INTEGRATION OF DESIGN APPLICATIONS WITH BUILDING MODELS

CHUCK EASTMAN, TAY SHENG JENG, ROY CHOWDBURY
Design Computing, College of Architecture, Georgia Institute of Technology, Atlanta, GA. USA
KIM JACOBSEN,
Department of Planning, Technical University of Denmark, Denmark

Abstract:
This paper reviews various issues in the integration of applications with a building model. First, we present three different architectures for interfacing applications to a building model, with three different structures for applying maps between datasets. The limitations and advantages of these alternatives are reviewed. Then we review the mechanisms for interfacing an application to a building data model, allowing iteration execution and the recognition of instance additions, modifications and deletions.

1. INTRODUCTION

As the various professionals in the building industry become more experienced in the use of Information Technology (IT), they will utilize an increasing number of intelligent design applications. They are likely to include computer applications for designing special facilities, for designing with and detailing a range of construction technologies, and applying a wide range of performance analyses and/or simulations. Numerous other applications will emerge for design and detailing of proprietary products. Overall, this is the anticipated transition to knowledge-based design, where each of these applications encapsulate the knowledge in some specific area.

Given this growing diversity of applications, design will become increasingly characterized by the use of different specialized representations. We are skeptical of efforts to define a single representation or data structure able to support all important applications. Each varied representation has some shared data that is organized differently for each use. Some applications also require unique data not used in the others. Each organization of data is called a *class*. Because of the need to use many of these applications in an integrated way, we assume that integration facilities will eventually become available to support the use and dynamic extension of a suite of applications. We assume that these integration technologies will be centered around a building product model. Here, we consider a *building product model* to be a set of technologies supporting the integrated representation of a building.

Because of the need for applications to use different classes, an important aspect of a building model are the routines that convert data from one class to another, called *maps*. Maps have received major attention recently in research [Verhoef, Liebich and Armor, 1995], [Khedro, Eastman, Junge and Liebich, 1996] and within the ISO STEP community [Bailey, 1996], [RPI, 1996].

In this paper, our interest is in the effective integration of applications with a building model during design. We are especially concerned with the process coordination involved in integrating intelligent applications, and to support the necessary information flows found in design processes, such as iteration, coordination and collaboration. In the next section, we examine and compare different methods of integrating applications with a building product model and their different capabilities and limitations. Then we turn to a finer grained examination of the structuring of application interfaces needed to support typical design processes. The work is presented as part of the ongoing work of the EDM research group, now at Georgia Institute of Technology, Atlanta.

2. ARCHITECTURES FOR INTEGRATING APPLICATIONS

There are a range of methods for interfacing applications to a building model. Here, we review three different architectures. Each is described, then their capabilities and limitations are reviewed.

Because a building product model may include more than just a description of a building, but also maps and other data exchange technology, we distinguish between the *building data model*, consisting only of the structure describing the building, from the *model environment*, which includes the building data model and other associated exchange technologies. We also distinguish between two types of maps. Some maps are between an application and the building data model. They are partially outside of the model environment and thus are called *external maps*. Other maps are between entities that are within the model environment and are called *internal maps*. All maps are of one of these two types.

2.1 INTERFACE ARCHITECTURE CRITERIA

There are a wide variety of issues that can affect the usefulness of how applications are interfaced with a building product model. This review is based on five kinds of criteria:

Visibility of Conversion Logic: A shortcoming of most data exchange processes is the lack of visibility regarding what data will be produced. For example, in a system that uses only NURBS surfaces, when mapping to another system using a variety of surface types do all surfaces remain NURBS or converted to their simplest equivalent? Often, more than one conversion is possible for an entity. One conversion may be preferable during early design stages (the simplest) and another (the most accurate) at later design stages. In practice, hand tuning is often required and often some custom programming when such exchanges are expected to be repeated. Also, the instances to be exchanged may not be visible for review. By making all class conversions

internal to the model environment, the user potentially has access to controls for all aspects of the exchange process.

Allowing Updates From Multiple Sources: Data exchange processes have focused almost exclusively on file-to-file exchange among pairwise applications. In design, a repository allowing multiple applications to build up a building model, from which others can read and then write, has a long history as a desired alternative[Carrera and Kalay, 1994]. However, the updating of such a model is complex, and supported differently by each approach.

Allowing Incremental Updates: In addition to the general mode of update, part of a practical working design environment is to control the entity instances examined within an application, by selecting or filtering the desired set. This reduces the size of exchanged datasets and allows focus on specific issues, for example in dealing with coordination of changes.

Extensibility of the Building Data Model: Extensions to the building data model allow changes to the building's enclosed activities, to the construction technologies used to build it, or to the kinds of analysis applied to it. Several researchers, including the co-authors, believe this is a fundamental requirement for a building model supporting design [Eastman,1991],[Galle,1995]. Yet some particular difficulties of making extensions to the building data model will be described, for which easy solutions do not exist.

Support for Collaboration: Collaboration methods include identifying changes made to the design by a user, or the differences among multiple proposed changes by different users. Other collaboration actions include methods to flag or propagate changes, automatic routing of changes for review and "whiteboard" tools allowing multiple people to share and update design data. Each of these collaboration capabilities requires certain facilities which may not be provided by the integration architecture. For example some types of collaboration require explicit representation of multiple versions of the data. Others require change propagation.

Each of these criteria are affected by the integration architecture. We now turn to the three architectures for linking applications with a building data model.

2.2 DIRECT MAPS TO/FROM A BUILDING DATA MODEL

What appears to be the simplest and most direct way of interfacing applications with a building data model is to implement maps directly to and from it. The approach is characterized in Figure One. Direct mapping means that maps both read and write the data structures of the application program and also must convert data to and from the structure of the building model. Such an approach can be used for pairwise data exchange as well as a repository for general integration. Direct mapping is the most common means to exchange data between two or more applications through a neutral representation. It is the kind of mapping most strongly supported by the ISO-STEP technologies, using application developed maps to an EXPRESS-defined building model [Bloor and Owens, 1995, Part 4].

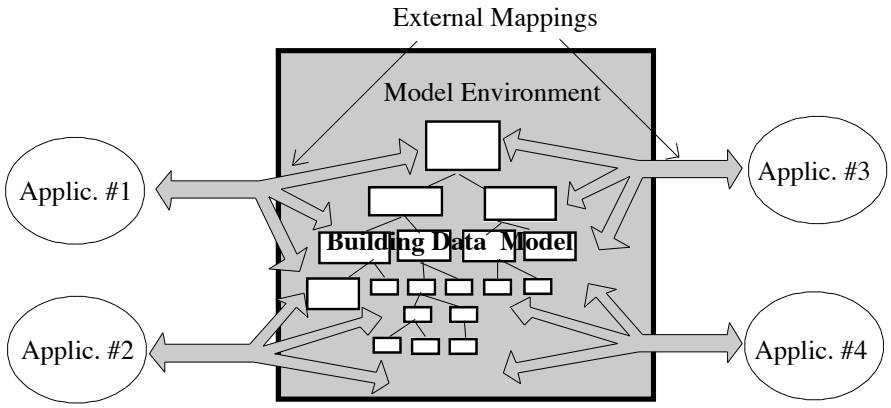

Figure One: A central building data model, with multiple applications mapping directly to/from it.

In direct mapping, the building data model is structured to support a specific set of applications. Typically, it will carry any information that is shared by two or more applications. The maps must do all class conversions needed between the model classes and the application classes. Because application data structures are only accessible in limited ways, the maps will be custom coded typically in a low level language such as C or C++ and the conversion of instances from one class to another will not be visible to users. The maps require significant coding effort and are not easily modifiable. To the degree that the new applications can access the data they need in the building model, new applications can be added. However, extending the central model to support new applications is usually not attempted, because the changes are likely to make incompatible the already existing maps.

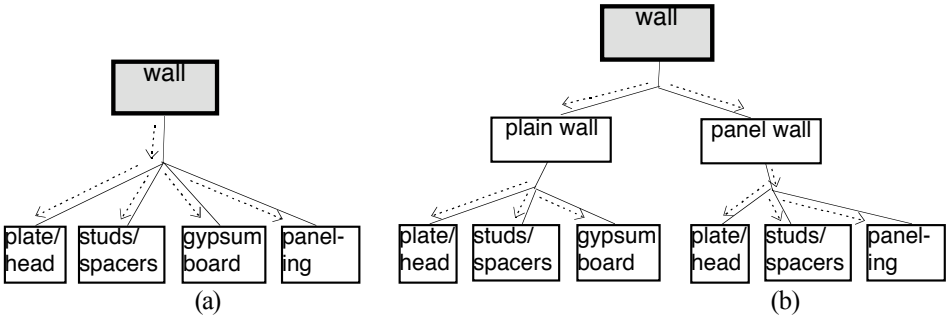

Figure Two: A wall model which is modified to accept an additional application. The addition may make maps for existing applications invalid.

To explain this limitation, we offer a small example. Suppose a wall is initially defined with a composition of its construction components, as shown in Figure Two(a). Access to the part attributes of a wall are through pathnames that access the wall then its part attributes, following the dotted arrows shown in Figure Two(a). Later, zones within the wall have to be added, representing, for example,

areas with different energy transmission properties. Adding these areas changes the access paths from the wall to its components, probably invalidating maps using them. Such changes would invalidate all mapping schemes known to the authors, including those proposed as extensions to EXPRESS. Thus while this approach allows limited extensions to be made, others are not supported.

Since the data stored in the building data model may be used by multiple applications, updates by applications should be cognizant of all the relationships in the building model that are needed by the applications. Because of the potential impact of such updates, in practice they need to be carefully reviewed before they are applied. For example management procedures might be developed to review all updates before they are accepted [Eastman and Shirley, 1994]. Such building models support strong vertical organizational structures, but not wide shallow networks emphasizing collaboration.

Maps typically extract all instances of the entity types used in an application. They may have conditions, for example in deriving loads, that only consider loads larger than some figure; thus they may have capabilities to filter selections of data instances. However, these filters are coded into the map and are not user tailorable. There is talk of developing facilities for user selection of entities to which maps apply [Hartwick,1997]. Currently, if an update is made by one application, then other applications must extract a full dataset and check manually what has been changed or learn through external messages.

2.3 MAPS TO DERIVED VIEWS FOR APPLICATION INTERFACES

A standard feature of relational database management systems is derivable views, a computed subschema of the database. The data in a view may be any subset of the database or derivable from it. A view is custom-built to provide the data needed for an application. In conventional use, the database manages the consistency of all view data, in that views are not stored, but computed from the building model as needed. Thus the view data is not redundant and cannot be inconsistent. This approach is different from the direct mapping method, described above, in that data extraction is a two-step process: (1) the first derives the needed data into a view, then (2) the data is mapped to the application. Both maps may do class conversions.

Application updates to the building model can be approached in either of two ways, with different impacts. One way is to allow an application to make an update back to its view. This is only possible in certain cases, in that many complex derivations cannot be reversed. For example, if an application uses the areas of spaces and these are derived from a floorplan and the application changes the area of some spaces, how should the floorplan be updated? Similar problems are encountered in view updates in object-oriented databases [Kim and Kelly, 1995]. The alternative way is for all updates to be limited to one or a few special applications, which can write to the building data model. This can work when a CAD system is used to create all data and all other applications read the data in order to analyze it. It is assumed that the special application can invert derivations and check that other relations are consistent. This is the approach characterized in Figure Three.

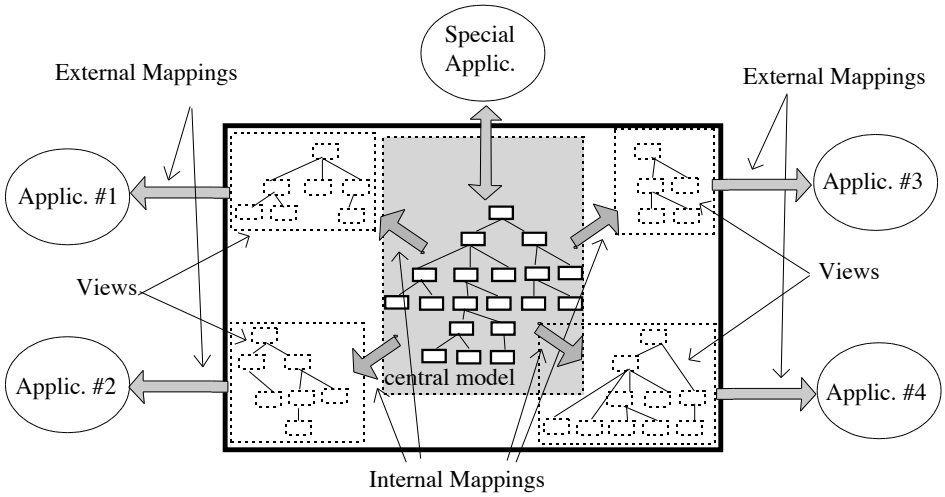

Figure Three: A Building model with derived views. The views can extract data from the building model. However, if updates are made, they must be done centrally, so the building model can manage the consistency

The map deriving a view, which is built into most database management systems, greatly simplifies the data extraction from the building model and can make some of the class conversions. Using the query language of the building data model, the model environment can extract and format the information making it easy to convert to the application through the external map. The external map is much smaller than in direct mapping, but still needs to be written in a low level language. Adding an application can be supported by writing new view generators, to the degree that the data they need can be derived from the building data model. Extension to the building data model requires massive changes, for the same reasons as the direct mapping approach.

The query language also supports user selection of instances, based on criteria such as timestamps that identify just-changed data. This can significantly facilitate coordination among a team of designers, because an update can carry just the data of significance, not the complete dataset. Additional, stronger forms of collaboration are not supported however; application data is not duplicated in the model; there is no notion of versions. Like the first approach, all design conflicts should be resolved before an update is made to the building model, forcing issues of coordination and collaboration to be made outside of the model.

It should be noted that even though SQL supports views, EXPRESS, used in ISO-STEP [Schenk and Wilson, 1993], does not. Bailey has defined an extension to EXPRESS, called EXPRESS-M, that supports views [Bailey, 1996]. Recently, EXPRESS-M has been merged into a larger mapping extension called EXPRESS-X [RPI,1996].

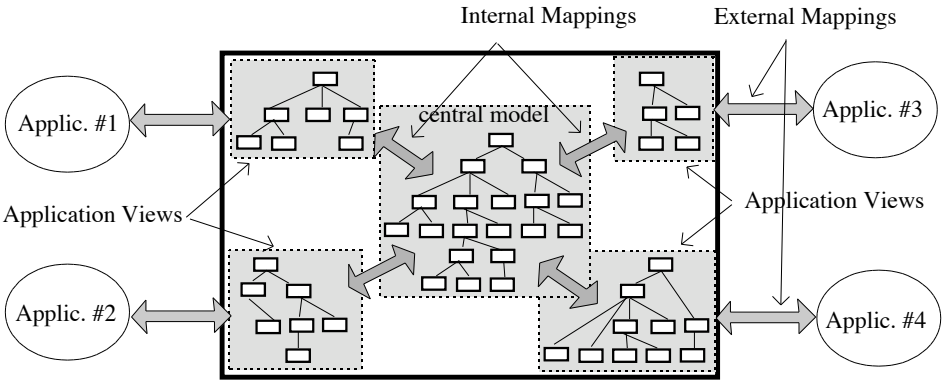

FIGURE Four: A schema architecture relying on design views which have their own models, and a central building model that carries equivalent DE classes.

2.4 SEPARATE DESIGN VIEWS WITH A CENTRAL MODEL

A problem with direct mapping is that the external map, which is probably written by the application developer in a low level language and thus is not public, makes all class conversions. What if, instead, the external map is simply a direct transfer of the application model into a format readable by the building model, and the building model makes all type conversions in internal maps. In this approach, external maps involve defining an isomorphic map from the application to/from an equivalent structure in a shared building model language. No class changes are required here. All the class conversions are made in the internal maps to/from the central model. The internal maps are part of the building model environment, and thus are visible and possibly adjustable. In such an approach, we call the application data model's view in the building model the *design view*.

There are several ways that the data within an design view may be organized. One is to define separate design views and a central building data model. To extract information into an application, it is first derived from the building model, with any class conversion needed, to the application's view, similar to standard derived views. However, the design view is stored (sometimes called in the database literature a "materialized view"). The application then accesses the design view data, does its design operations and eventually updates back to the design view. At this point, this view can be compared with the building data model or with other views -- as an issue of coordination. After review, all or some of its data may be mapped to the building model for later mapping to other design views. The central model carries the currently authorized building definition. This schema architecture is shown in Figure Four.

Here, all class conversions are made in internal mappings, which are part of the model environment. This allows them to be made visible and adjustable. Because redundant and potentially inconsistent design views may be carried in the model environment, collaboration tools can be developed to support resolving the

differences, for example by doing a difference comparison on views or by providing shared "whiteboard" views to others of proposed changes. A variety of change propagation methods also may be applied that allow a designer to assess the impact of a proposed change, by flagging aspects that may require additional changes, before it is submitted to the building model [Eastman, 1996], [Eastman, Parker and Jeng, 1997]. Thus this approach supports advanced forms of collaboration. Maps between the building data model and the design views can be developed to operate only on changed data, as we shall show in the next section.

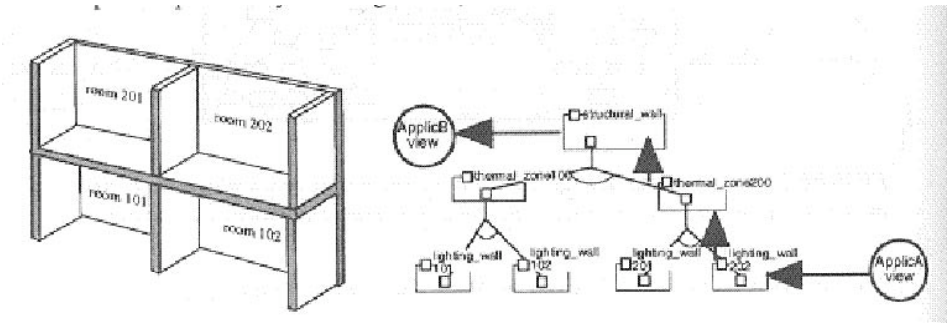

Figure 5: A wall is defined at several level of aggregation. Updates are made to the walls at all levels, by different applications. Maps are needed within the building data model to maintain the consistency of the different descriptions.

A problem not addressed in the first two approaches involves some issues of consistency maintenance within the building model. When some part of the design is created or modified, some other parts must also change. When the derivation is fixed and clearly defined, then there is not a problem. All data models are able to support derived data. However, in other cases, there is no single derivation path. For example, a wall may be part of several aggregated larger walls and any of these walls may be updated. This implies that in some cases, maps must exist within the building model, going in both directions between various entities in an aggregation lattice. A change in one level in the lattice may require updates to both the aggregations of the changed entity and its parts, and also derivable data. An example is shown in Figure 5. In the figure, a large wall is defined for dealing with structural shear, which is decomposed into the wall bounding each floor, representing energy zones. These are further decomposed into walls bounding singles spaces for lighting. Then if a single wall model, Application A, receives a window change, the translation to the aggregate wall in Application B, requires two maps between levels of aggregation and two class conversions, as shown in Figure 5b. In this arrangement, the translation of an entity from one design view to another may be a sequence of both changes of class within the same generic entity and a change in aggregation level. Translation becomes the concatenation of a series of individual class-to-class maps.

These maps allow changes to be managed up and down the aggregation lattice within the building model. For further examples, see [Eastman and Jeng, 1997]. Such maps are especially important if the building model is considered a repository for multiple applications, which incrementally update the model over time. For a

detailed presentation of the schema architecture based on design views, see Jeng and Eastman [1997].

Most often entity classes close to those in each design view will exist in the central model, particularly with concern to level of aggregation. Maps between design view and building data model will only deal with class conversions. If a new aggregation level is introduced by an added application, it must be added to the building data model when using the other two approaches. However, here the building model classes need not be revised if a new application requires some special classes, not supported in the central model. The classes at different levels of aggregation can be optionally defined within the design view and the maps placed there rather than the central model. For such cases, the central model structure does not change and the access paths of other application maps need not change. However, if multiple design views need the class data, it should probably be moved to the central model, allowing only one set of aggregation maps to serve all applications, rather than a separate set within multiple design views.

2.5 SUMMARY ANALYSIS OF INTERFACE ARCHITECTURES

The three architectures for interfacing applications to a building data model are summarized in the columns of Figure 6. The desired capabilities are shown in each row, along with the capabilities and/or limitations of each approach.

As seen in Figure Six, direct mapping is primarily a file-to-file exchange that does not support well iterative translation between applications or collaboration. It seems most appropriate during transitions between design stages. It requires strong review processes and policies if multiple updates are to be made to a central model. A derived view approach supports updates by one or a limited number of privileged applications. Some maps are visible, while others are not. In other ways, it is similar to direct mapping. The approach based on design views provides strong visibility of all aspects of the conversion process. It also strongly supports all forms of updates and collaboration. It does not strongly support model extensibility, though it is better in this regard than the others. EDM-2 relies on a building model architecture using design views, the third alternative [Eastman and Jeng, 1997].

	Direct Mapping	**Derived Views**	**Design views w/ Central Model**
Visibility of Conversion Logic	not visible; all in external maps;	limited visibility of those maps in view generators;	completely visible in internal maps;
Allowing Updates From Mutiple Sources	should reflect all aspects of the design; requires careful preview;	must be made by special application; updates by multiple applications not supported	updates made to isomorphic view; any application can make updates
Allows Incremental Updates	not supported	supports incremental reading, but all updates are limited	supported

Extensibility of Building Data Model	not practical	not practical	some extensions supported
Support for Collaboration	no support	limited support	strong support for collaboration

Figure 6. Table depicting the three application architectures reviewed and their response to the five criteria introduced in Section 2.1.

3. INTERFACING OF APPLICATIONS

Once an application architecture has been selected, many functional aspects of data exchange are determined. Other details of the application interface, however, determine the fine grained flow pertaining to the functionality of an individual update. Our interest is in the development of application interfaces that particularly support collaboration among a group of designers. We assume that there are a number of applications that both read data from the building data model and write to it, possibly multiple times. These interfaces must support iteration and design revision. Here, we review the issues and identify criteria for such an interface. Then we describe alternative strategies for achieving the criteria. We assume that the building data model is based on the multiple design views approach presented in the last section, and shown diagrammatically in Figure 4. That is, data exchange consists of isomorphic external maps and class varying internal maps.

General criteria for an application interface should include:

a) an application should be able to generate an initial part of a design and submit it to the building model; if not changed by another application, the initiating application should be able to retrieve the submitted work and continue, as if it was stored locally. That is, *the mappings between the building model and application should be isomorphic and complete*;
b) the application must be able to update part of a design, in a manner allowing modification of specific design entities that are identifiable within the building model, and that can be distributed back to other applications. To do this, *the external mappings must allow maintenance of the identity of the entities it receives and updates*;
c) updates by internal maps to the building data model and then to other applications must maintain the identity of entities, so that changes made by other applciations can be propagated back to other applications that use the data; that is, *entity identity musy be maintained throughout the building model;*
d) *the application interface should support making incremental design changes to the product*, which includes modifying previously defined entities, adding new entities, deleting entities.

A higher level and more general requirement is that the model environment be able to provide significant aid in managing the integrity of the design, as relations are

defined, assessed or analyzed and satisfied. Some aspects of this capability is provided by EDM-2's constraint management capabilities [Eastman, 1996], [Eastman, Parker and Jeng, 1997]. This capability can apply to any data carried within the building model.

3.1 SEMANTIC DEFINITION OF A DESIGN VIEW

An important top level issue is the semantic characterization of a design view. Is it simply an interface to an application, supporting all uses of that application? What about multiple tasks undertaken by different team members, using the same tool? What about using the same application on different parts of the design?

We consider an design view to be a unit of integrity, a unit of process and possibly a unit within the organization. A design view is a unit of integrity because an update made to a design view is assumed to be consistent within the view but possibly not across the whole building model environment. It is a unit of process, corresponding to a design task, in that two different design tasks may be undertaken with the same tool, but arrive at inconsistent designs, which must be resolved. Thus each task defined for a project, even though using the same tool as other tasks, should have its own design view. A design view is a unit of organization, in that it is notified when the integrity state of its data changes. It also has responsibilities in maintaining the consistency of the description within its view. Reporting and responsibilities suggest that it corresponds to a unit of organization.

In summary:
- a separate design view should be defined for each assignable design task;
- the same design view should be used when revisions are made based on that task
- sometimes tasks get revised, when the task scope significantly changes, it should be reflected in a new design view;
- multiple tasks using the same application should each have their own design views, for collaboration purposes.

3.2 INSERTS FROM AN APPLICATION AND LATER UPDATES

Applications can update a building data model in at least two different ways. One is that each time an application executes, including updates, it writes out the complete dataset, as if it were a new one. Another way is to make incremental updates to just the modified entities. Regardless of how the application update works to its own saved files, we consider it important that within the building model, support is given to incremental modifications to an existing set of entities. This has two implications: (1) partial updates designate only the design data that has changed, and (2) the identity of modified entities can be maintained across design views and the building data model.

Identity identification in incremental updates can be accomplished in two ways:
(1) use the entity IDs that are carried within the application itself. These will exist in all applications that themselves do incremental updates. The application IDs

may change from session to session if incremental updates are not supported. In this case, a table of ID conversions is required when entities are read from a design view to the application[1].
(2) Alternatively, a new ID may be associated with entities when they are read from a design view. This requires that someplace within each entity data structure there exists a slot that can carry the ID.

These two methods can be mixed, in that different application interfaces operating on the same product model can use different strategies to maintain object identity. In EDM-2 the initial interfaces have been defined using an additional ID added to each entity. The ID is added by the building data model whenever it receives an instance without an ID.

3.3 TWO-WAY MAPPINGS

Once an application has created an entity instance, all applications that read that data and possibly update it, must be able to report back to the building data model what instance has been modified, which ones deleted and also identify new entities that are added to the design view.

Since the entity IDs are assigned by the building model, not by the application, this provides a means to identify newly added entities. All entities written by the application to its design view that have IDs were earlier read from the building model; these are not new. All those without entity IDs are new and require the insertion of a new entity instance.

In reading from a design view, only some of the entity instances in the view may be read. When an update is made, all those changed can identified by doing a comparison of the old values with the new ones. If no change was made by the application, no update is made. All updated values also are flagged. In addition, however, some entity instances may be deleted and these cannot be distinguished from those not Read into the application. In neither case are they written back. In order to make this distinction, each entity instance within a design view has a flag field. If it is read for a session, the flag is set. All updates involving a change are flagged one way, while no change turns off the flag. Any entity instances whose flag remains set after an update have been deleted by the application.

---

[1] It is important that no operations can change the entity ID numbers during an application session. For example, the Bentley Microstation COMPACT operation changes IDs when it deletes flagged entities. This would corrupt the ability to read and write back entities, maintaining their identity.

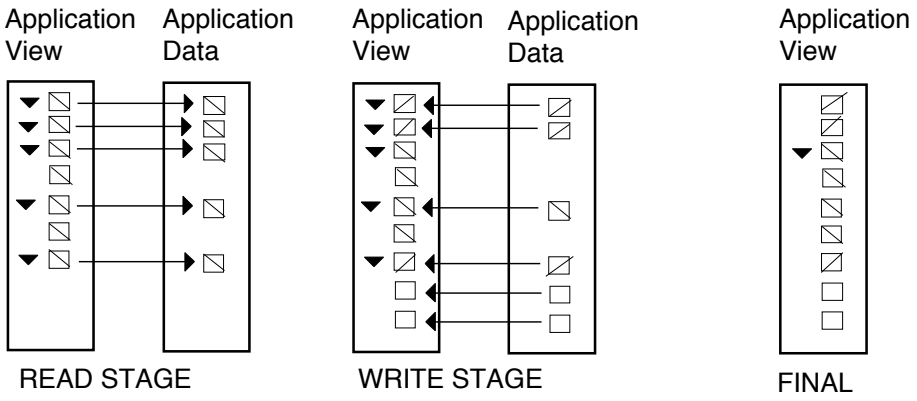

Figure 7. The management of updates from an application back to a design view.

Figure 7 diagrams an example of a design view and its transfer to the external application, its update and final condition. Within each stage, the design view and application dataset are shown as enclosing boxes. Within them, each entity is shown as a small box. In the Read Stage, each entity instance read is flagged, shown to the left of each entity. A slash in a box indicates that it has an ID. Because entities are assigned IDs by the building model, all entities read from the design view have them. After the application runs, it makes an update back to the design view. If the entity instance written back has been modified, its slash has been reversed. Three entities were modified. All entities with IDs are written back and the flags (triangular on left) in the design view are eliminated. Some entities are written back without IDs and these must be new. One entity is not written back and after the Write Stage, it is left with its flag on. This indicates deletion.

These bookkeeping methods allow unambiguous updates from an application to a building model, supporting addition, modification and deletion. They cover all cases of incremental updates, except one. If an incremental update is to be made, the subset of design needs to be determined at the time of reading from the application's design view. When data is written back to the design view, all the data must be written, so as to allow deletions to be distinguished. Partial write-backs from the application are not supported.

3.4 EXAMPLE

The initial capabilities of the incremental update operations and their support for collaboration can be demonstrated in their impleentation in EDM-2. In the example, an initial application has laid out the service core of an office building. This is written to the application's design view, then passed onto the building model. This base design is then passed to several other designers who will review and detail various parts. One of these is a person who will review and detail the stairway. The stairway portion of the design is mapped into this person's application design view, then externally mapped into the application. The stairway designer determines that one landing roust he widened and another stair tread added. The person updates the design view, where the changes and additions are identified. At this point, the person making the changes is notified, as shown in

Figure 8. The record of changes can also be sent to the original designer or to others potentially affected, using collaboration techniques presented elsewhere. Some collaboration capabilities are described in [Jeng and Eastman, 1997] and [Eastman, Parker and Jeng, 1997].

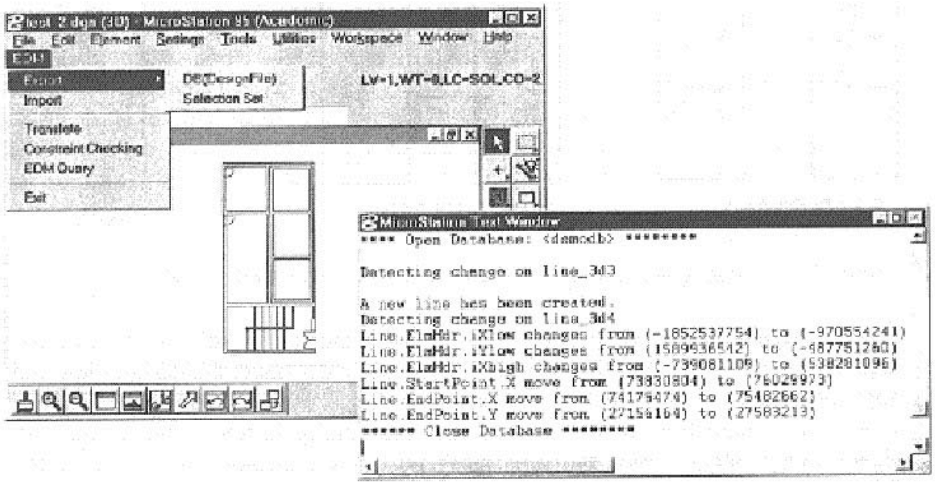

Figure 8. Example screen after a designer has updated a portion of the design of a building's service core, with a building model window in the application environment showing the effects of the update.

4. SUMMARY

As technology advances to support multiple designers using heterogeneous applications, many issues arise in the realization of an effective multi-user collaborative design environment. These issues include: supporting visibility of the conversion logic going to an application; allowing updates from multiple sources; supporting incremental updates: facilitating extensions of the building model to support new applications, and support for collaboration. We have shown that the architecture how applications are integrated with a building model greatly affect these issues, We believe that an architecture based on design views, our appraoch number three, is best able to support the needs of collaborative design. We have also presented details showing how to support incremental updates and flagging low level design changes. This capahilitiy appears to provide a framework potentially supporting high-level forms of collaboration.

Our overall goals are to develop the needed capabilities for creative team design both at a distance and temporally separated. and also for enhancing the support for designprocesses that respond to much higher levels of complexity.

NOTE: This work has been supported by the National Science Foundation, grant No. IRI-9319982.

REFERENCES

Assal, H and C. Eastman, [1995] “Engineering Database as a Medium for translation”, 1995 CIB W-78 Symposium, Stanford, Calif.

Bailey, Ian, [1996] EXPRESS-M Reference Manual, ISO TC184/sc4/wg5 N243, CIMIO Ltd, Brunel Science Park, Surry, England, 12 August, 1996.

Bloor, Susan, and Owens [1995] Product Modeling book

G. Carrera and Y. Kalay (eds.), [1994], Knowledge-Based Computer-Aided Architectural Design, Butterworths-Heinemann Press, N.Y..

Eastman, C.M., [1992], "A data model analysis of modularity and extensibility in building databases", Building and Environ., 27:2, pp. 135-148.

Eastman, C.M. and G. Shirley, [1994], "The management of design information flows", in S. Dasu and C. Eastman (eds.) Management of Design: Engineering and Management Processes, Kluver Press, N.Y.

C. Eastman, M.S. Cho, T.S. Jeng and H.H. Assal, [1995] “A Data Model And Database Supporting Integrity Management”,1995 ASCE Intern. Computing Congress, Atlanta, GA.

C. Eastman, H. Assal, and T. Jeng [1995] “Structure Of A Product Database Supporting Model Evolution”, 1995 CIB W-78 Symposium, Stanford, Calif.

Eastman, C.M. [1996] “Managing Integrity in Design Information Flows” Computer Aided Design (May, 1996), 28:6/7, pp.551-565.

Eastman, C.M., D. S. Parker, T.S. Jeng, [1997] "Managing the Integrity of Design Data Generated by Multiple Applications: The Principle of Patching", Research in Engr. Design (in process).

Eastman, C.M. and and T.S. Jeng, [1997] "A Database Supporting Evolutionary Product Model Development for Design" Automation and Construction (in process).

Galle, Per, [1995] “Towards integrated ‘intelligent’ and compliant computer modeling of buildings”, Automation in Construction, 4: 3, (October), pp. 189-211,

Hartwick, 1997 -- new paper

Eastman, C.M. and T.S. Jeng, [1997], "

Khedro, T. C. Eastman, R. Junge, and T. Liebich, [1996] “Translation Methods for Integrated Building Engineering, ASCE Conference on Computing, Anaheim, CA, Ju, 1996.

Kim, Won, and Wm. Kelly, [1995] "On View Support in Object-Oriented Database Systems", in Modern Database Systems: the Object Model, Interoperability and Beyond, W. Kim (ed.) Addison Wesley, 1995, pp. 108-129.

Rensselear Polytechnic Institute[1996] ISO TC184/SC4/WG5, EXPRESS-X Reference Manual, Working Draft, May 28, 1996, Lab for Industrial Infrastructure, Rensselear Polytechnic Institute, Troy, NY.

Schenk, D.A. and P.R, Wilson [1994], Information Modeling the EXPRESS Way, Oxford U. Press, N.Y.

Verhoef, M., T. Liebich, R. Armor, [1995], "A Multi-Paradigm Mapping Method Survey", Proc. CIB Workshop on Computers and Information in Construction, M Fisher, K. Law, B. Luiten (eds.), Stanford, U, pp. 233-247.