A Concept for CAD Systems with Persistent Versioned Data Models

D. G. Beer, Bauhaus University Weimar; Informatik im Bauwesen (daniel.beer@bauing.uni-weimar.de)
B. Firmenich, Bauhaus University Weimar; CAD in der Bauinformatik (berthold.firmenich@bauing.uni-weimar.de)
T. Richter, Bauhaus University Weimar; Informatik im Bauwesen (torsten.richter@bauing.uni-weimar.de)
K. Beucke, Bauhaus University Weimar; Informatik im Bauwesen (karl.beucke@bauing.uni-weimar.de)

Summary

The synchronous distributed processing of common source code in the software development process is supported by well proven methods. The planning process has similarities with the software development process. However, there are no consistent and similarly successful methods for applications in construction projects. A new approach is proposed in this contribution.

1 Introduction

Distributed processing: The distributed processing of a shared product model instance is characterized by three iterative phases (figure 1): a) the phase of loading subsets of the product model instance (Firmenich 2002), (Beer and Firmenich 2003), (Beer et al. 2003), b) the phase of the local and independent processing of this subset and c) the phase of discarding or storing the results (Richter et al. 2003). These three phases belong to a single long transaction that can take up to weeks. In this process the second phase generally governs the time demands for executing the long transaction.

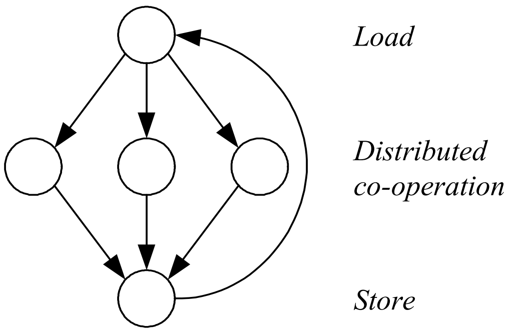


Figure 1: Distributed processing of a shared product model instance (Firmenich 2002)

Conflict management: It is proposed to provide copies of the selected product model objects to allow for synchronous parallel co-operation. If the processed copies are stored as new object versions, the history of the planning process is available and the evolution of the planning process may be traced (Firmenich 2002). Processing copies of objects ensures the consistency of the existing product model instance. This procedure also guarantees that the work of the planners can be stored without violating data integrity. It is assumed that the merging of two or more versions is executed at a later stage: This has to be done interactively by the user.

2 Solution Approach

Solution approach: The solution approach is based upon two elements (figure 2): a shared project and a private workspace for each planner. The workspace is represented by available CAD systems that have to be augmented by operations for the distributed co-operation.

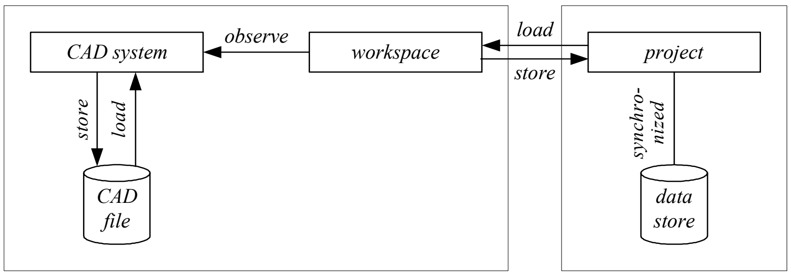


Figure 2: Solution approach

Implementation concept: The general implementation concept derived from figure 2 is shown in figure 3.

The project is represented by the class `Project`. It uses the functionality of a server, implemented by the class `Server`. The server is multi-threaded to manage multiple sessions. Thus, more than one workspace may communicate with the project at the same time. The persistency functionality comes with an own persistence framework implementation. This framework is described in detail in (Beer et al. 2004). The `project` class is executable (shown as bold letters).

The workspace is represented by the interface `Workspace`. The class `WorkspaceAdapter` is a specific implementation of this interface. It defines the operations for the distributed co-operation. The connection with the CAD system is a specific task: The class `WorkspaceCIB` implements the connection with the CAD system of the pilot implementation. This class is executable. All workspaces use the functionality of a client, given by the class `Client`.

Messages are exchanged between the clients and the server sessions. An abstract class `Msg` defines the interface. All types of messages to be sent via the Internet are represented by classes inherited from class `Msg`. Thus, the message handling is simplified.

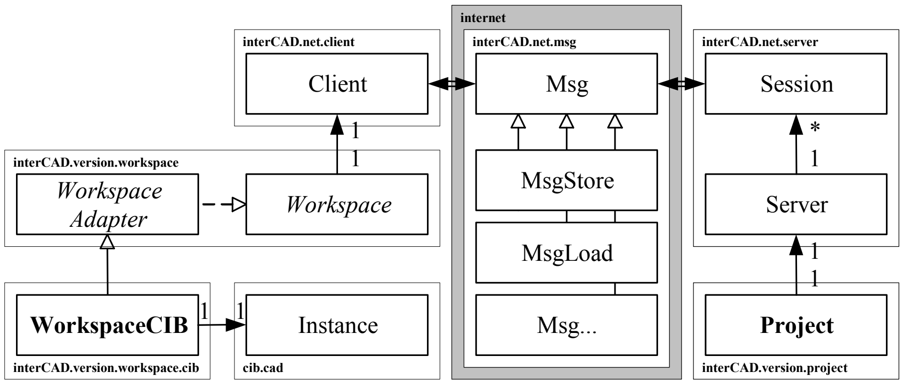


Figure 3: Implementation concept

The elements of the solution approach and the simplified class protocols are given in the following sections.

3 Project

Introduction: The project manages the persistent versioned product model instance shared between all planners. However, objects can not be edited directly: Instead, new object versions must be derived and loaded in the workspace where they can be edited. The versioned product model is described by (Firmenich 2002). A short introduction follows.

3.1 Concepts

Object versions: The object versions of the project are included in set M_P. The elements of this set may not be modified.

M_P	Set of object versions of the project.

Virtual object versions: Virtual object versions are used to show that an object version was created for the first time or that an object version was deleted (figure 4). Virtual object versions are stored in set Δ. All (virtual) object versions are stored in set M_Δ.

Δ	Set of virtual object versions.
M_Δ	Set of object versions and virtual object versions. $M_\Delta = M_P \cup \Delta$

Identification: Object versions are copied for the distributed processing of the instance of a shared product model. The identity of an object version differs from the identity of the object version's copy. This is why separate persistent identifiers are used with the project as identifier space (namespace). A unique persistent identifier is assigned to each object version.

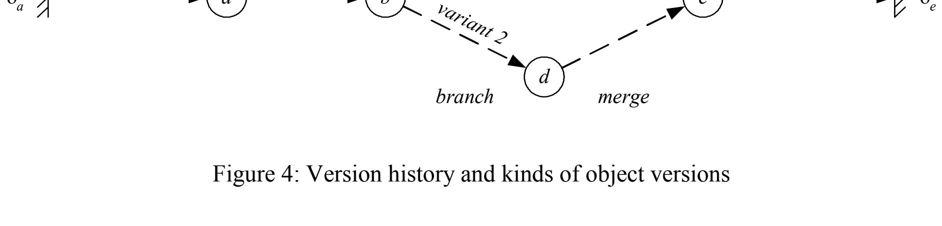
I_P	Set of persistent identifiers, created by the project.
$\boldsymbol{I}$	Identification relation. Bijective mapping $M_P \rightarrow I_P$

Dependencies between object versions are not considered in this paper in order to reduce complexity. Structured sets of object versions are described in (Firmenich 2002). The importance of such dependencies – abstracted as mathematical relations – are shown in (Pahl and Beucke 2000). An update mechanism for structured sets is given by (Hanff 2003).

Version history: All pairs of object versions – whereas the second object version is derived from the first object version – are stored in V. An example is given in figure 4.

$\boldsymbol{V}$ Version history. $\boldsymbol{V} \subseteq M_\Delta \times M_\Delta$

$(\delta_a, a) \in \boldsymbol{V}$	Initial object version a without previous version(s).
$(a, b) \in \boldsymbol{V}$	Object version a with revision b.
$(e, \delta_e) \in \boldsymbol{V}$	Deleted object version e.



first (initial) object version
revision
variant 1
variant 2
branch
merge
deleted (virtual) object version
δa
a
b
c
d
e
δe

Figure 4: Version history and kinds of object versions

3.2 Implementation concept

Project: The protocol of the `Project` class is given in figure 5. The `main` method of the project creates a new project instance. The constructor initializes the persistence framework (Beer et al. 2004), the message handling and starts the server. The methods `load` and `store` handle corresponding messages. They load copies of existing object versions from the project and store newly derived object versions to the project.

```
class Project {
    static void main(String[] args);
    Project();
    Msg load(MsgLoad msg);
    Msg store(MsgStore msg);
}
```

Figure 5: Class `Project`

Server: The `Server` class is shown in figure 6. The `Server` class defines a constructor that needs a port number and a listener that reacts on incoming messages. The constructor starts an infinite loop to wait for new clients to connect. For each client a new session is created that communicates with the client via messages.

```
class Server {
    public Server(int port, MsgListener msgListener);
}
```

Figure 6: Class `Server`

Session: The class `Session` (figure 7) is a thread that is started for each client. The constructor needs the name of the workspace, the listener that reacts on incoming messages as well as two streams for incoming and outgoing message objects. The method `run` – inherited from class `Thread` – is the main method of the thread. It waits for incoming messages and handles them. Therefore it uses the listener methods for access to the data of the project.

```
class Session extends Thread {
    public Session(String name, MsgListener msgListener,
        ObjectOutputStream out, ObjectInputStream in);
    public final void run();
}
```

Figure 7: Class `Session`

Messages: The protocol of the abstract message class `Msg` is given in figure 8. There are only methods for accessing private attributes: the request and the response of the message modeled is as `Object`. Specific messages are derived from class `Msg`.

```
abstract class Msg implements Serializable {
    void setRequest(Object request);
    Object getRequest();
    void setResponse(Object response);
    Object getResponse();
}
```

Figure 8: Class `Msg`

Example: The request of the `MsgLoad` class (figure 9) is a string that defines a subset of the versioned product model instance. The string represents a feature logic (Zeller 1997) term. The response is a map of persistent identifiers and newly created object versions.

```
class MsgLoad extends Msg implements Serializable {
    void setRequest(String request);
    String getRequest();
    void setResponse(Map response);
    Map getResponse();
}
```

Figure 9: Class `MsgLoad`

Conclusion: The versioned product model instance is the basis for the parallel co-operation.

4 Requirements on commercially available CAD Systems

Introduction: Object versions from the project are loaded as CAD components. The local data management of available CAD systems allows offline processing in the distributed planning process, e.g. on mobile devices and notebooks. The transaction ends by discarding or storing the changed object versions in the project. It is convenient to use the functionality of available CAD systems.

Requirements: A CAD system for the distributed co-operation has to offer an environment to implement a workspace. The workspace acts as an observer on the CAD components. The CAD system has to offer a suitable observer concept. Furthermore the workspace has to define new commands in the CAD system for the loading from and the storing to the project. Thus, a programming interface (API) that allows the access to the CAD components and the definition of user commands is required. All these requirements are fulfilled by most available CAD systems.

Conclusion: Although available CAD systems only support the second phase of the distributed co-operation they are an ideal software environment for the implementation of a workspace.

5 Workspace

Introduction: Objects to be edited are loaded from the project as new object versions in the planner's workspace. The workspace provides CAD functionality and additional operations for the distributed co-operation. This functionality is implemented within a specific CAD system: The CAD operations must be observed in order to support the distributed co-operation of the project.

5.1 Basics

Object versions: All object versions of the workspace are included in set *M*.

M	Set of object versions of the workspace.

Identification: Object versions are copied during the exchange with the project. The identity of an object version differs from the identity of the copy of the object version. This is why separate identifiers are used. Loaded object versions are assigned to unique persistent identifiers, newly created object versions are assigned to unique transient identifiers. The identifier space (namespace) of the transient identifiers is the workspace.

I_W Set of all (transient and persistent) identifiers of the workspace.

$\boldsymbol{I}$ Identification relation. Bijective mapping $M \to I_W$

Dependencies between object versions are not considered in this paper in order to reduce complexity. Structured sets of object versions are described by (Firmenich 2002). The importance of such dependencies – abstracted as mathematical relations – are shown in (Pahl and Beucke 2000). An update mechanism for structured sets is given by (Hanff 2003).

Object version types: There are four types of object versions that are included in disjunctive sets. Unmodified object versions need not be stored. Deleted object versions, newly created object versions and modified object versions have to be stored and uniquely distinguished in order to be able to maintain the version history (figure 10).

M_U Set of unmodified object versions after loading from project. $M_U \subseteq M$

M_D Set of deleted object versions. $M_D \subseteq M$

M_N Set of newly created object versions. $M_N \subseteq M$

M_M Set of modified object versions. $M_M \subseteq M$

CAD operations: A planner can load CAD components from a file, modify these components (e.g. translate, rotate, mirror, …) and delete these components. These operations have to be observed by the workspace in order to ensure that all object versions are included in the correct sets defined above. This is necessary to construct the version history of the project.

Version history: The version history built in the project depends on the operations executed in the workspace. Figure 10 shows all eight combinations the operations that can be executed. The first four cases are associated with object versions loaded from the project. The other four cases are associated with newly created object versions. This is equivalent to the loading from a local CAD file. Object versions can either be newly created or loaded.

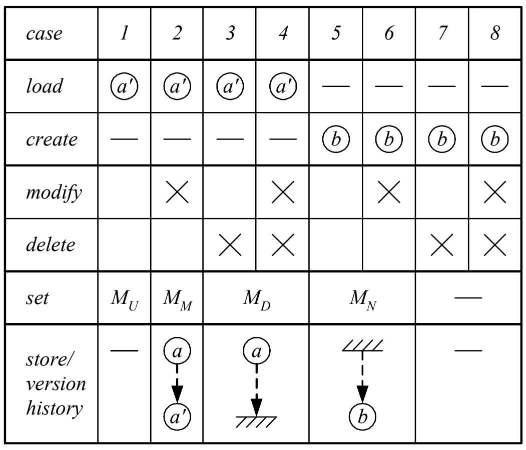


Figure 10: Operations and the consequences on storage and version history

Object versions loaded but not processed (case 1) are not stored. Object versions loaded and only modified are stored as derived object versions (case 2). Object versions loaded and finally deleted are stored as virtual object versions (case 3 and 4).

Object versions newly created and not processed (case 5) or modified (case 6) are stored as first versions of an object. Object versions newly created and finally deleted are not stored (case 7 and 8).

5.2 Implementation concept

Client: The class `Client` (figure 11) has a constructor that creates a `Socket` for the connection via the Internet. The address of the server and a port number are passed as arguments. Currently, the client is not multi-threaded: it sends messages (method `communicate`) and blocks until the server replies. Errors are handled by `NetExceptions`. In the future, asynchronous communication will be supported.

```
class Client {
    Client (String server, int port) throws NetException;
    Msg communicate (Msg msg) throws NetException;
}
```

Figure 11: Class `Client`

Workspace: The main methods of the workspace are defined in the interface `Workspace` (figure 12). The workspace uses a client for the communication with the project. The CAD components are observed by an observable set. All other methods shown in figure 12 implement the operations for the distributed co-operation. The workspace is implemented by an adapter.

```
interface Workspace {
    void setObservableSet(ObservableSet s);
    Set load() throws WorkspaceException;
    int store() throws WorkspaceException;
    public void shell();
    public void addLink();
    public void removeLink();
    public void update();
    public void unload();
    public void merge();
    public void modernize();
    public void reduce();
    ...
}
```

Figure 12: Interface `Workspace`

Workspace adapter: The `WorkspaceAdapter` class (figure 13) implements the `Workspace`. It defines the sets and relations mentioned above as private attributes. The set *M* of all object versions is observable. The constructor needs the address of the server and the port number in order to instantiate the workspace's client. The observable set has methods to react on CAD operations (create/ load, modify, delete). Thus, the workspace adapter can react adequately (figure 10).

```
class WorkspaceAdapter implements Workspace {
    private ObservableSet M;
    private Set M_M, M_U, M_N, M_D;
    private Map V, B;

    WorkspaceAdapter(String server, int port)
        throws WorkspaceException;
    final void setObservableSet(ObservableSet s);
}
```

Figure 13: Class `WorkspaceAdapter`

Specific workspace: A specific workspace implementation (figure 14) is extended from the workspace adapter. The constructor needs the parameters for the server to call the constructor of the super class `Workspace`. Furthermore, the constructor defines commands for the distributed co-operation and registers them with the CAD system. The graphical user interface (GUI) of the CAD system is extended and the observable set *M* is connected with the CAD's data model.

The method `shell` starts the CAD shell and the graphical user interface. The workspace itself is started by the `main` method that defines the server parameters, starts the server and calls the `shell` method.

```
class WorkspaceCIB extends WorkspaceAdapter {
    WorkspaceCIB(String server, int port)
        throws WorkspaceException;
    void shell();
    static void main(String[] args);
}
```

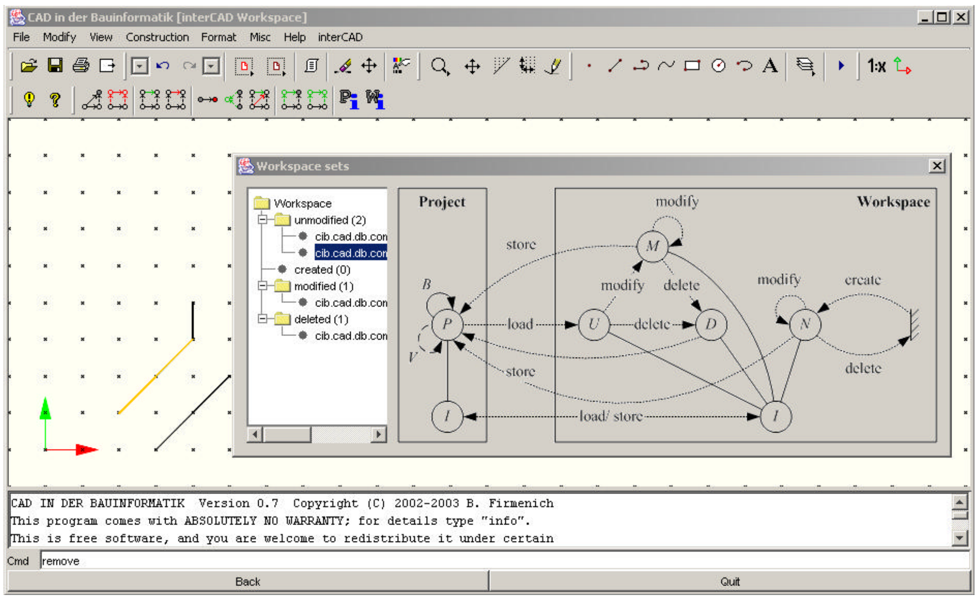
Figure 14: Class `WorkspaceCIB`

Conclusion: The implementation effort for a specific CAD system has been proven to be relatively low.

6 Pilot implementation

Project: The project is a pure command line Java™ application with multi-threaded server functionality. Important tasks are the access on persistent data and the communication with the workspaces.

CAD system: The CAD system used for the pilot implementation 'interCAD' was developed in Java™ at the professorship 'CAD in der Bauinformatik' of the Bauhaus University Weimar for use in teaching and research (Firmenich and Beucke 2004). The source code is available as open source. The CAD system supports user commands, the observation of the CAD model and the extension of the graphical user interface. Thus, all requirements defined above are satisfied for use as a workspace.

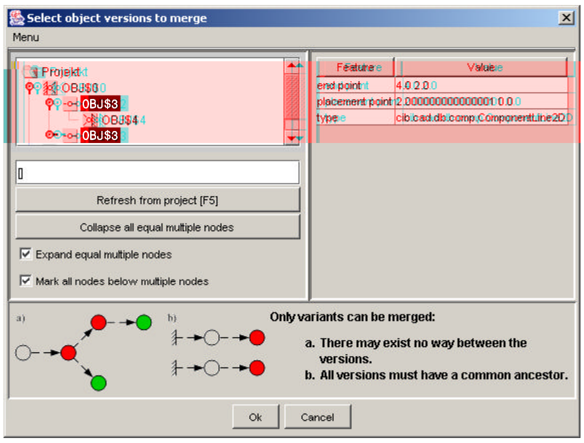


```
CAD IN DER BAUINFORMATIK  Version 0.7  Copyright (C) 2002-2003 B. Firmenich
This program comes with ABSOLUTELY NO WARRANTY; for details type "info".
This is free software, and you are welcome to redistribute it under certain
Cmd remove
```

Figure 15: CAD system and workspace

Workspace: For the implementation of the specific workspace with Java™ no adaptations of the CAD system were required. Only new commands for loading and storing were defined as well as the observation of CAD components via Java™ listeners and an extension of the CAD's user interface. Figure 15 shows the pilot implementation of the workspace with the graphical user interface of the CAD system. The sets of the workspace are represented by a tree.

Operations: The operations for the distributed co-operation are available as Java™ classes (Firmenich 2002). They were implemented using Feature Logic (Zeller 1997). The workspace provides corresponding user dialogs (figure 16).


Select object versions to merge
Menu
Projekt
OBJ$0
OBJ$3
OBJ$4
OBJ$3
Refresh from project [F5]
Collapse all equal multiple nodes
Expand equal multiple nodes
Mark all nodes below multiple nodes
a)
b)
Ok
Cancel

Feature	Value
end point	4.0 2.0
placement point	2.0000000000000001 0.0
type	cib.cad.db.comp.ComponentLine2D

Only variants can be merged:

a. There may exist no way between the versions.
b. All versions must have a common ancestor.

Figure 16: Operations for the distributed co-operation (example: *merge*)

Conclusion: The solution approach can be proven to support all three phases of the distributed co-operation and the functionality of available CAD systems.

7 Outlook

Scenarios: Relevant AEC examples have to be described and checked to test the practical applicability. The concept and the pilot implementation have to be verified with these examples.

Performance: The performance of the access on the versioned project data has to be measured and improved. Minor tests with loading/ storing of unstructured sets from/ into the project show the performance (figure 17). The complexity is polynomial (linear for storing and quadratic for loading). Tests for structured sets (with relations between object versions) have been simulated, too. They show similar results.

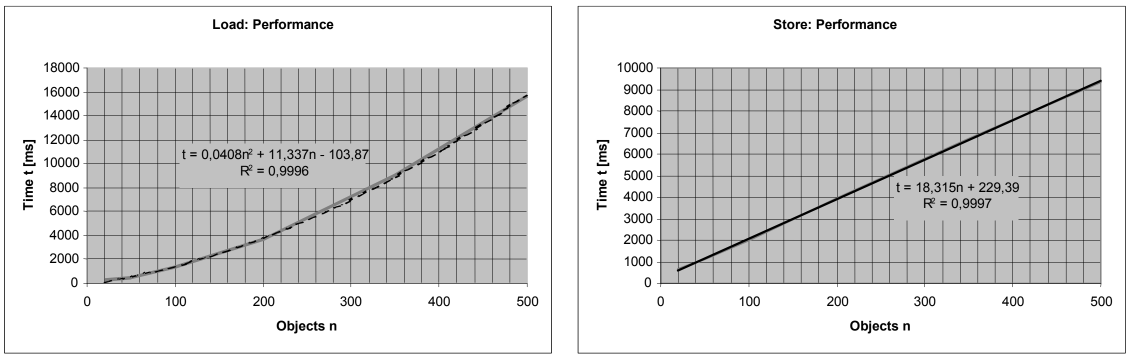


Figure 17: Performance of project access via Internet

Workspace: A further workspace implementation for a commercial CAD system will follow the pilot implementation. The common part will be verified. New requirements will arise by the use of different CAD systems: Objects of different applications have to be treated uniformly. This fact proves the generalizability of the developed concepts for available CAD systems.

Intention: The final result will be a pilot implementation for the distributed processing of real AEC projects with available CAD systems.

8 Acknowledgement

The authors gratefully acknowledge the financial support of this project by the German Research Foundation (Deutsche Forschungsgemeinschaft – DFG) within the scope of the priority program 'Network-based Co-operative Planning Processes in Structural Engineering' (SPP 1103).

9 References

Beer, D. G., and B. Firmenich (2003). Freigabestände von strukturierten Objektversionsmengen in Bauprojekten. In Digital Proceedings des Internationalen Kolloquiums über Anwendungen der Informatik und Mathematik in Architektur und Bauwesen (IKM). <http://euklid.bauing.uni-weimar.de/papers.php?lang=de&what=20> (13 April 2004).

Beer, D. G., B. Firmenich, and K. Beucke (2003). Motivation für eine Sprache zur Handhabung strukturierter Objekversionsmengen. In Tagungsband zum 15. Forum Bauinformatik. Aachen: Shaker.

Beer, D. G., B. Firmenich, T. Richter, and K. Beucke (2004). A Persistence Interface for Versioned Object Models. In Proceedings of the Vth European Conference of Product and Process Modeling (ECPPM). Istanbul: Istanbul Technical University. – Planned paper.

Firmenich, B. (2002). CAD im Bauplanungsprozess: Verteilte Bearbeitung einer strukturierten Menge von Objektversionen. < http://www.shaker.de/Online-Gesamtkatalog/Details.idc?ISBN=3-8265-9924-1> (13 April 2004).

Firmenich, B., and K. Beucke (2004): CAD in Computer Aided Civil Engineering: a Particular Approach for Research and Education. In: Proceedings of the Vth European Conference on Product and Process Modelling (ECPPM). Istanbul: Istanbul Technical University. – Planned paper.

Hanff, J. (2003). Abhängigkeiten zwischen Objekten in ingenieurwissenschaftlichen Anwendungen. <http://www.shaker.de/Online-Gesamtkatalog/Details.idc?ISBN=3-8322-2280-4> (13 April 2004).

Pahl, P.J., and K. Beucke (2000). Neuere Konzepte des CAD im Bauwesen: Stand und Entwicklungen. In Digital Proceedings des Internationalen Kolloquiums über Anwendungen der Informatik und Mathematik in Architektur und Bauwesen (IKM). <http://euklid.bauing.uni-weimar.de/ikm2000/docs/089/089d.html> (13 April 2004).

Richter, T., B. Firmenich, and K. Beucke (2003). Ein Java-Paket zur Verarbeitung von Datenstrukturen in beliebigen Datenquellen. In Tagungsband zum 15. Forum Bauinformatik. Aachen: Shaker.

Zeller, A. (1997). Configuration Management with Version Sets – A Unified Software Versioning Model and its Applications. Braunschweig: Fachbereich Mathematik und Informatik.