

# CoObRA - a small step for development tools to collaborative environments

Christian Schneider, Albert Zündorf  
University of Kassel, Germany  
christian.schneider@uni-kassel.de, zuendorf@uni-kassel.de

Jörg Niere  
University of Siegen, Germany  
joerg.niere@uni-siegen.de

## Abstract

*Working on large software projects commonly requires software development in teams, which requires a tight interaction within the development team. Usually each developer is responsible for a certain part of the whole system. Therefore, it is necessary to let developers work independently without disturbing their teammates as well as allowing them to share their results with their teammates at a certain time and to integrate the developed software parts.*

*Common practice is to use CASE-tools to support large software projects and a Software Configuration Management (SCM) system that supports team coordination. Popular SCM systems are usually based upon text files and fail in versioning binary files or object-oriented structures on a fine-grained level, as usually produced by CASE-tools. On the other hand, current CASE-tools insufficiently support versioning and team coordination.*

*CoObRA is a framework that offers undo/redo-, persistency-, multi-user-support and version control techniques for applications based on object-oriented data models in general. The major advantage is that it provides an easy-to-use mechanism requiring very low integration costs. CoObRA was successfully integrated in the Fujaba Tool Suite and is also part of Fujaba's code generation facilities that offer CoObRA's features to applications generated with Fujaba. However, CoObRA may easily be used outside Fujaba, too.*

## 1 Introduction

Most software development processes require several kinds of diagrams in the different development phases, e.g. the Rational Unified Process [JBR99] or the Fujaba Unified Process [DGZ03]. Therefore, state of the art tools for Computer Aided Software Engineering (CASE) provide a broad range of diagram editors.

It is already common practice to use Software Configuration Management (SCM) systems for source code. But current CASE-Tools commonly lack full SCM support for

the diagram data, though this is necessary - especially for large software projects. This has a simple reason: the development costs for supporting versioning and merging are very high for most CASE-Tools. Similarly, features like Undo/Redo, recovery mechanisms, and multi-user white board co-operation require considerable implementation efforts.

Throughout the rest of this paper we will focus on applications that use object-oriented data models, which includes most diagram editors and CASE-tools.

Storing application data in systems based on object-oriented data models means either storing binary data, which cannot be merged, or storing a textual representation (e.g. XML), where other problems arise. Most of them are related to detecting and solving conflicts. In addition, in serialized object-oriented data structures simple operations like adding two different objects concurrently may accidentally produce conflicts while merging, as data for both objects may be written at the end (the same position) of each (text) file, cf. [ZWR01]. Conventional SCM systems will not be able to handle such situations correctly as they do not have semantic information about the application data.

CoObRA addresses these shortcomings of common SCM systems by providing a persistence and versioning mechanism which supports merging of versions on object level. Another main aspect of CoObRA is the support of an optimistic locking concept in order to allow engineers to have their own copy of all project data and to work independently from a permanent connection to a repository.

The following chapter shows our general usage of deltas in object-oriented data models. Chapter 3 discusses technical realization. We continue with some experiences in chapter 4. This is followed by discussion of references and by conclusions and futures work.

## 2 The CoObRA approach

In our approach the application data is considered to be completely described by the changes that have effected on an initially empty object-oriented data structure. Therefore, we have to characterize object data.

## 2.1 Object data characterization

CoObRA assumes that an object structure snapshot consists of a set of objects, where each object is specified by

- the class of the object
- the values of its plain fields (plain attributes)
- the content of multi-valued fields (collection attributes)

The values of plain and multi-valued fields may either be of basic type as e.g. boolean, integer, or string, or of reference types i.e. pointers to other objects.

A field is considered to be a logical property of an object. Depending on the used programming language such a logical property is usually implemented by a programming language field (storing the actual data) and a set of encapsulating access methods. Multi-valued fields will be called collection fields, where the collection may either be of type set, list or map (qualified field).

According to this ‘simple’ object structure representation the alteration to the application data can be split into atomic changes of the following types:

- creation of an object
- removal of an object (destruction)
- altering a field value (of a plain field)
- adding/removing a value to/from a collection field (including changes to maps)

## 2.2 Concept

Instead of storing the state of a whole object structure, CoObRA logs the changes to this structure starting at its creation. These changes are stored in a repository, available for further processing. To provide undo functionality, beside the new value for a field, the old value is stored on field alteration, too. With this additional information every single change can now easily be reversed:

- to revert the creation of an object, it is destroyed
- to revert the removal of an object may require restoring all field value depending on the used implementation of the removal (see 2.3)
- to revert altering a field value (of a plain field), the field is set to the old value
- to revert adding a value to a collection field (including changes to maps), the value (key) must be removed again
- to revert removing a value, it is added again

To provide a persistency service for an application, the changes may be written to disk or database. To do this, (locally) unique identifiers for all persistent objects are required. Additionally, a mechanism to serialize basic field values in an appropriate form has to be provided (commonly by a library). Along with persistency CoObRA may provide a recovery service by writing each change to disk or database, immediately. Upon next launch of the application, the incomplete change log can be detected and the last status of the application data may be restored<sup>1</sup>. Restoring from a repository means loading the changes list and re-playing the changes (redo).

In addition, the changes may be used to enable co-operation among multiple application instances. All recorded changes are forwarded from one application to the other (and in reverse direction). The receiving application instance simply replays the changes. Thereby the two applications work synchronously on replicated object structures in a white-board-mode. (How we deal with conflicts of concurrent modifications is discussed below.)

As the persistency mechanism already stores changes, no computation of deltas between different versions of an object structure is necessary. Versioning may easily be set up by grouping several changes and by labelling them with version numbers. This versioning may happen locally or it may be managed by a server. The latter employs a checkin/update concept as known from conventional SCM systems.

The key feature of CoObRA’s version management approach is its optimistic locking concept. As stated in the abstract, we want to enable developers to work independently from their teammates for a certain period of time (some days, perhaps off-line). They work on their local copy of the project data. Only at certain points in time, e.g. after finishing some extension or modification, they may send their contributions to the other team members or they may incorporate the changes of other team members in their local copy of the project data. Actually, we will use a project data repository for coordinating the contributions of the various team members. Each of the clients synchronizes its local data with this so-called version repository in two steps called update and checkin.

In the following explanations, we assume that there exists a versioning repository containing already, e.g. 40 versions of the data of some software development project. In addition, there are two developers Mary and Bob which both have two copies of the current version 40 of the project data on their personal computer. Now both developers work for about a day on two different software development tasks.

---

<sup>1</sup>which does not include the execution stack

### 2.2.1 Checkin

We assume, Mary finishes her work first and she now sends her changes to the versioning repository using a checkin command. Obtaining the changes between Mary's current version and her starting point version 40 is easy, since CoObRA protocols all changes already. However, since the change data is sent to a repository on another computer and it is later merged with changes of other applications, we need to ensure that objects created in different applications are serialized using different unique identifiers. Therefore, each client have to obtain unique identifiers from the version repository, first. Now, the changes are serialized and sent to the version repository. In our example, this creates version 41 in the repository and Mary is now synchronized with this version 41.

### 2.2.2 Update

Now Bob finishes his work. Before he is allowed to send his changes to the repository, he must first merge his work with the contributions of other team-mates. Therefore, CoObRA compares the version number of his last synchronization (in our example 40) with the current version number of the repository (in our case 41). Then the changes that have been contributed by other team members, meanwhile, are sent to Bob's application.

Bob's application could now just replay the received changes. However, these server changes may have so-called merge conflicts with some of Bob's local changes, e.g. a field may have been changed on the server to one value and in Bob's local data to another (for example see 2.2.3). Or in one version an object has been deleted that is modified in the other version. In CoObRA, changes that have been accepted by the repository get a higher priority in the case of such merge conflicts. This means, CoObRA enforces the server changes on the clients local copy. To achieve this, CoObRA's update operation first resets the local object structure to the version of the last synchronization by reverting all local changes (but keeping the corresponding delta). Now the changes retrieved from the server are applied without any merge conflicts. In our example, Bob's local project data reflects now version 41.

Now, the client's local changes are applied, again. In case of merge conflicts, the corresponding changes are not executed but kept in a separate list. This list is returned to the application, which can e.g. show them to the user who may resolve them, manually. After update and conflict resolution, Bob may now send his changes with respect to version 41 to the version repository using a checkin command. This results in version 42. (If Mary or some other team-mate has already checked-in a version 42, meanwhile, Bob has to go through another update/merge cycle. This happens seldom.) At a time of her convenience, Mary or any

other team-mate may incorporate the changes of version 42 in her local copy (latest before her next checkin).

### 2.2.3 Merge Conflict Example

CoObRA reports merge conflicts to the application, only. This generic conflict report reflects the applications internal data model. Usually, a CoObRA based application has to analyze these generic conflict reports in order to present them appropriately to the user's view.

For an example, imagine a master repository containing a class "Level". Client A and client B retrieve the current version. They do now both change the class name of their local copies, concurrently. Client A changes it to "Storey" and client B changes it to "Floor".

We assume that client A is faster and checks in first. Client B cannot checkin as the server contains more changes than client B already knows. Client B has to update its local repository. The class name of its local object is reverted to the original "Level" as B's local changes are reverted. Then the new changes from the server are replayed setting the class name to "Storey". Finally B's local changes are replayed, too. Doing this a conflict arises: The class name has already been altered to "Storey" and can not be set from "Level" (the 'old' value) to "Floor".

CoObRA reports this conflict with a generic message like 'field "Name" of object "#44.565" of type "Class" has been changed from "Level" to "Storey" in version "41", local change to "Floor" discarded'. The client application has to inform its user of the conflict and provide appropriate solution strategies. In this case the user has to decide which spelling is correct.

### 2.2.4 Large changes lists

If an application logs changes to its object structure over some time. The list of changes may grow to a significant size. As long as the application wants to provide full undo functionality this has to be accepted as functionality costs. However, if only persistency is required, the changes list should not be significantly larger than another serialization method. To achieve this, the list of changes is compacted. This means subsequent changes to the same field are summarized and changes to objects that were removed are discarded. This includes mutual elimination of add- and remove-changes and combination of several alterations to one field.

If we use a version repository, we need to keep track of all relevant changes between two version labels. However, the change list between two version labels may still be compacted as above in order to reduce the size of the repository and the size of deltas that are transferred on checkin and update.

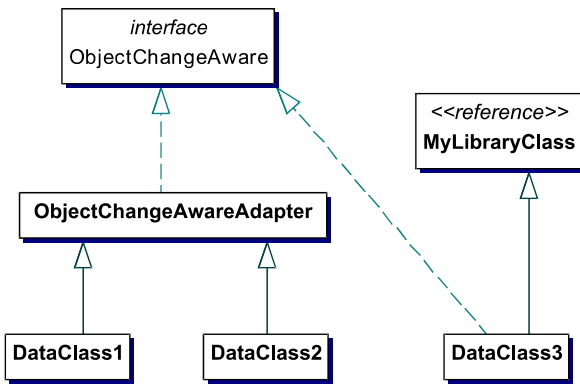
### 2.2.5 Grouping of changes for user level undo/redo

The change information recorded by CoObRA has a very fine granularity by default. But if a user of an application clicks on the undo button in the toolbar he expects his last action to be undone. To make an application capable of this behaviour the changes have to be grouped by user actions. This can easily be achieved if the command pattern [GHJV95] is used in the application: upon each start of a command a new group for the following changes is opened, which is closed upon completion of the command.

### 2.3 Technical Realisation

As prove of concept we have implemented CoObRA in Java, but most imperative programming languages would be well suited.

The CoObRA framework consists of some server applications and some client libraries. As a basis for collecting change protocols, CoObRA provides the interface class `ObjectChangeAware` and the library class `ObjectChangeAwareAdapter`, cf. Figure 1. In order to enable CoObRA functionality for an application's class, the class may either implement the `ObjectChangeAware` interface or more conveniently it inherits from class `ObjectChangeAwareAdapter`. In the latter case, i.e. the constructor of class `ObjectChangeAwareAdapter` automatically adds new objects to the local CoObRA change list. In the former case, the user has to implement this himself.



**Figure 1. Classes of persistent (CoObRA enabled) object must implement a specific interface or extend an adapter class.**

In the next step, all changes to plain fields and collection fields have to be reported to CoObRA. Therefore, a helper class provides appropriate methods for firing these

changes. These methods have to be called on any change to an attribute of an CoObRA enabled object.

Fortunately, in Java it is common practice to access fields by calling access methods only, cf. Java Beans [Pra97]. If this is already the case, the calls for firing the changes may just be added to the setters of all attributes. Adding the calls to all setters may easily be achieved using modern refactoring mechanisms as provided by most modern IDEs or using an appropriate aspect weaving mechanism. However, if the application is developed with the Fujaba CASE tool, these calls are automatically added to all setters of CoObRA enabled classes by the Fujaba code generator.

Another popular concept for the notification of changes is the Observer pattern, see [GHJV95]. Java's graphical user interface libraries support this pattern by so-called property change events (see `PropertyChangeEvent` in [SunJDK]). Thus, there is some likelihood, that Java applications with a modern graphical user interface fire appropriate `PropertyChangeEvent` events, already. In this case, it is easy to add CoObRA components as listeners for these change events. This may already provide CoObRA with all required change information. This approach has successfully been used to add CoObRA functionality to the Fujaba environment itself.

If some attribute fields are not properly encapsulated by access methods, one may either add direct CoObRA calls after each attribute modification or better one may introduce encapsulation, now. The latter approach is again supported by many refactoring mechanisms of current IDEs.

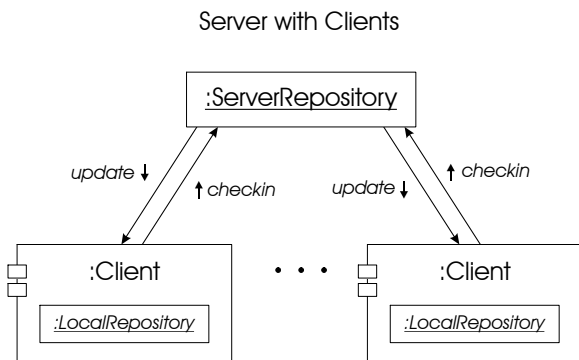
Once proper reporting of change events is achieved, the CoObRA libraries add sequence numbers to all change and store them in a local repository in chronological order (sorted by sequence number). Now the whole CoObRA functionality is available.

In Java, undo- and redo-operations have been implemented generically by using the reflection API of the Java Development Kit (see `java.lang.reflect` in [SunJDK]). To alter a field value CoObRA searches for a method with the prefix `set` followed by the name of the field. Additionally the method must have a parameter with a type matching the type of the field. For collection fields similar rules are used (i.e. prefixes `addTo`, `removeFrom`). In programming languages without reflection (or if one want's to avoid the reflection mechanism), this functionality could be achieved by generating/writing a generic setter method, that gets the name of the considered attribute as a parameter and that uses a switch-case construct to call the corresponding set-method, directly.

The current implementation for reverting an object removal actually does nothing. As Java objects are not destructed, in contrast to e.g. C++, Java applications have to remove all references to an object that should be garbage collected. The current repository implementation delays the

garbage collection of isolated objects until the history information (change log) is deleted or compacted (see above). To undo the deletion of an object it simply restores the field values (incl. references). Alternatively, one could discard the references upon delete notification. This would make garbage collection of such objects possible. In this case, a deletion would be reverted by re-creating the object and by retrieving its attribute values from the changes list.

To support the multi-user functionality, a generic server application operating on an XML representation of the changes has been implemented. In addition to the update and checkin commands the server supports retrieval of global object identifiers (unique for this server) and simple user authentication. The local repository supports substitution of the local object identifiers before checkin. This enables the client to be connected to the server only for checking in or updating as clients communicate with the server only (no peer to peer communication, figure 2).



**Figure 2. Client-server setup for CoObRA enabled applications**

The compact operation has been implemented by sorting the changes by affected object, field name and key (if qualified field). The actual compact algorithm then iterates through the re-sorted list of changes once to remove and summarize changes. Provided that a suited sorting algorithm is used this should have complexity  $O(\log(n) * n)$ . As this is the most cost intensive (regarding complexity) part of our implementation, adding entries to the change lists have complexity  $O(1)$ , we are confident that it scales pretty well.

### 2.3.1 Dealing with libraries

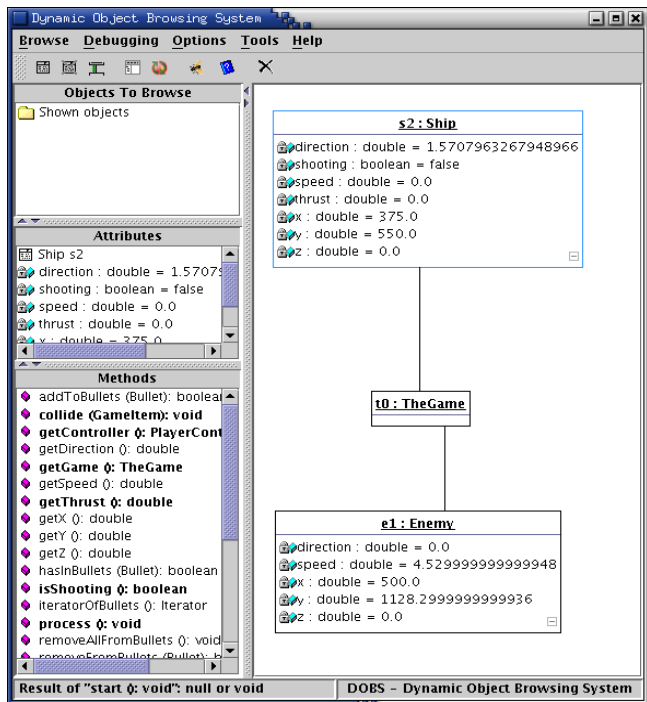
In order to store changes or to send them to the repository, they need to be serialized. For this purpose, CoObRA employs a simple XML format. This XML format is generated by our XMLReflect library. This library again uses the

Java reflection API, to access object fields and to achieve a generic encoding of attribute values.

Special problems arise, if libraries are used, that cannot be adapted to the CoObRA change notification mechanisms. For objects belonging to classes of such a library, CoObRA can only provide limited support. Changes to such objects are not recorded in our change lists and thus, such changes are not handled by our undo/redo and persistency and versioning mechanisms. CoObRA may handle such library data like plain integer or string attributes. This means, our XMLReflect mechanisms may be adapted to serialize library data. However, changes to such data needs to be signaled appropriately, as CoObRA considers these values to be immutable.

## 3 Practical Experience

Up to now, we have used CoObRA together with Fujaba in a number of student and research projects. These projects develop the behavior of the desired application using Fujaba and add the CoObRA features using Fujaba's code generators. Then, the developers validate their application with the help of our dynamic object browsing system DOBS, cf. Figure 3.



**Figure 3. The Dynamic Object Browsing System (DOBS)**

DOBS shows a cut-out of the internal object structure of a running Java Virtual Machine as an UML object diagram.

In addition, DOBS allows for adding and removing objects and links, interactively, and for modifying attribute values and for invoking methods on selected objects. One may either execute the called method step by step in a debugger and use DOBS to visualize the evolution of the application's internal object structure or one may just execute the method in a single step and DOBS will then reflect the resulting object structure.

If the object structure is CoObRA enabled, e.g. using Fujaba's code generators, then DOBS is now able to provide generic undo/redo, persistency and multi-user capabilities. One may invoke some method for validation purpose. If the resulting object structure is unexpected or erroneous, one may just press undo to go back to the situation before the method invocation and then one may execute the method call again, now in stepwise debug modus. One may store a session and continue it later. Or, one may connect to a CoObRA repository enabling versioning of the application's object data. In the latter case, one may also enable an automatic checkin and update mode. In this mode, every change is immediately transferred to the CoObRA repository and any other (DOBS) tool connected to the same repository receives a notification and performs an update operation in its local workspace. As an effect, multiple users can use DOBS in some kind of white board modus.

These CoObRA features have added a tremendous value to our validation environment DOBS and to the overall Fujaba development process (FUP). We use them intensively in almost all of our projects, now.

As an example for adding the CoObRA mechanisms to an existing (CASE) tool, we have of course added CoObRA to the Fujaba CASE tool itself. The core of the Fujaba CASE tool consists of roughly 600000 LOC in Java. Before this exercise, Fujaba had some generic persistency mechanism but no Undo/Redo and no multi user support. Luckily, the internal abstract syntax graph (or meta model) of Fujaba has already met most of the criteria for the application of the CoObRA mechanism, i.e. most fields have already been properly encapsulated by access methods. In addition, these access methods were already firing Java AWT property change events utilized by the graphical user interface of Fujaba. Thus adding CoObRA to Fujaba "just" required to subscribe for these change events.

Actually, adding CoObRA functionality to the "clean" parts of Fujaba was considerably simple. However, there were some "elder" parts of Fujaba that did not stick to the usual coding standards for attribute encapsulation that strictly. Thus, we needed some more time to clean these parts up. In addition, some access methods of Fujaba include complex side effects as e.g. the renaming of a class. These cases required special care. Finally, all features were available to Fujaba. Altogether, we were able to add the CoObRA functionality to Fujaba within roughly two person

months.

We have applied the new mechanisms and especially the versioning functionality with great success in a number of student exercises with about 60 students in 14 groups of 4 to 5 students each working for about two weeks on a group project. Within these student projects we hit some scaling problems, most of them already solved, but overall the CoObRA functionality facilitated the group collaboration considerably in contrast to last year's exercises. Astonishingly, we observed quite a number of concurrent modifications by different team members but we did not observe any severe merge conflicts or merging problems. The rare cases usually dealt with concurrent renaming of some commonly used class. Fujaba simply prints the changes that could not be replayed and then discards the data. In these cases, CoObRA's textual merge conflict reports were sufficient for our students to resolve the conflicts, easily.

We use the CoObRA mechanism now in all Fujaba projects at University of Kassel with great success.

## 4 Related Work

CoObRA utilizes generic undo/redo, persistence and concurrent multiuser support. To our best knowledge, there exists no approach that supports all features, but some of them. Therefore, in the following we present a number of approaches, tools and also commercial products for each of CoObRA's features, separately.

Persistence and multiuser support is the classical domain of software configuration management (SCM). Popular SCM systems are RCS or SourceSafe [Mic97]. Most of them base on a pessimistic locking concept, which does not allow concurrent modifications. Those systems are suitable for small development groups, only. Optimistic locking systems such as CVS are widely used in Open Source Projects, for example Apache or SourceForge projects, but CVS is still restricted to a textual representation of the documents. Versioning is not facilitated appropriately when using graph-like structures such as XMI-documents or more general abstract syntax graph representations of diagrams.

Rho and Wu [RW98] present an approach to versioning and merging software diagrams such as UML diagrams. Zündorf et al. present [Ro00, ZWR01] an approach that is also integrated in the Fujaba development environment [FNTZ98]. However, these approaches are not suited to support undo/redo.

In order to support persistence, databases are the usual solution. Java itself supports persistence by providing the Java Data Objects (JDO) API [TMMB04], which is an interface, only. For example Versant and Poet implement the JDO and therefore provide persistence based on an object-oriented database called KODO [Kodo04]. Hence the database also supports transactions, implement-

ing undo/redo means to open and close corresponding transactions manually on more or less fine-grained editing operations. The approach does not provide concurrent multiuser support and is also limited to the JDO, which makes supplementary addition of persistence to an existing application hard to achieve.

In addition to databases that handle Java Data Objects, there exist also general approaches constructing CASE-tools, such as IPSEN [Nag96] and PISET [KMP99]. Both approaches underly a specific object-oriented database, namely GRAS [KSW96] and H-PCTE [Kel92], respectively. The databases provide persistence and transactions and undo/redo functionality. But they do not provide optimistic locking concepts and merging support. All edit operations result in a change transaction directly on the database.

The idea of using incremental changes and persistence storage to provide undo/redo functionality, version control and cooperative tool support has also been presented by Emmerich [Emm96] and Grundy et al. [GHM96]. Especially Grundy et al. use the change descriptions in their meta tool, and in the tools generated by the meta tool. Our approach is not restricted to be used in a specific tool or meta tool, but provides common interfaces to support also already existing tools.

## 5 Conclusion and future work

CoObRA is a lightweight framework for adding generic undo/redo, recovery, versioning and multi-user support to object-oriented applications. Especially, CoObRA enables concurrent (off-line) work on local copies of the project data on the basis of an optimistic locking concept. Concurrent modifications by different team members are then merged using a generic mechanism.

CoObRA protocols all modifications on the application's internal data structure in so-called change lists with very low runtime overhead. These change lists are a very flexible means to support CoObRA's functionality. Obtaining these change lists is easily achieved using Fujaba's code generation mechanisms or using event notification mechanisms that may already be in place.

We have fitted the Fujaba tool with the CoObRA framework within only two person month. Our experiences with CoObRA enabled version of Fujaba are very satisfying. The new functionality adds a tremendous value to our tool and enables flexible teamwork. To our own surprise, merging of concurrent modifications works very smoothly. Integration costs for CoObRA can possibly be reduced further on by using automated byte code instrumentation for Java applications.

In our experience, we did not recognize a considerable runtime overhead for recording the change lists. Even the

consumption of main memory is very low. Storing and retrieving the data is not significantly slower in comparison to our previous persistency mechanism. Using compression techniques such as zip, the disk space consumption after compacting the change lists is comparable to the files produced by the old mechanism.

CoObRA is designed to provide versioning for object-oriented data. However, in software development projects, usually a lot of textual data exists that is conveniently managed using a conventional SCM system such as CVS. To enable configuration management for the whole data of a software development project, a sound integration of CoObRA and conventional SCM systems would be very beneficial. One idea for this problem is to put the CoObRA change lists under the control of a common (text-based) SCM system using some appropriate text format. In the case of optimistic locking this will likely cause merge conflicts on the CoObRA change lists. However, CoObRA could analyze the conflicts as flagged by the textual SCM system and retrieve both versions of the change list and then use its own merging mechanism in order to deal with the conflicts more reasonably.

As for other SCM systems, we foresee the demand for hierarchically organized repositories. Several small developer groups may want to have a personal version, an inner-group-version and a corporate-version of the software they are developing. Due to the flexibility of CoObRA's change lists, this functionality should be easy to achieve.

Another major aspect for CASE tools is scalability. CoObRA works well if the application data fits into the clients main memory. In large software projects the data may easily exceed this limit. One may circumvent this problem by loading only the currently required part of the specification while other parts are kept on disk or server only. The concept of loading only a part of a repository may be extended by loading required objects on demand. While our flexible change lists provide us with a sufficient basis for such a functionality, such a mechanism imposes new restrictions on the applications' data structures. CoObRA needs to be able to limit and control the side-effects of constructors and setters of on-demand loaded objects on already loaded parts of the application data in order to avoid corruption of these parts.

If these extensions can be achieved, CoObRA would become a light-weight alternative to object-oriented database systems. The major difference between an extended CoObRA and an OO database would be the transaction concept. CoObRA replaces usual transaction concepts with optimistic locking and merge concepts. As experiences with SCM systems show, such an optimistic locking concept is much more appropriate for software development processes than usual database transactions. In addition, CoObRA's optimistic locking concept enables a very simple caching

concept. Each user works on his own writable copy. This is a major advantage in terms of performance.

## References

- [Beck99] K. Beck: Extreme Programming Explained: Embrace Change; Addison-Wesley, ISBN 0-201-61641-6, 1999.
- [CW98] R. Conradi and B. Westfechtel: Version models for software configuration management, ACM Computing Surveys, 30(2):232-282, June 1998
- [DGZ03] I. Diethelm, L. Geiger, A. Zündorf: Systematic Story Driven Modeling; Technical Report, Universität Kassel, 2002.
- [Emm96] W. Emmerich: Tool Specification with GTSL. In Proc. of 8th International Workshop on Software Specification and Design, IEEE CS Press, Schloss Velen, Germany, 1996
- [FNTZ98] T. Fischer, J. Niere, L. Torunski and A. Zündorf: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Proc. of the 6<sup>th</sup> International Workshop on Theory and Application of Graph Transformation, Paderborn, Germany
- [GHJV95] E. Gamma and R. Helm and R. Johnson and J. Vlissides: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995
- [GHM96] J. Grundy, J. Hosking and W. Mugridge: Supporting flexible consistency management via discrete change description propagation, Software Practice and Experience, 26(9), 1996
- [JBR99] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process; Addison-Wesley, ISBN 0-201-57169-2, 1999.
- [Kel92] Udo Kelter: H-PCTE - a high performance object management system for system development environments. In Proc. of Intl. Computer Software and Application Conference, Illinois, IEEE Press, 1992
- [KMP99] Udo Kelter, Marc Monecke and Dirk Platz: Constructing distributed SDEs using an active repository. In Proc. 1st Intl. Symposium on Constructing Software Engineering Tools, Los Angeles, CA, USA, 1999
- [Kodo04] Versant GmbH: Kodo JDO: robust, highperforming, feature-rich implementation of the Java Data Objects standard. Online: <http://www.versant.com>
- [KSW96] N. Kiesel, A. Schürr and B. Westfechtel: GRAS: A Graph-Oriented Software Engineering Database System. In The IPSEN Approach, LNCS 1170, Springer Verlag, Berlin 1996
- [Mic97] Microsoft Corporation: Managing Projects with Visual SourceSafe, Redmond, Washington, 1997.
- [Nag96] Manfred Nagl, editor: Building Tightly-Integrated Software Development Environments: The IPSEN Approach. Lecture Notes in Computer Science (LNCS) 1170, Springer Verlag, Berlin, 1996
- [Pra97] S. Prashant: Java Beans developer's resource Prentice Hall, 1997
- [Ro00] Ingo Rockel: Versionierungs- und Mischkonzepte für UML Diagramme, Diplomarbeit Universität-GH Paderborn im September 2000
- [RW98] J. Rho and C. Wu: An efficient version model of software diagrams. In Proc. 5th Asia-Pacific Software Engineering Conference, Taipei, Taiwan, ROC, IEEE Computer Society, 1998
- [SunJDK] Sun Microsystems Inc.: Java™ 2 SDK, Standard Edition Documentation, Version 1.4.0, <http://java.sun.com/j2se/1.4/docs/>
- [TMMB04] Sameer Tyagi, Michael Vorburger, Keiron McCammon and Heiko Bobzin: Core Java Data Objects Prentice Hall PTR / Sun Microsystems Press, 2004
- [Whi00] B.A. White: Software Configuration Management Strategies and Rational ClearCase. Addison-Wesley, 2000
- [ZWR01] Albert Zündorf, Jörg Wadsack and Ingo Rockel: Merging graph-like object structures. In Proc. of 10th International Workshop on Software Configuration Management, Toronto, Canada, 2001