Supporting State-based Transactions in Collaborative Product Modelling Environments

M. Weise & P. Katranuschkov
Institute for Construction Informatics, Technical University of Dresden, Germany

ABSTRACT: The up to date state and the consistency of shared building model data are of utmost importance for the achievement of efficient model-based collaborative work. However, in engineering design these are not easy tasks. Design activities are typically carried out in *long transactions* that are characterized by the following three subtasks: *check-out* of the needed design data into a private workspace, *making design changes* within the private workspace, and *check-in of a new model state* into the shared model repository to make changes and decisions visible to the other designers. As a result of various existing semantic interoperability problems, in the new model state both actual design changes and data loss have to be considered that cannot be easily distinguished. To help tackle these problems we suggest a *delta-based versioning approach* whose essence is in storing design changes instead of complete design states. This approach is then used as basis to support the three data processing stages of a design step within a collaborative work environment, namely (1) creation of the needed and manageable model subset by removing all irrelevant design data, (2) storing the design changes, and (3) restoring the removed data by an "undo" operation of the first step. In the paper we present the used semantics for describing design changes, their transformation to deltas and the scope and limits of the suggested *undo* operation. At the end we provide an example of the use of the suggested approach with the industry standard IFC model and discuss its potential and needed further research.

1 PROBLEM DEFINITION

Model-based collaborative work is a widely known, well-defined area, tightly associated with coordination and cooperation in design teamwork. Amongst the most challenging problems within this area is the *consistency of shared model data*. It can be subdivided into two inter-related tasks: (1) ensuring interoperability to enable loss-free data exchange, and (2) efficient data management to control parallel data changes, while warranting consistent design states.

1.1 *Interoperability problems in collaborative work*

Interoperability can be treated on several levels. The relevant aspects here are *systemic* and *semantic* interoperability (Katranuschkov 2001). The first focuses on the technical process of accessing/exchanging the data, whereas the second focuses on the meaning of the exchanged data, i.e. how to understand and use the data in the context of collaborative work.

Solutions for systemic interoperability are mainly seen in providing syntactically standardised low level access to the data using an API like SDAI, or protocols like SOAP or CORBA. This allows to replace file based data exchange, which is commonly seen as a bottleneck for efficient cooperation, but does not solve problems related to semantic "misunderstandings" of the data, and does not provide answers how changes done in parallel can be managed.

Semantic interoperability, on the other side, addresses the definition of the used data. It is dealing with product modelling as well as with methods enabling the mapping of data between different product model schemas. Today, it is widely accepted that both techniques are needed in data exchange, but their appropriate combination is still in discussion. The basis of product modelling is commonly provided by a unified *meta model* which is used to formalise domain knowledge. Additionally, a *mapping language* is used to define interdependencies between different model schemas, thereby allowing to combine the knowledge of the *used product model instances* (see Figure 1).

In this context a product model instance is defined as a set of object instances related to a specific product data model which themselves comprise a set of attribute values. Typically, a product model instance will be a subset of the shared building information model (BIM).

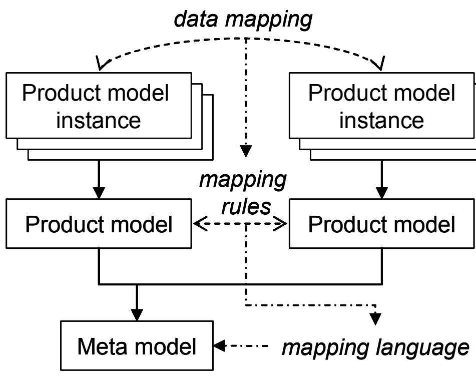

Figure 1. High level concept of semantic interoperability.

1.2 *Data management problems*

Appropriate management of the shared model data is necessary to provide up to date design information and to ensure consistency.

From the viewpoint of database technology, a design activity in collaborative design is typically carried out as a *long transaction* which is characterised by a sequence of three subtasks: (1) *check-out* of the needed design data into a private workspace, (2) making design changes within the private workspace, and (3) *check-in* of a new model state into the shared workspace to make the data changes and design decisions visible to the other designers. However, due to time constraints such design activities are typically carried out in parallel. Consequently, there is a need to synchronize data access. This can either be achieved by restrictive, counter-productive data locks, or else, methods have to be developed to merge the diverging design data at certain coordination time points. In the latter case, beside controlling data access the most challenging task is to regain the consistency of the model data at the coordination points. For the solution of such problems various knowledge-based approaches have been suggested that are capable to partially evaluate consistency or to support certain design decisions. However, in spite of all efforts, these problems are still open.

1.3 *Observations from design practice*

In the last years advanced model-based work started to penetrate design practice. However, even though a number of the above discussed concepts have since been adopted, there are still many short-comings that handicap the realisation of the outlined data sharing approach for collaborative work.

For integration and interoperability issues the ISO 10303 standard (STEP) has been widely acknowledged. Its basic methodology specified in the parts ISO 10303-11 to 21 is being used to define specific product model standards such as CIS/2, IFC, OKSTRA and STEP AP 225. However, these standards are developed with little harmonisation with regard to each other. Their use is currently limited to neutral data exchange between design applications within pre-defined use scenarios. Furthermore, the quality of semantic interoperability heavily depends on the used applications, i.e. their import/export functionality allowing to interact with the shared model data. Consequently, the roundtrip of design data has to deal both with data loss and altered object structures.

Due to many recognised short-comings of file-based data exchange, shared product data environments are starting to be introduced. However, since there is no commonly accepted API to data management environments, design applications are still limited to file-based data exchange. Fine-grained data access as suggested by the SABLE project (Houboux et al. 2005) is constrained by technical aspects such as network traffic and, more importantly, the requirement from design practice to allow off-line modifications. Thus, the concept of application scenarios, i.e. defining model subsets for well defined business cases, is currently the most detailed data access level for practical use. Concurrency control of design changes is mainly realised by simple locking mechanisms or "first come – first served" strategies which consequently reduces the flexibility of the design process. Finally, the problem of data consistency is currently limited to check rather simple constraints defined by the underlying data structure which cannot guarantee the semantic integrity of design changes.

It can be concluded that, even though on theoretical level the concepts for interoperability and data management comprising model definitions, mapping rules and consistency checking seem to be clear and reasonable, they are still not achieved in practice. The reasons for that are multifarious and thoroughly discussed in a number of papers (cf. Turk 2001, Amor & Faraj 2001, Bazjanac 2002, Weise et al. 2004). Thus, in practice we have to deal with only partially integrated data, and this does not seem to be only a temporary handicap. Methods to overcome such practical short-comings are no less important than good product models or sophisticated database and communication tools.

2 SUGGESTED APPROACH

The baseline of our approach is that collaboration must be traceable for the involved designers and that *design changes* are among the most important data in the iterative design process. The essence is that, instead of continuously tracing all design changes (which could only be done on application level and is therefore hardly realistic), we consider only the new model states which include *all* changes done by the designer within a full design step. To do that, we suggest a set of methods allowing to shift change analysis from the actual new data states to the *data changes* (deltas) that have caused these new data states. As additional benefit, the suggested methods provide also a basis to reduce data loss in practice.

2.1 *Capturing design steps*

From data management viewpoint a design step can be characterized by (1) the used design data, i.e. the data needed to carry out the design step, and (2) the design changes, i.e. the result of the design step. This reduces the design step to its input and output, the least common denominator for supporting design applications as black-box systems. Figure 2 below shows an example design step of designer D_A who is using three objects for his design changes. Instead of storing a new design state, the changes are stored by using a *minimal change vocabulary*. Hence, we propose version management of design data by using deltas.

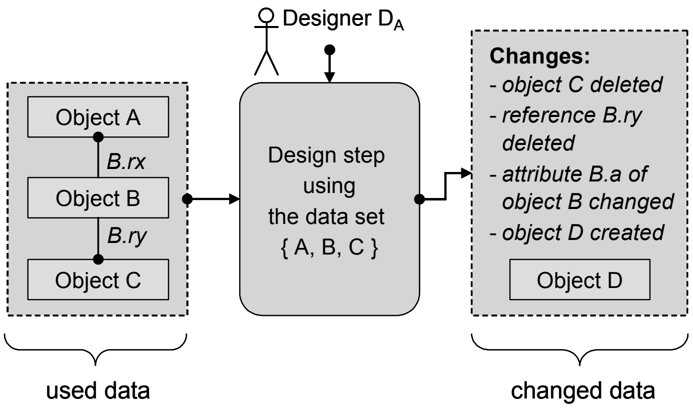

Figure 2. Example of a design step characterized by the used and the changed design data.

Compared to other application areas of version management, such as software and mechanical engineering, we see important differences for design processes in building construction, which hinders the use of available solutions like CVS or Subversion. Additionally, there is lack of methods enabling efficient version management. Therefore, before detailing the suggested delta approach, we discuss requirements for representing design changes and their integration into the overall design process.

2.2 *Requirements to represent design changes*

In order to support different phases of the design process we have to deal with significant changes of the shared product model instance. Such changes are caused both by the nature of design, i.e. the progress from sketch to detailing, and supporting IT processes such as mapping, matching and merging. Thus, the model data cannot be treated as a static object structure, changed only by the values of attributes. We have to deal with a kind of *object evolution*, where objects are sometimes split into other objects, sometimes unified to a single object, or changed to instances of another object type.

To tackle such changes in the object structure we need to extend the change vocabulary. Conceptually, we are dealing with the following types of change information: *basic changes* (creation, deletion, changing of objects and attributes), and *complex structural changes* (splitting, unification and type evolution of objects). Figure 3 shows schematically these types of change information. Additionally, we have to consider design changes carried out in parallel, thereby creating alternative design states, as well as merging of alternatives to regain a unified design solution.

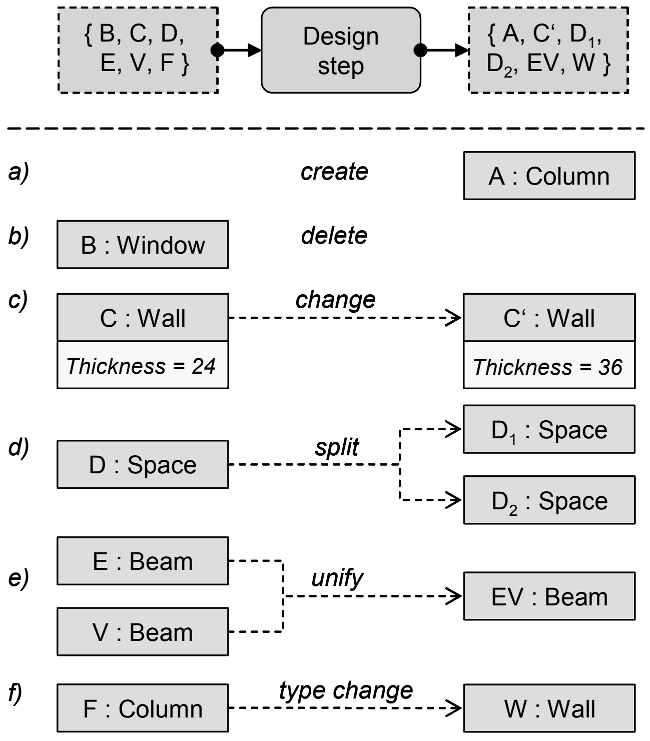

Figure 3. Information concepts needed to represent the data changes in a design step.

2.3 *Integration of change information into the design process*

In our approach, instead of continuous tracking of the changed design data, we only deal with discrete new design states. This requires additional services to identify the data changes. Moreover, a design step is carried out using a *subset* of the shared product model instance which is constrained by the capabilities of the used design application(s) to correctly import and interpret the provided data. This state-based way of working with model subsets results in additional risk for data loss which has to be reduced by the data management approach.

To be able to differentiate data changes from data loss, we divide a design step into three data processing stages: (1) selection of the needed data subset, (2) modification of the data, and (3) re-integration of the changed data into the full shared model instance. Each of these stages will be represented by data changes that can then be evaluated by the other designers, thereby providing them with as much information as possible to recognise the *intended* changes of the design step. Consequently, the selection of a model subset will be provided by removing all irrelevant design data so that a new design state is created that contains only the requested data. However, with this approach to the creation of a model subset we have to restore the removed data at the final stage. This is done by using a specific '*undo*' operation. Thus, we differentiate between changes applied to the model subset and additional 'adjustments' needed to update the shared product model instance.

These three generalised stages of a design step are illustrated on Figure 4, together with the creation of new sets of model changes at each stage.

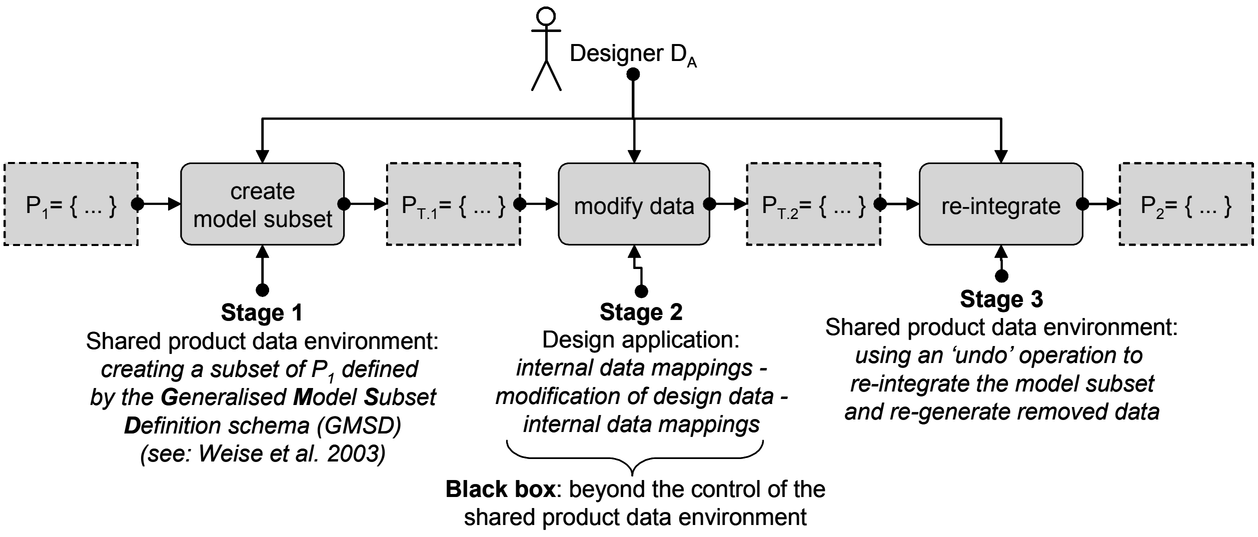
Designer D_A
P_1= { ... }
create model subset
$P_{T.1}$= { ... }
modify data
$P_{T.2}$= { ... }
re-integrate
P_2= { ... }
Stage 1
Shared product data environment:
creating a subset of P_1 defined by the Generalised Model Subset Definition schema (GMSD) (see: Weise et al. 2003)
Stage 2
Design application:
internal data mappings - modification of design data - internal data mappings
Black box: beyond the control of the shared product data environment
Stage 3
Shared product data environment:
using an 'undo' operation to re-integrate the model subset and re-generate removed data

Figure 4. Break down of a design step into three stages with distinct types of data changes at each stage.

To support the described three stages we have developed generic methods for defining and creating model subsets (Weise et al. 2003) as well as for identifying data changes (Weise et al. 2004). Based on these services we are capable to support state-based transactions using a subset of the shared product model instance.

For the principal design step shown on Figure 4 the following high-level operations can be formally defined: (a) create model subset $P_{T.1} \subseteq P_1$, (b) compare $P_{T.1}$ and $P_{T.2}$, leading to the eq. $P_{T.1} + \Delta P = P_{T.2}$ that is solved by our comparison algorithm, and (c) undo (re-integrate), applying identified changes to the design state P_1 so that the updated product model instance P_2 can be derived by $P_1 + \Delta P = P_2$.

However, in the realisation of these operations there are several practical problems that need to be dealt with. In the comparison of design states, due to the problem of missing object identifiers e.g. in IFC, we cannot ensure that the changed design state is described only by *true* changes, i.e. the changes made by the user are not always equal to the detected changes ($\Delta P_{user} \neq \Delta P_{compare}$). Another problem with IFC is that unique identifier are by definition invariant with regard to the object state which is in conflict with the change types d) and e) on Figure 3. Also, by using the suggested *undo* operation incorrect change results can lead to inconsistencies of the updated product model instance P_2. Moreover, the data changes in a new design state may not correspond to the semantic changes intended by the user. For example, if a design application is changing a globally used length unit from *m* to *mm*, it means no semantic change but results in a problem to update the shared product model instance. Consequently, the *undo* operation creating an updated shared product model instance has to be supervised by the designers to correct eventual inconsistencies.

3 VERSION MODEL

In order to capture data changes correctly, we have to deal with *change-based versioning* which has to satisfy the requirements of the defined change vocabulary. However, whilst there are many version management methods existing to date, they are all developed and used in more tightly integrated domains and cannot be readily adopted in an ICT environment for building design due to several critical differences in technical and organisational aspects.

Westfechtel and Conradi (1998) compare software configuration management with engineering data management and outline basic differences and similarities. They identify as a major difference the complexity of engineering data managed in product data models instead of text files. They mention also the problem of integrating different engineering design tools but anticipate standardized data representations solving the problem of semantic interoperability. However, even if a common shared model is agreed, as e.g. IFC, design tools will still be using their own dedicated data models that will not (and cannot) be fully harmonized with the shared model. Shifting the interoperability problems to their responsibility is neither practical nor realistic.

3.1 *Objectives of the version management*

In our version model we are dealing with well defined configurations of data objects, each describing a product model instance created in the design process. Beside the known advantages of version management for collaborative work, a specific aspect of the suggested version model is to compensate data loss caused by the existing interoperability problem. The objectives are:

- to enable error-free and consistent design steps, including the 3 subtasks: (1) *check-out*, (2) *local data changes* via design tools treated as black boxes from the viewpoint of data management, and (3) *check-in / re-integration* of the data into the common shared model instance;
- to inform the design team about identified changes;
- to provide access to earlier model versions thereby facilitating the management of conflicts via collaborative decisions.

We are not dealing with problems such as configuration management or dynamic composition of object versions to create new design solutions. Since design solutions are always created within the outlined design steps, such issues are not of interest.

3.2 *From objects to object versions*

Each design state can be handled as a product model instance defined by a set of objects, each consisting of a set of attributes. This abstraction provides the basis for the concept of the suggested version model.

$$A := \left\{ a \mid a \text{ is an attribute, representing any data value or object reference} \right\}$$

$$O := \left\{ o \subset A \mid o \text{ is an object defined by a set of attributes} \right\}$$

$$P := \left\{ p \subseteq O \mid p \text{ is a product model instance defined by a set of objects} \right\}$$

If a new product model instance must be derived from an existing product model instance, a new set of objects has to be created. This new object set is defined so that changed objects are replaced by the new object versions. To identify the changed objects we use a version relationship between new and replaced object versions. However, in contrast to other approaches we do not differentiate between objects and object versions. Thus, we define the version relationship as follows:

$$R_{VN} := \left\{ (x,y) \in O \times O \right\}$$

$$R_{VN}(x,y) - \text{ version relationship where } x \text{ is replaced by } y$$

Consequently, we are able to differentiate between the outlined change types by using basic set relational operations as shown on Figure 5. Furthermore, combinations of these change types are possible to represent more complex changes. For example, an object version can be changed by unification and type evolution at the same time.

Let $p, q \in P$ where p is a direct predecessor of q. Then:

created objects: $x :\Leftrightarrow \underset{x \in (q-p)}{\forall} \; \neg \underset{y \in p}{\exists} R_{VN}(y,x)$

changed objects: $x :\Leftrightarrow \underset{x \in (q-p)}{\forall} \; \underset{y \in p}{\exists} R_{VN}(y,x)$

deleted objects: $x :\Leftrightarrow \underset{x \in (p-q)}{\forall} \; \neg \underset{y \in q}{\exists} R_{VN}(x,y)$

split objects: $x :\Leftrightarrow \underset{x \in (p-q)}{\forall} \; \underset{y,z \in q, y \neq z}{\exists} R_{VN}(x,y) \wedge R_{VN}(x,z)$

unified objects: $x :\Leftrightarrow \underset{x \in (q-p)}{\forall} \; \underset{y,z \in p, y \neq z}{\exists} R_{VN}(y,x) \wedge R_{VN}(z,x)$

By analysing the object type it is also possible to identify type changes. If the object type is defined as shown below, all data changes caused by type change can be determined.

Let:

$K := \left\{ k \mid k \text{ is a type definition} \right\}$

$R_{OK} := \left\{ (o,k) \in O \times K \right\}$

$R_{OK}(o,k)$ be an 'instance - type' relationship

Then type change of objects is defined by:

$$x :\Leftrightarrow \underset{x \in q, y \in p}{\forall} \; \underset{a,b \in K, a \neq b}{\exists} R_{OK}(x,a) \wedge R_{OK}(y,b)$$

Figure 5. Representation of the different change types by means of set relational operations.

3.3 *From object versions to deltas*

Generally, replacement of objects is needed if (1) objects were changed by at least one attribute or split, unified or changed in type, or (2) a change is forced by consistency constraints of the version model to ensure integrity of the object states. In all other cases no replacement of objects is required.

As outlined before an object is treated as a set of attributes defining an object state. However, since changed objects are connected to replaced objects via version relationships, they can be represented only by the changed attributes. Whenever unchanged attributes are needed, they have to be determined from the replaced objects by traversal of the object history. The goal of this approach is to manage as few as possible changed objects and attributes enforced by integrity constraints of the underlying version model. Consequently, we try to omit an update of references unless it is not possible to unambiguously resolve referenced objects. The rationale is to avoid propagation of object updates to a huge number of unchanged objects. This *proliferation problem* is illustrated on Figure 6 which shows how the creation of artificially changed objects would be enforced if updates of references are performed in the version model.

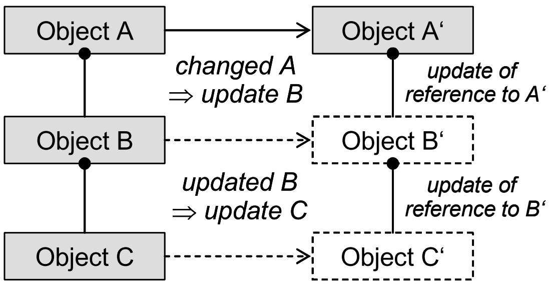

Figure 6. Proliferation of reference updates, as required by the version model.

To ensure consistency of the identified deltas, certain integrity constraints of the underlying version model have to be checked. If a version relationship between objects represents a one-to-one connection, integrity is not violated. Otherwise, the integrity of attribute values and object references that were additionally updated to avoid inconsistencies needs to be verified. Two examples of such cases are shown on Figure 7, where an update of references is only needed for the split Object A.

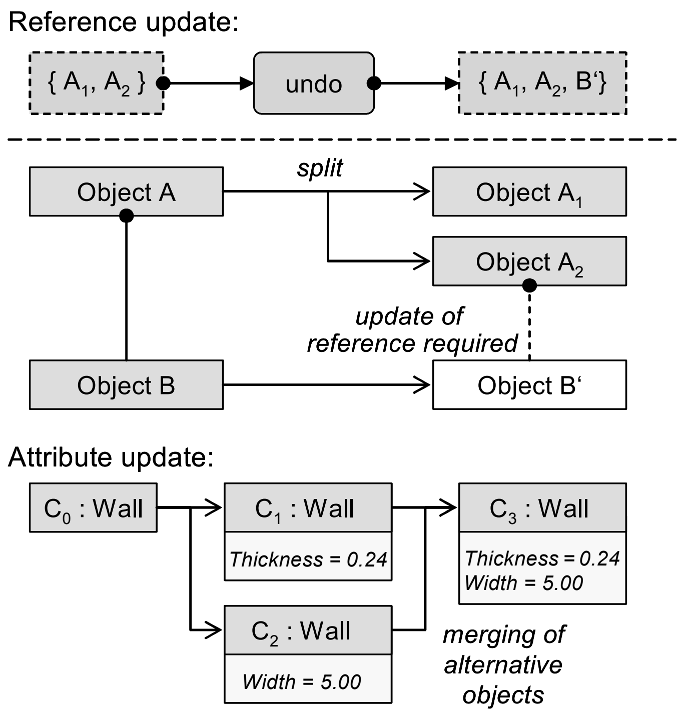

Figure 7. Update of object references and attribute values required by integrity constraints.

As we subsume that integrity of references has to be guaranteed for the changed model set by the used design applications, an update of references needs to be checked only for the *undo* operation. Such a check fails, if the design changes identified by the comparison algorithm are not compatible to the remaining product model data. Hence, the integrity constraints for references are directly related to required design changes.

In contrast to reference updates, adding of attributes is caused by unified (or merged) objects. Since changed attributes are stored in the object, we have to ensure that missing attributes can be calculated independently from the used version path. Thus, delta attributes contained in a unified object are not directly linked to changes, so that they have to be derived by traversing the whole relevant object branch from the beginning. For example, to determine the design changes between objects C_3 and C_1 on Figure 7, all deltas between C_1 and the object branch beginning at C_0 have to be compared with the deltas from C_3. From the deltas stored in C_3 we can then determine the change to the *Width* attribute.

Because of these consistency constraints, the deltas managed by the underlying version model are in general a superset of the data changes. However, the difference between deltas and actual changes is significantly reduced by the suggested structure of the version model. The changes that are an important information for the users can be easily derived from the deltas stored in the object history.

3.4 *Undo operation on deltas*

An attribute value replaced at the stage of creating the used model subset is set to the string *'replaced'* or, in the case of an attribute defining a set of values, to a *subset* of the replaced information. Figure 8 shows the replacement of the attribute *Width* enforcing the creation of a new object version C_1. The idea of the suggested *undo* operation is to invert this process by replacing all attribute values used to define Stage 1 by their former values. Thus, in the updated object version C_3 the value of the *Width* attribute of C_1/C_2 will be replaced by the value of the *Width* attribute of C_0. As long as these *'replaced'* attributes are not changed in the modifications within Stage 2, they can be automatically replaced for all changed and unchanged objects. However, if replaced attributes are defining references to other objects, their integrity is also checked.

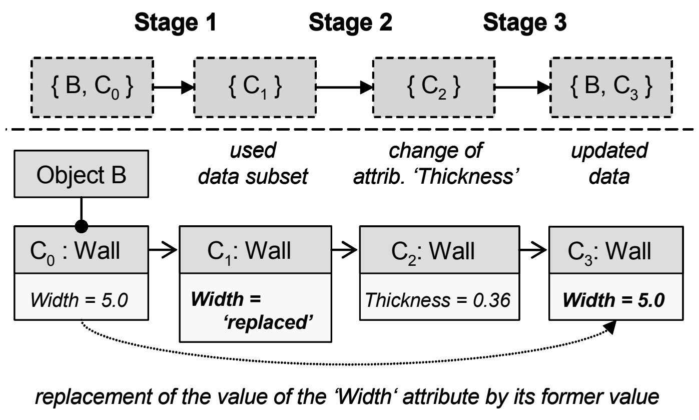

Figure 8. Re-creation of attributes by inverting their replacement.

The correct outcome of Stage 3 strongly depends on the correct identification of the changes done by the user. However, the suggested *undo* operation can warrant only the identification of low-level data conflicts caused by changes in the data structure, whereas data conflicts caused deliberately by the user or resulting from some external operation such as the mapping to/from an external model schema may or may not be correctly recognised. Therefore we subsume that Stage 3 will always be performed interactively. The user has to be aware of his responsibility to consistently update the shared product model instance. The developed delta approach supports the process but cannot perform it fully automatically.

4 EXAMPLE FROM IFC

By the time of this writing the suggested approach has been specifically tested for scenarios using the IFC Project Model (Wix & Liebich 2001). The goal of IFC is to integrate data of different domains and therefore it *has to* deal with model subsets. Such subsets are officially defined by the IAI to support different data exchange scenarios such as the coordination of building design, the transition from architectural to structural design etc (IAI-ISG 2003).

We have already tested several sub-cases of the mentioned scenarios with quite satisfactory results. To illustrate the developed approach, in this section we present a simple data roundtrip example for one building storey, downsized in accordance with the page limitations of the paper.

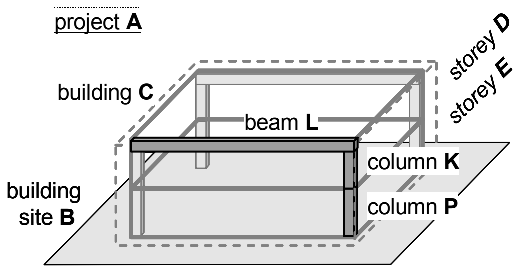

Figure 9. Example building structure.

Figure 9 presents the example building structure and Figure 10 shows a part of the data structures and the respective modifications in the three stages of the example design step. The depicted IFC elements are named and indicated by darker colour on Figure 9.

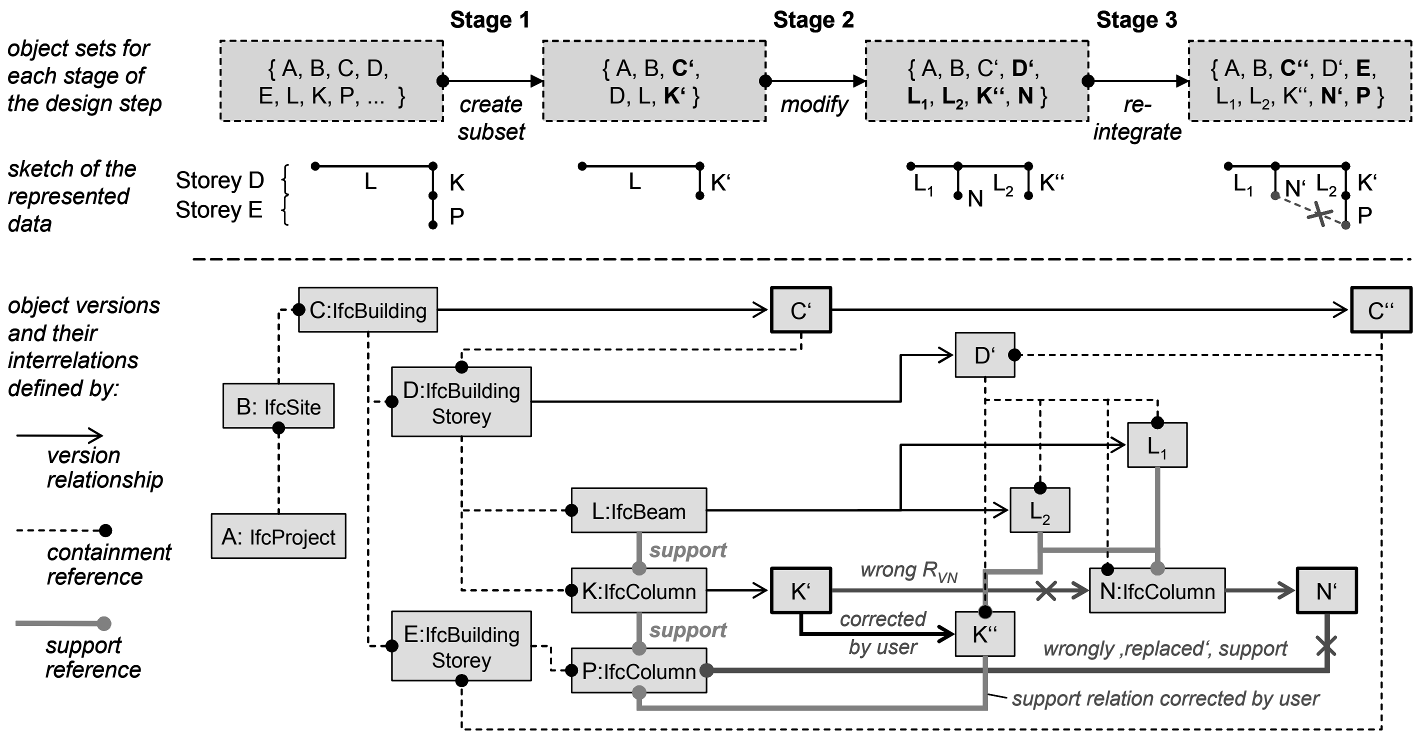

Figure 10. Schematic presentation of part of the IFC data structure, the changes in objects and relations and the respective delta-based object versions for the example from Figure 9.

Using a single storey of an IFC instance means to remove all other storeys from the existing project hierarchy, defined by spatial containers comprised of instances of the object types *IfcProject*, *IfcSite*, and *IfcBuilding*. Design coordination requires to handle various 'contained' element types such as floors, columns, walls, opening etc, whereas other elements like e.g. furniture or plumbing may not be needed. Therefore, in Stage 1 not all objects referenced from the *IfcBuildingStorey* instance D will be included in the partial model subset. To reflect these changes in the data structure, new delta versions for C and K are created, namely C' and K'. However, in accordance with the suggested approach, no new versions for A, B, D and L are needed.

The next Stage 2 of the shown scenario comprises the modifications done by the designer, which includes the creation of a new column instance N and the 'splitting' of the beam instance L into two new instances L_1 and L_2, and the identification of these changes by the data management system. In this particular example, L_1 and L_2 will be correctly recognised by the comparison algorithm whereas N may be wrongly identified as a change to K', and K'' as the new column object. D' is correctly created to reflect the design change to beam L.

In Stage 3 the re-integration of the data into the shared model instance takes place. Here, due to the wrong assignment of column N as a change version of K', a wrong support connection for N to column P on storey E will be suggested by the *undo* operation. This has to be corrected by the designer, leading to replacement of the version relationship R_{vn} (K',N) by R_{vn} (K',K''), and the support reference *supp* (N,P) by *supp* (K'',P) respectively. The new version N' of column N will be automatically created by the *undo* operation to reflect the changed support reference. Of course, the technical adequacy of the changed load-bearing structure of the building must also be checked and approved by the structural engineer. This cannot be a task for the data management system.

This short example gives an impression of the large potential of the suggested approach for goal-oriented reduction of the data to what is really needed for a particular design task, at the same time ensuring consistency and coordination of the shared model data. For real projects where a shared model instance can easily grow to several gigabytes this is a clear benefit in terms of space, time and efficiency of the collaborative work.

5 CONCLUSIONS

The presented delta-based versioning approach provides a solid basis to manage the data changes created during design tasks that are performed as long transactions to a shared model database. Typical implementations of such databases are seen in Web-based model server environments (Eurostep 2003, Houbaux et al. 2005).

In order to tackle existing interoperability problems the design step is subdivided into *three stages*, namely (1) selection of needed design data, (2) modification of selected data and (3) re-integration of the changed design data to update the shared product model instance. Each of these stages is stored in terms of the generated changes in a *version model*, thereby allowing reviewing of each stage by the other designers. Capturing of interdependencies between design states is provided via a basic *change vocabulary*, which allows also to deal with design refinements, such as type change, splitting and unification of objects. This provides for higher flexibility to capture design steps compared to the ID-based concepts of more traditional object oriented approaches. The changes themselves are managed by a set of *deltas* which represent a data change and thereby reduce the amount of needed data for versioning significantly.

Based on the suggested approach, the management of design data allows to:

– get access to every design states created in the design process,
– be aware of data changes between different design states, and
– support the roundtrip of model subsets.

The approach enables active user involvement in the management of shared product data in a collaborative work environments. It can be realised on short-term to enhance current integration methods and eliminate much of the existing deficiencies.

6 ACKNOWLEDGEMENTS

The authors wish to acknowledge the support of the German Research Foundation (DFG) within the framework programme SPP 1103.

REFERENCES

Amor, R. & Faraj, I. 2001. *Misconceptions about Integrated Project Databases.* ITcon Vol. 6. http://itcon.org/2001/5/

Bazjanac, V. 2002. *Early lessons from deployment of IFC compatible software.* In: Proc. of the 4th ECPPM, Balkema.

Eurostep 2003. *Eurostep Model Server.* http://ems.eurostep.fi/

Houbaux, P., Hemio, T. & Karstila, K. 2005. *SABLE - Simple Access to the Building Lifecycle Exchange.* http://www.blis-project.org/~sable/.

IAI-ISG 2003. *IFC 2x View Definitions*, available from: http://www.bauwesen.fh-muenchen.de/iai/iai_isg/doc/IAI-ISG-88.htm

Katranuschkov, P. 2001. *A Mapping Language for Concurrent Engineering Processes.* Ph. D. Thesis, TU Dresden, Germany.

Turk, Z. 2001. *Phenomenological Foundations of Conceptual Product Modelling in AEC.* International Journal of AI in Engineering, Vol. 15, pages 83-92.

Weise, M., Katranuschkov, P. & Scherer, R. J. 2003. *Generalised Model Subset Definition Schema.* In: Proc. of the CIB-W78 Conference 2003, Auckland, New Zealand.

Weise, M., Katranuschkov, P. & Scherer, R. J. 2004. *Generic Services for the Support of Evolving Building Model Data.* In: Proc. of the Xth ICCCBE, Weimar, Germany.

Westfechtel, B. & Conradi, R. 1998. *Software Configuration Management and Engineering Data Management: Differences and Similarities.* In: Proc. of the SCM-8 Symposium on System Configuration Management, Springer.

Wix, J. & Liebich, T. 2001. *Industry Foundation Classes IFC 2x, © International Alliance for Interoperability.* http://www.iai-ev.de/spezifikation/IFC2x/index.htm.