'EXTENSIBLE ENTERPRISE COMPUTING FOR CONSTRUCTION' AS A NECESSARY PRE-CURSOR FOR COLLABOATIVE ENGINEERING

Robert Aish
Bentley Systems

ABSTRACT: Our focus is to consider the construction industry as essentially an information processing system. In its ideal form, practitioners (each with an individual internal representation of design intent) interact with other practitioners by first interacting with an information processing system that manages various shared external representation of design intent. The underlying assumption (from an information technologist's perspective) is that design data is held in a sufficiently complete representation, and that changes to this representation are transactions that move the representation from one consistent state to another. We might call this 'enterprise computing' for construction. However such an approach by itself, might be insufficient because the construction process is essentially open-ended. New building types, design configurations and construction techniques are constantly being invented. This necessitates that any computing system to be deployed in this sector has sufficient extensibility to capture and manage new representations.

KEYWORDS: Collaborative Engineering, Long Transaction, Change- Merge, Design History, Extensibility, Schema Evolution.

1. INTRODUCTION

This ideal of 'enterprise computing' for construction can be compared to the realities of current practice.

- Due to its fragmentation, the construction industry generally perceives its use of information technology in terms of multiple discrete 'individual' systems (with the resulting proliferation of discrete documents) rather than as an enterprise systems.
- The drawing tradition, which represents building in 2D, with different representations of the same design split across multiple independently editable documents inhibits consistent management of design and the use of analytical tools.

While these may be familiar arguments, there are new object oriented and data management tools emerging from key software developer, such as Bentley Systems, that are designed to address the specific needs of a 'construction enterprise', namely geometric generality, multiple application semantics, multi-user access, and transaction management. These systems also address the scalability and reliability issues required for deployment in practice. Again, arguments for (and advantages of) systems of this type have been discussed in the research literature for more than two decades. The difference is that these systems are ready for deployment.

But with this prospect for a broader application of 'Enterprise Computing' for Construction, there are associated other significant issues which may concern both the 'strategic' and the 'creative' practitioners, namely:

- Semantic completeness: building a sufficiently complete multi-disciplinary representation of design intent
- Data integrity: where any intelligent components are used, these should not become 'orphaned', for example, by object "instance" data being detached from the definitions of the corresponding class
- Data longevity: the integrity of design and other data should be maintained for the life-time of the building, across new hardware platforms and operating systems. Upgrades to the application and any intelligent components should not disrupt or invalidate existing data
- Parallelisation of design: individual designers or engineers should be able to work in parallel, and then be able to synchronize their changes to design data with co-workers
- Expressibility: architectural design and construction engineering are open-ended domains. Additional intelligent components should be capable of being added on a "per project" basis.

Within this context, this paper will explore the essential 'tension' that exists within the Architecture and Construction sectors. On the one hand, there is a perceived need by construction managers for computing tools based on clearly defined and agreed schema to control the construction process (thereby giving economic advantage, comparability, etc.). On the other hand, creative designers who are under other competitive pressures, are expecting a different set of computing tools to allow the exploration of new building configurations and construction geometry. While in the former case a standardisation of schema (as the foundation of a traditional "Enterprise Computing" system) would appear to be in order, in the later case the essential 'open-ended-ness' of the creative process demands "extensibility" as a pre-requisite of any computing system.

These differing requirements (and indeed, attitudes) within the user community, presents software developers with interesting challenges. What technologies (for example, object and/or relational) and what 'domain abstractions' are appropriate foundations for solutions for these differing requirements. Or indeed, what technologies and 'domain abstractions' can be used as the basis for broader set of applications whose design is intended to unify across this apparent "management-creative" divide…hence the theme of this paper: "'Extensible Enterprise Computing' for Construction".

Fundamentally, this is not exclusively an issue of technology. We need to address both the technical and cultural issues if we are to realise our collective ambition of providing effective tools with which to support collaboration between the diverse range of interests that occur within the Architecture and Construction sectors.

2. RELATING COMPUTING CONCEPTS TO ENGINEERING ENTERPRISES

To pursue this argument we are going to consider the following computing concepts and the related concepts in the world of enterprise computing, and observe their use with the ProjectBank data repository:

Computing Concept	Related Enterprise Computing Concept
Normalisation	Model v. Report (or Drawing)
Transaction	Consistency of Design
Long Transaction	Parallelisation of Design
Change Merge	Coordination (synchronisation)
Revisions	Design History

3. NORMALISATION AND MODELS V. REPORTS (OR DRAWINGS)

The concept of normalisation of data, requires that an item of data is recorded only once. Any subsequent use of that data requires a reference to the original item. The computing system that supports normalisation must maintain 'referential integrity' between such secondary uses of the item and the original data. A collection of such original data items might be thought of as a model (in the most general sense). Reports may be derived from such a model. It is inappropriate to edit or change derived data. If an item of data (in a report) is required to be changed then the original (model) data must be first changed and the report re-derived.

The 'model' is the single representation of reality which is accepted as being sufficiently complete. In the context of construction, it is unlikely that the geometric dimensionality of that model can be less than that of the corresponding reality (i.e. 3D). Drawings are a form of report (but not the only type) and should be derived from the model. As with any reporting process, a report will be consistent if the underlying model is consistent. Additionally, all drawings/reports extracted from a model at a given moment will be mutually consistent.

Central to the operational efficiency of a model and drawing/reporting system is to establish complete bi-directionality between the model and the report. First, it is extremely useful to support 'back navigation' from items in reports to items in the model, to help in identifying how to update the model. (For example, I see a wall in this drawing. I wish to change the wall. Please navigate to the definition of the wall in the building model so that I can effect the required change). Second, if the model is changed it may be useful to be notified as to which parts of the drawings/reports are invalid, and to regenerate only these regions. (MicroStation TriForma is an example of a model and drawing/reporting system which supports this set of functionality)

4. TRANSACTIONS AND CONSISTENCY OF DESIGN

Transactions are an indivisible operation on data that starts with the data in some consistent state, and leaves that data in a new or different consistent state. So a transaction can be thought of as a mechanism for making changes to data, while maintaining the consistency of that data. Transactions are individual (or atomic) in the sense that it is not possible to complete only part of a transaction. The transaction must be completed in its entirety or not at all. If the transaction is completed it is 'committed'. If it is not completed, the data is 'rolled back' to its original state.

In the context of construction, we can use transactions to maintain the consistency of design data, but this in turn depends on how much of the semantics of design are made explicit within the system.

At the lowest level, a transaction-based system must maintain referential integrity. For example, if a window is placed in a wall, then the wall maintains a reference to the window. If the window

is to be deleted, then the wall must be notified that it no longer contains a window, (hence setting the reference to the window to 'null') before the window is deleted, otherwise the wall will hold a reference to a non-existing object. This example is basic house keeping and does not encapsulate any significant application semantics.

We can consider an example of a transaction-based CAD system which does incorporate some application semantics. This system might provide a single operation for adding a window to wall, which combined the process of placing the window at the desire location in the wall AND the process of cutting the appropriate size hole in the wall. (With a similar consistent 'remove window' operation that both deleted the window and restored the wall). These two processes (placement and hole cutting) are quite distinct (requiring quite different manipulations both in a computation geometry sense and in real construction terms). The essential nature of the transaction approach is to combine related process into a single all-or-nothing operation. We can immediately see that combining a modeling approach to design with a transaction approach to data management begins to give added advantages to the users.

We can imagine other CAD systems, which would allow the window to be placed without cutting the hole, or allow the window to be removed without restoring the wall. These operations might result in a window frame superimposed on a wall, or an opening in a wall without a window frame. These states may or may not be considered to be a valid and consistent for the design, depending on one's approach to construction semantics. Both these (possibly inconsistent) cases satisfy the minimum requirements, that of referential integrity, but may not meet the requirements of some higher-level application semantics. Again, it is a matter of debate how much a CAD system should impose application semantics and validation rules on users. Certainly it is an advantage to have this option available.

Finally, we might not be using a modelling system (in the specific sense of the word 'model' referring to 3D building representation). Instead we may be using a conventional 2D drafting system. Previously, we referred to a model and to a drawing extracted from the model as a form of report (i.e. drawing/report). The fundamental difference here is that each drawing is essentially an independent model (in the general sense of the word 'model') (i.e. a drawing/model). Here it might still be possible to use a transaction based approach, to add a window to a wall (for example) in the plan drawing/model, but without adding the window to the wall in the 'corresponding' elevation drawing/model. Each drawing/model is itself consistent, but in the drafting systems no attempt is made to maintain the consistency between the different drawings. Transactions can change the state of one model, but has no rules to maintain the state of any other models. Indeed the correspondence of models or items within such models may only be in the mind of the users, and is not explicitly recorded and therefore is not available for computation use.

The conclusion here is that while modeling approaches to design and the concept of a transaction are each extremely useful in isolation, the combination (resulting in transactions based on domain specific validation rules) multiplies the values of the separate functions.

5. LONG TRANSACTIONS AND THE PARALLELISATION OF DESIGN

The idea of a transaction is fundamental to maintaining the consistency of design data. However, the transaction only relates to an individual user. If we have multiple users (such as a project team) then there is the conflicting need to share data with one's colleagues but also to work at an individual level on a sub-set of the project data, possibly over an extended period of time.

As we know, it takes time to create or modify a design. During this gestation period, various design alternatives may be explored. During any one of these design cycles the design may be incomplete. The individual practitioner may not want to publish his work to his colleagues in the design team until his work is complete. In a 'shared' short transaction system, there is the possibility of all users seeing the changes of all other users immediately these occur. This is unrealistic and most probably undesirable. The user who generated a particular change does not want to publish incomplete intermediate solutions, nor does he wish to be bombarded with other users' incomplete intermediate solutions. The idea of a permanently shared repository with constantly updating short transactions is generally not thought to be appropriate to the work of design teams.

The idea of the long transaction is to enable users to work on a copy of all or part of a model for an extended timescale. The long transaction begins with the selection of the model or sub model to be worked on. The long transaction may include a whole series of short transactions, but these are completely private to that user. The long transaction finishes when the user commits his changed model back to the central shared data repository.

One approach is to adopt a pessimistic strategy where, for any shared data, there may be many readers but only one writer. The problem here is that the first user to 'grab' a particular data set (or file) for modification and update, essentially locks out the remaining users. Instead, we have adopted an optimistic strategy which allow the same data to be simultaneous modified by more than one user. There are three reasons for this:

First, not to allow an optimistic strategy would create a complete blockage to parallel working. Second, we are assuming that management will usually coordinate members of a design team so as not to allow users to make conflicting changes to the same data items. Third, we have created a technology which helps to resolve conflicting changes, if these should occur. This is not a closed system with hard coded rules. This technology enables future application developers to define their own rules for identifying and resolving conflicting changes, since what constitute a conflicting change depends on the specific context and application semantics

Long transactions extend the concept of the transaction to a multi-user team and enables the parallelisation of design (which is fundamental for collaborative engineering).

6. 'CHANGE-MERGE' AND COORDINATION (SYNCHRONISATION)

As we have seen, long transactions allow multiple users to start parallel design sessions. At the end of these sessions, there is a need to gather these parallel 'strands' back into a single unified and resolved design statement.

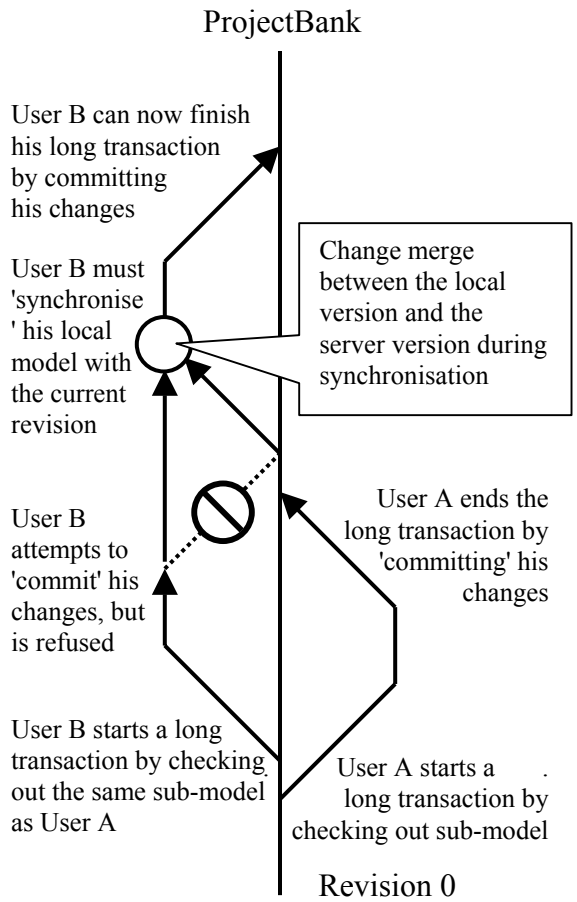

Fig. 1 Project Bank workflow

One of the underlying themes which we are focussing on, is to maintain the consistency of design data. It is important to remember that in order to maintain this consistency, it is only possible to finish a long transaction (by committing the data back to the data repository) if the differences between the local (workstation model) and the repository (server model) are due only to the changes that the user has made to his local model and that the data in the repository is unchanged since the start of the transaction.

Let us imagine that we have a ProjectBank repository with data at revision 0, and two users, A and B (Fig. 1). User A starts a long transaction, then user B starts another long transaction. At the end of A's long transaction he is able to commit his changes back to the repository, because the repository is in the same state (revision 0) as when A started his long transaction. The only changes that are relevant have occurred on A's workstation. When user A commits his changes to the ProjectBank repository a new revision (1) is created. The difference between revisions 0 and revision 1 represents only the changes made by user A.

By committing his changes, user A has essentially published these to his co-workers. User B is notified that there are changes on the ProjectBank repository that he does not have on his workstation. Now user B wants to commit his changes. But the repository is at revision 1, and is no longer in the same state as it was when B started his long transaction (which was at revision 0). For a whole host of reasons, B cannot continue with his 'commit'. Instead he must "synchronise" his version of the data with the current state of the repository (including the recent changes committed by A).

The synchronisation allow User B to see (and possibly respond to) A's changes. However another reason for synchronisation is to prepare B's model to be 'committed' to the ProjectBank repository. The pre-condition for a 'commit' is that the only difference between the ProjectBank repository and the model on User B's workstation are the changes made by B. So having

synchronised with the current state of the repository, B is now free to commit his changes, resulting in a new revision on the ProjectBank repository (2).

So far we have described the activities of User A and B as 'changes' without going into the specifics. Essentially there are two types of changes: non-conflicting and conflicting. A non-conflicting change might occur when two users change some unrelated or incidental attribute of an objects, for example, one user changes the location of an item, while another user changes the item's colour. A conflicting change might occur when two users each make different changes to the same attribute of an objects or an inconsistent change to two (or more) related attributes of an objects, for example, one user changes the length of a beam and another user changes its depth. What constitutes a conflicting or non-conflicting change depends on the application semantics and the related validation logic.

In creating an application schema, we require the software developer to implement a 'changeMerge' method. This method can encode the rules which allow alternative versions of an object to be queried to determine if they are essentially compatible (and therefore can be automatically merged) or are incompatible. If the later is the case, then both versions are presented to the user who can choose which version should be accepted

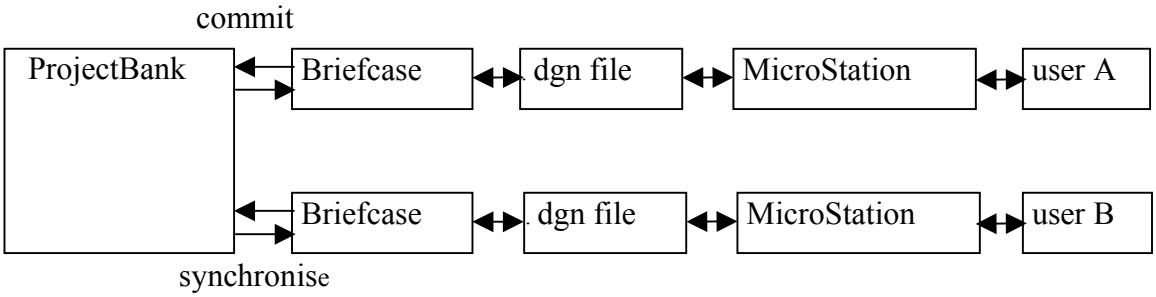

Fig. 2 ProjectBank used as a shared repository between two users

We can see that that the change-merge process is not (and necessarily cannot be) completely automatic, particularly when conflicting changes are encountered. Resolution of conflicting changes requires human intervention. Therefore this is a process which must be carried out, not just with the user's participation, but under his control (on the client application). It cannot be carried out on the server side (within the repository). Therefore the user must update (or synchronise) his version of the model, by down loading the current state of the model from the repository and resolving any conflicting changes locally.

To help in this process the user has access to a Revision Explorer (Fig. 3) which identifies unchanged elements (in grey), added elements (in green), deleted elements (in red), changed elements, in their pre-changed state (in light blue) and in their post-changed state (in dark blue). The user is then in a position to commit his changes, because at this moment the only difference between the version of the model on the repository and his (client) version are his uncommitted changes.

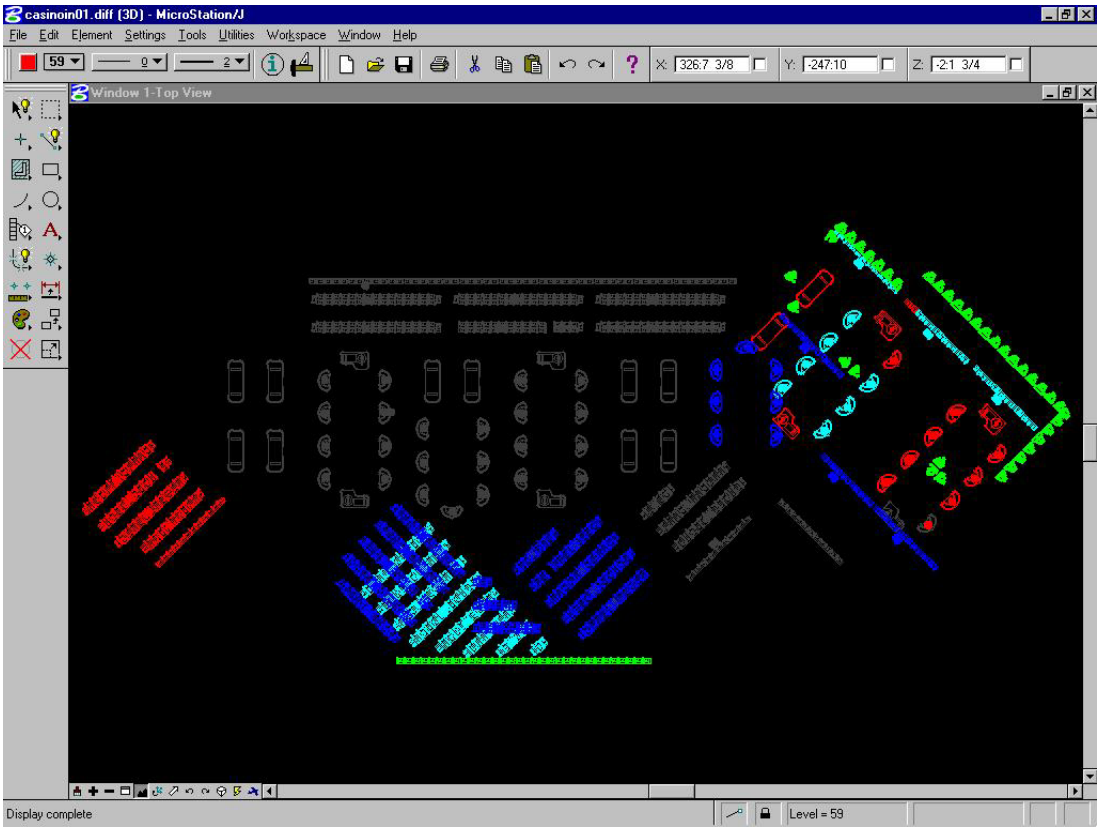

Fig. 3 ProjectBank 'Revision exporer', allows users to compare two versions of the same design, which identifies unchanged elements (in grey), added elements (in green), deleted elements (in red), changed elements, in their pre-changed state (in light blue) and in their post-changed state (in dark blue).

7. REVISIONS AND DESIGN HISTORY

Every time a user 'commits' to the ProjectBank repository a new revision is created. This is not a complete copy of the model, but rather the changes (or 'delta') from the previous revision. There are two advantages here: First, the storage is much more compact than the equivalent set of files (which would have laboriously duplicated all the unchanged data as well). Second, ProjectBank explicitly records all changes to all items at the 'component' level, not at the file level. This means that the user can use the 'Element History' function (Fig. 4) to review changes to a specific item or component or group, on a change by change basis. In addition the user can compare the complete design model or project at different moments in time, on a component by component basis. As we can see ProjectBank is NOT a document management systems.

This functionality opens up important management scenarios. For example, an item, which had been deleted in a previous revision, is not permanently lost, because it is available in the preceding revision. In fact, it can be effectively undeleted, by being brought forward from the revision preceding its deletion to the current version of the model. The generality of this is that the user can construct a new version of the model, by selectively including items from any

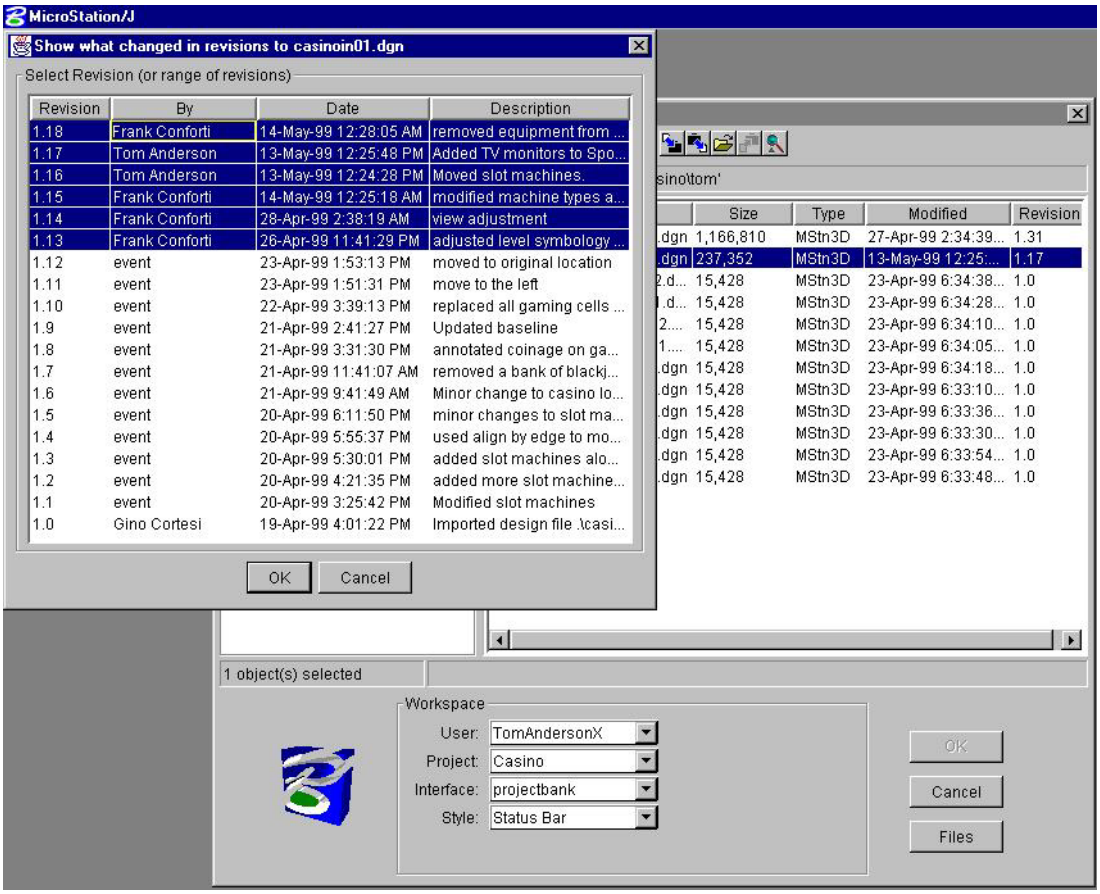

Fig. 4 ProjectBank 'Element History' function allows the user to review changes to a specific item or component or group, on a change by change basis and to precisely identify "who did what, when". It provides a complete 'audit trail' of the design process.

previous revision. When 'committed' to ProjectBank, this becomes the latest revision, but all the previous revision are still intact.

The ability to 'play with' design history in this way helps managers to monitor the evolution of design, to identify and value the contribution of individual users, to combine features from alternative design and to re-use previously discarded concepts.

8. BENEFITS

The fundamental advantage that ProjectBank offers is the opportunities for greater consistency and control of information. In particular the ‘annotation’ of transactions naturally provides an audit trail. The explicit parallelisation of design open new management possibilities, with the opportunities for ‘ownership’ of this management function. As with any 'computerisation' we have to accept that there may be losers as well as winners in this process.

When information is 'shared', how do previous ownership conventions get reformed. Here we are not talking about conventional 'data exchange' but a more dynamic operation where multiple consultants may be accessing and revising design data at the component level within a single

repository. The revision history will faithful record "who did what, when", but the concept of the ownership of data may necessarily need to be re-caste. For example, the architect may draw the initial wall. The engineering consultant may suggest and implements a modification to the wall. Does the wall still 'belong' to the architect? Or is the wall 'owned' by the last person to modify it? What happens if another consultant undoes the modification of the engineer (puts it back to state in which it was left by the architect.) Does the wall now 'belong' to that consultant (who actually made the last modification) or to the architect (who defined the state in which the wall is currently)? Do we own the process or the product (the result of the process) or do we each own the modifications (that part of the process) which we contributed? Certainly, with the 'Element History' function and the 'Revision Explorer' we can, for the first time, accurately track and audit this whole activity. Hopefully this will open up new and real opportunities to collaborate and to move away from the adversarial tradition.

ProjectBank is the first software product built with JMDL technology, (JMDL = Java MicroStation Development language). JMDL is a complete superset of Java but with the additional functionality of persistence and transaction management built into the 'Virtual Machine' (or VM). The logical extension to ProjectBank is to use JMDL to build more capable editors for "Engineering Component Modelling" (or ECM). In fact, ECM uses ProjectBank as its native file format, so ECM will have collaborative engineering built in from day one. ProjectBank starts the migration process to ECM, with the prospect of advanced applications which can encapsulate engineering knowledge. With ECM there is the opportunity for multiple applications "schema" (A schema is a collection of related classes that are intended to work together and to support a particular application domain or workflow). These schema can encapsulate the components and representations which characterise that particular application domain. The schema can enforce application or project rules (for example on 'merge change' and transaction validity). The schema can expose different components (or data types) to the user or to external applications. Some of the issues we are currently addressing with the development of application schema involve the implementation of more general abstractions, including dependency, deferral and extensibility.

In the short term, ProjectBank enables users to gain immediate benefits of collaborative engineering, with no discontinuity with legacy data and applications, while in the long term, ProjectBank enables users with existing MicroSation data to work with early adopters of ECM applications (Fig. 5).

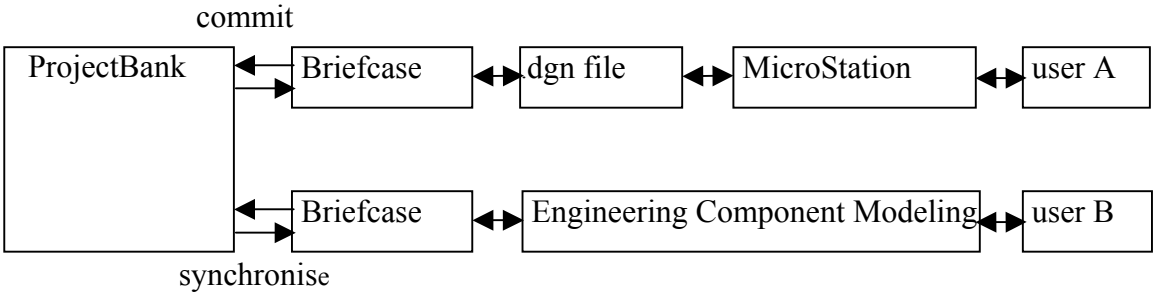

Fig. 5 ProjectBank as a transitional mechanism, to allow early adopters of MicroStation Engineering Component Modeling to work with legacy data.

9. EXTENSIBILTY AND SCHEMA EVOLUTION

Construction Projects occur over time. The whole life-span of a building, from inception, through use and refurbishment to demolition can span decades. During this time new functional user requirements will be established and new construction technologies may be invented. The converse may also be true. Components used in the original building may not be available during a subsequent refurbishment. Even if such components are available, then current versions of these components may have a revised specification, which may include additional attributes or references to other components or documents. In addition, the computing systems on which the original building was designed may not be available during the refurbishment or controlled demolition phase. If we base our design models on object-oriented technology then it may not be possible to even view the design data except by executing display methods on the design objects.

At the moment we are in a transitional phase. Some object oriented design applications are being used which offers more advanced representations and design semantics than were previously available in CAD systems (based on geometric primitives and procedural code). However, because of the apprehension about the longevity of object oriented design systems, current archiving practiced is based around the use of much lower semantics levels, for example extracted drawings, plot files and even raster file formats, with a consequential loss of structure.

An alternative approach looks at the fundamental computing issues involved. Java (and JMDL, which is a true super set of Java) is designed to be a multi-platform application language which runs in a 'Virtual Machine'. Applications are not 'ported' to different systems, only the Virtual Machine is required to be ported. The applications run unaltered in the Virtual Machine (although the use of "just-in-time" compilers (JIT) for specific hardware can give performance improvements). The advantages of the virtual machine, as original proposed, was to enable the same applications to run on different hardware and operating systems, essentially in the same "time domain". But in the context of the extended time span of construction projects we can foresee the use of Virtual Machines as an important technology to enable access to object oriented design model on different base systems across different "time domains".

Another critical factor is to maintain the integrity of the object (data) and class (definitions). In traditional applications (and indeed in many object-oriented applications) data is stored in one format and class definitions in another (DLL's). This raises the possibility of data and programs (or objects and classes) getting separated, again with unfortunate consequences. The solution is to encapsulate the class definitions in the object data store.

Finally, it is recognised that the representation of buildings evolve over time. This requires that the class "schema" should be capable of evolving, to adequately reflect changes in the application domain. In addition, data from previous schemas need to evolve so as to be accessed by current applications. Schema evolution impose strict rules on software developers. These rules require that the 'public interface' (or API) of a class can be extended, but not diminished. However, the behaviour of the methods can be changed, for example to allow intelligent mapping between objects from past to present schema. These rules are a reasonable price to pay for a properly controlled approach to data longevity and re-use, which is a key concern when applying computing to such temporal activities as construction.