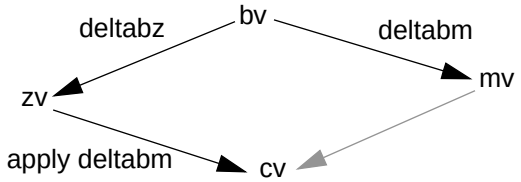

Figure 7 Computing common version by applying deltabm to zv

3-Way Merging / Applying an Object Structure Delta

Let us assume, we have a base version bv of an object structure and users `zuendorf` and `maroc` create two successor versions zv and mv, concurrently. We would now like to merge these changes into a common version cv, cf. Figure 7.

Basically, the common version cv may be computed by applying deltabm to version zv. A delta is applied to an object structure by „just“ executing the corresponding create/delete object, create/delete link, and change attribute entries of the delta. Again we need a lookup table to turn persistent object ids from the delta into references to actual objects. We use session or user id prefixes to the persistent object ids in order to avoid concurrently created objects with equal persistent ids as described for textual merging mechanisms. If no conflicts occur, this algorithm works straight forward and creates an object structure cv that contains the changes of both users. Most of the problems discussed for textual merge mechanisms do not occur. For example adding a class created by user `maroc` to the class diagram does not create a merge conflict, since the items set has no dedicated insertion position where a conflict could occur. However, there are still conflicts possible.

Conflicts on object structure based merging

The simplest conflict occurs if both users have modified the same basic attribute, e.g. both users change the name of an existing class in different ways. Similarly, createLink operations may conflict if the corresponding association has cardinality to-one. It may even be considered as a conflict if one user deletes a to-one link and another user replaces that link by a link to another target. In addition, one user may have deleted an object while the other user changes an attribute of that object or adds a link to that object.

An object structure based merging mechanism must detect such situations and deal with them, somehow. A simple approach is to always prioritize the changes of one of the users. Another idea is to compute a kind of merged result. For string based attributes one could just concatenate the conflicting names. In our CASE tool example this would result in a class with a concatenated name. The user could identify this problem within the CASE tool and resolve the conflict manually. Generally, this is a very dangerous approach. Probably, a special table of attributes that shall be handled this way should be used. However, conflicts related to to-one links and conflicts related to deleted and concurrently modified objects can not be solved this way. In general, such merge conflicts require a user decision.

Unfortunately, our object structure based merging mechanism may report merge conflicts on an object structure

level, only. The user of a CASE tool may have some difficulties to resolve a conflict saying „conflicting assignments to attribute age of object 2.33 of type XYZ“. The user would probably like to deal with such conflicts via the GUI of his or her CASE tool.

4 CONCLUSIONS AND FUTURE WORK

There have been a number of approaches trying to improve merge mechanisms for software documents, [Wes91b, YHR92, LvO92, BHR95]. Most of these approaches tried to exploit higher level semantic knowledge in order to deal with merge conflicts more sophisticatedly. Like our approach, many of these approaches create an object structure based representation of the software documents in order to analyse the changes more thoroughly. Our approach has been heavily influenced by the work of [Wes91a]. [Wes91a] introduced the idea of unique persistent object ids as a basis for object structure based merging. Somehow, we follow the idea of operation based merging of [LvO92]. However, [LvO92] requires the recording of change operations while we employ a delta algorithm. We assume that explicit operation recording is hard to implement.

We think that our approach improves the object structure merging problem by introducing a very generic concept that will be easily adapted to new object structures and tools. We plan to develop tool support, that allows to reverse engineer existing object structures and to generate a merge mechanism for such structures.

We plan to deal with the merge conflict problem using a table driven approach. A configurable conflict resolution table shall allow to define different conflict handling strategies for different kinds of conflicts on different types of objects, links, and attributes. One conflict handling method could be to add appropriate conflict marker objects to the object structure. Then, the CASE tool could be extended by a dialog presenting the merge conflicts in an appropriate way and allowing the user to enter merge decisions on a logical level. In this way, a generic merge mechanism may be integrated with different tools.

We have implemented the object structure based merging mechanism in our Fujaba CASE tool. This implementation was not as simple as described. First of all, our approach assumes a graph-like object structure. This means, it must be possible to delete some object and to determine all other objects referencing to this object and to reset all these obsolete references in order to avoid dangling references. In our implementation, all references are implemented as pairs of forward/backward pointers. This allows to determine all neighbors of a deleted object, easily.

Second, we assume that each object has a unique persistent number even across session boundaries. Especially, this does not allow to compare independently created object structures. Basically, this idea stems from [Wes91b]

Third, we assume that some adding or deleting of objects or links and some attribute modifications turn a valid object structure into another valid object structure. Theoretically, this holds, since the resulting object structure respects all constraints imposed by the corresponding meta model. For example, no coordinately constraints of associations are violated. However, usually tools impose other hidden consistency constraints on object structures. For example an explicit attribute, counting the number of existing classes may be employed. Adding and removing class objects should change this counter, too. This is not known to our merge mechanism. Due to our experiences, tools that are able to read a text based representation of their object structure are usually well prepared to deal with object structures that can be represented by such a textual description. Thus, we plan to apply our mechanism to XML based object structure representations.

We have minimized such constraints in our Fujaba environment. We have been successful for class diagrams, but there are still some problems with some behavior diagrams. Similarly, we have a preliminary version of a merge dialog, only. We would like to apply this approach to other tools that have a „long transactions“ problem, in order to validate its feasibility and its genericity.

For the Fujaba environment see
`http://www.uni-paderborn.de/cs/fujaba/`

REFERENCES

[BHR95] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[Ced93] P. Cederqvist. *CVS Manual: Version Management with CVS*. Signum Support, 1993.

[CVS] CVS. *Concurrent Versions System - The open standard for version control*. http://www.cvshome.org/.

[LvO92] E. Lippe and N. van Oosterom. Operation-based Merging. In *Proc. of the 5th Symposium on Software Development Environments (SDE5)*, volume 17(5), pages 78–87, Tyson’s Corner, Virginia, December 1992. ACM SIGSOFT Software Engineering Notes.

[Roc75] M.J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[Tic85] W.F. Tichy. RCS - a system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

[Wes91a] B. Westfechtel. *Revision Control in an Integrated Software Development Environment*. PhD thesis, Aachen University of Technology, Aachen, Germany, 1991.

[Wes91b] B. Westfechtel. Structure-Oriented Merging of Revisions of Software Documents. In P.H. Feiler, editor, *Proc. of the 3rd International Workshop on Software Configuration Management*, Trondheim, Norway, June 1991. ACM Press.

[YHR92] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3):310–354, July 1992.