Chapter 10

Mapping Controller

The development of a generic mapping controller has been a neglected area of research amongst the components of an integrated building design system. To date, the majority of integrated building design systems have employed hand-coded translators to move data from their IDM to the design tools used. This has been due partly to the lack of a language which allows a high-level specification of mappings between models, though VML, and other mapping languages, have now partly solved that problem.

Given a mapping specification between two schemas, there can be many implementations which will achieve the correct movement of data between the models. In many cases the style of implementation will be dependent upon the type of IBDS being developed. For example, an IBDS utilising many KBS's could require a very interactive connection between the IDM and each design tool. There are three levels of control that can be offered by a mapping controller for a particular mapping. These are:

Complete: the whole database is mapped every time data needs to be passed to or from the IDM, destructively overwriting the data in the data-store it is being mapped to. For this type of control no information about previous mappings needs be maintained, as the whole mapping is re-evaluated every time it is required. This is the scenario put forward in the ISO-STEP standard, where a single file representing the informational content of an AP is passed to the design tool requiring information (and the same to pass information back). While this is the simplest case to implement, it is also the most expensive computationally as the whole database must be mapped every time a design tool is invoked.

Modified: only the data changed since the previous mapping is passed to or from the IDM, where it merges with the previously mapped data. This method provides the same result as a complete mapping, but as only modified data is mapped it requires less computation. This

level of control assumes that there is a method of determining the changes between the model at the time of the previous mapping and the current state of the model. Information about previous mappings may be kept for this type of control to simplify the mapping of modifications, but is not absolutely necessary.

Interactive: an adaptation of the *modified* mapping controller where individual changes are mapped as they are recorded. As with the *modified* mapping controller, only modified data is mapped to or from the IDM to merge with the existing data. This level of control allows for very interactive design tools to be used in an IBDS, especially where the data requirements may not be ascertainable when starting the design tool (e.g., a KBS whose input is based on what it has previously seen as well as what its user is asking for). However, this level of mapping can lead to long periods of inconsistency in the models being mapped to, while information is entered by the actor using the design tool. The length of time over which actor changes are made is likely to be orders of magnitude greater than the time taken to map a set of changes as in the complete or modified scenarios above.

Though the result of a mapping using any of these three strategies would be identical, the time taken to achieve it, and the structure of the mapping controller required to implement it, will be very different. In the mapping controller described in this chapter, the second two control strategies (modified and interactive) are implemented. This is mainly due to the fact that they offer the greatest range of interactional possibilities for design tools in an IBDS. Implementation of a partial update mapping controller also allows modes of interaction for collaborative designers to be examined, as well as investigating strategies for tracking previous data mappings to reduce the computational workload in mapping data.

The remainder of this chapter details the working of a mapping controller. This starts with a consideration of what model changes need to be tracked and how this was achieved. The process of mapping is described, starting with considerations of ensuring the correct state of the data-stores to perform the mapping, and then the details of matching up objects and solving equations are defined.

10.1 Data-Store Modification Records

For a mapping controller to be able to handle partial mappings it must be able to determine the changes to a data-store between any two of its previously published states. When maintaining a mapping between two data-stores the mapping controller also needs to know when the state of one of the data-stores has changed. There are many ways that each of these requirements could be realised. In this section an implementation of persistent data-stores in Snart is detailed which provides all the information required by a mapping controller.

All versions of Snart, the implementation language of this system, have had an implementation of data-stores of varying complexity added to them (Grundy 1993). In its simplest form a data-store provides a named space inside which objects can be created. The most complex implementation of a data-store in Snart allows all objects created in a named space to be identified, and provides hooks to add data-store dependent functionality onto the creation and deletion of objects in the space.

These data stores have also been specialised to provide persistency. The persistent data-store developed for this project allows all the objects created in a space to be saved to a file, the name of which is specified when the space is created. It also allows the reloading of the persistent data-stores, with object identifier renaming if required, into the running Snart environment (see Appendix C for a complete description of object spaces and persistent stores).

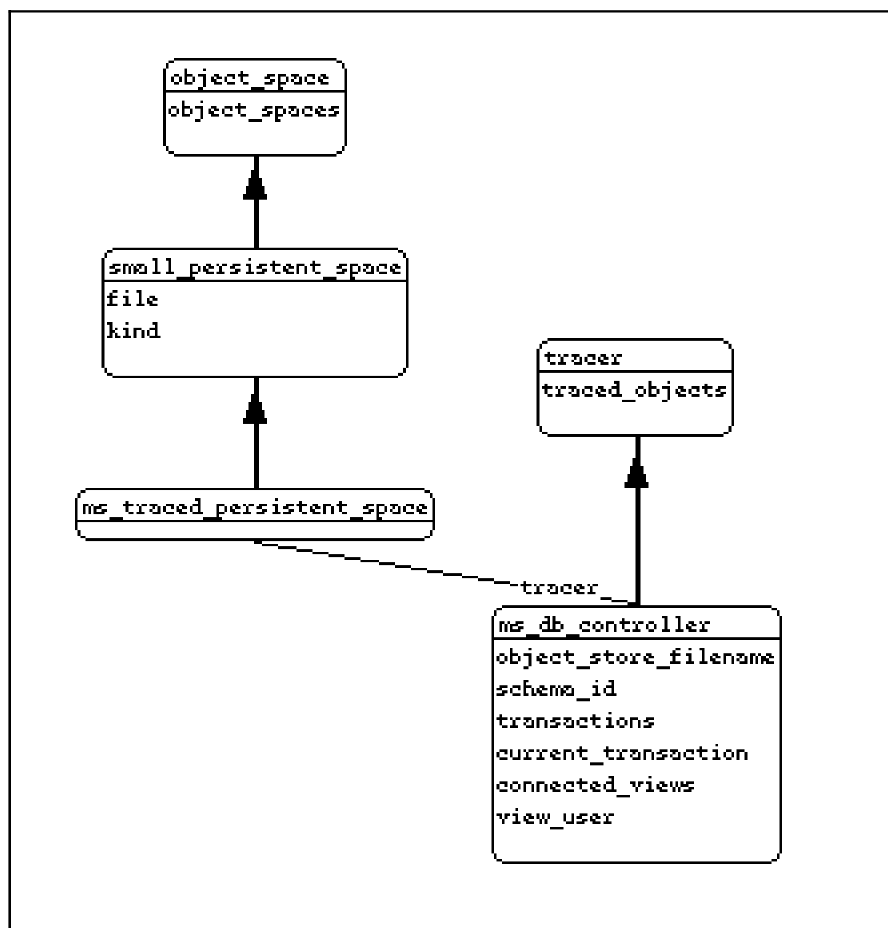

Figure 10.1 Structure of the traced persistent space in Snart

To provide the functionality required for a mapping controller, this persistent data-store allows all objects created in a data-store to be traced. This was achieved mainly by utilising the functionality of a tracer class, provided in Snart, specialised for use as a controller for the data-store (see the class hierarchies shown in Figure 10.1). Using this arrangement, all object creations are caught by the *ms_traced_persistent_space* object which ensures that they are traced by the *ms_db_controller* object for that data-store. All attribute value instantiations and method calls are caught by the

ms_db_controller object and, if necessary, recorded.

In each data-store there is a single *ms_db_controller* object which catches all events pertaining to traced objects. The *ms_db_controller* object collates modifications into a structure named a transaction, which contains all work performed for a particular task (e.g., specifying the space layout or completing a change request). The *ms_db_controller* has a notion of *working* and *finished* transactions. There may be many *working* transactions open at any time, one of which will be a default transaction in which modifications are recorded, unless specified otherwise. As each transaction collates information on a particular task in a project, having several working transactions allows an actor to manage several tasks at once, moving between them as ideas, deadlines or project managers dictate.

When the *ms_db_controller* is notified of an event that should be recorded, a new modification number is assigned (a monotonically increasing number) and a record of the event is stored in the default transaction. The raw events which are stored in a transaction are:

create_object:	creation of an object, with optional parameters
delete_object:	deletion of an object
add_value:	initial specification of a value of an attribute
change_value:	modification of a value of an attribute
delete_value:	deletion of a value of an attribute
invoked_method:	calling of a method of an object, with optional parameters
add_facet:	initial specification of a value of a facet of an attribute
change_facet:	modification of a value of a facet of an attribute
delete_facet:	deletion of a value of a facet of an attribute

The record of these events contains previous values, where these are known, so that a modification can be reversed (undone) or, at the transaction level, a whole transaction may be reversed. The transaction system also contains a notion of an aggregate transaction, which allows multiple transactions to be collated together and subsequently treated as a single transaction. These services are available to all design tools using the persistent traced space, though few have been developed to make use of them (e.g., implementing an undo/redo function inside a design tool).

When an actor has completed a transaction, a label for the transaction must be supplied. This label allows the actor to specify, in a human readable form, what work was undertaken during the course of the transaction. It is this label which will be used, initially by other actors, to deduce what work was completed in a given transaction. This label is used to form a unique identification for each transaction (See Figure 10.3 for examples of transaction labels) by appending a unique, to the data-store, transaction number (a monotonically increasing number) along with the filename of the data-store. This label is guaranteed to be unique for all data-stores on a system, and with a little padding out (e.g., written in URL form) could be unique over any number of machines.

When a transaction is complete, as signalled to the *ms_db_controller*, the raw event data is processed until there is at most one record for each object that was modified. In this form, multiple changes to a single attribute and its facets are stored as one item. Objects which were created, modified and then deleted inside the transaction have no record in the processed form. This compaction introduces constraints on the type of method that can be mapped between systems utilising a transaction-based mapping. Mapped methods are constrained to have no side-effects on other related objects. This constraint is not enforced, so the system will attempt to map any method specified in a mapping. Where methods have side-effects this could lead to failed method calls due to non-existent objects. Three types of processed events are stored:

create:	lists the modified attributes and method calls of the object which has been created
modify:	lists the modified attributes and method calls of the object which has been changed
delete:	records that the object has been deleted

Although the processing collates modifications which could be widely dispersed in the transaction, the initial ordering of object use remains constant. Therefore, the final ordering of processed modifications reflects the order in which object creation, the first modification of an object, and object deletion, were encountered.

When the processed form of a completed transaction has been calculated, the *ms_db_controller* informs all objects which have registered an interest in the data-store (usually mapping controllers) that a new transaction has been completed in the data-store. Interactive mapping controllers are also informed of each modification which occurs in the data-store.

As the description of the modification records might indicate, the number (and space requirements) of the stored modification records can build up very quickly, especially in an object-oriented application which requires much computation, and hence many method calls. To manage this storage requirement in the limited application space of the Prolog development environment a menu item allows the storage of these method call records to be turned on or off.

This data-store implementation provides the notification of modifications for each design tool and the IDM used in the IBDS. In this IBDS every design tool must interface through a data-store of this form to gain, or pass back, information from the IDM. This means that a mapped model of the IDM exists for every design tool used in the IBDS. In the examples used throughout this thesis most of the design tools are implemented in the Snart environment (i.e., PlanEntry, ThermalDesigner, FaceEditor). However, the VISION-3D design tool is a stand-alone tool which requires a data-file to be read in to supply the information for the tool to operate. Hence, the use of both independent and tightly coupled design tools is catered for using the described data-store implementation.

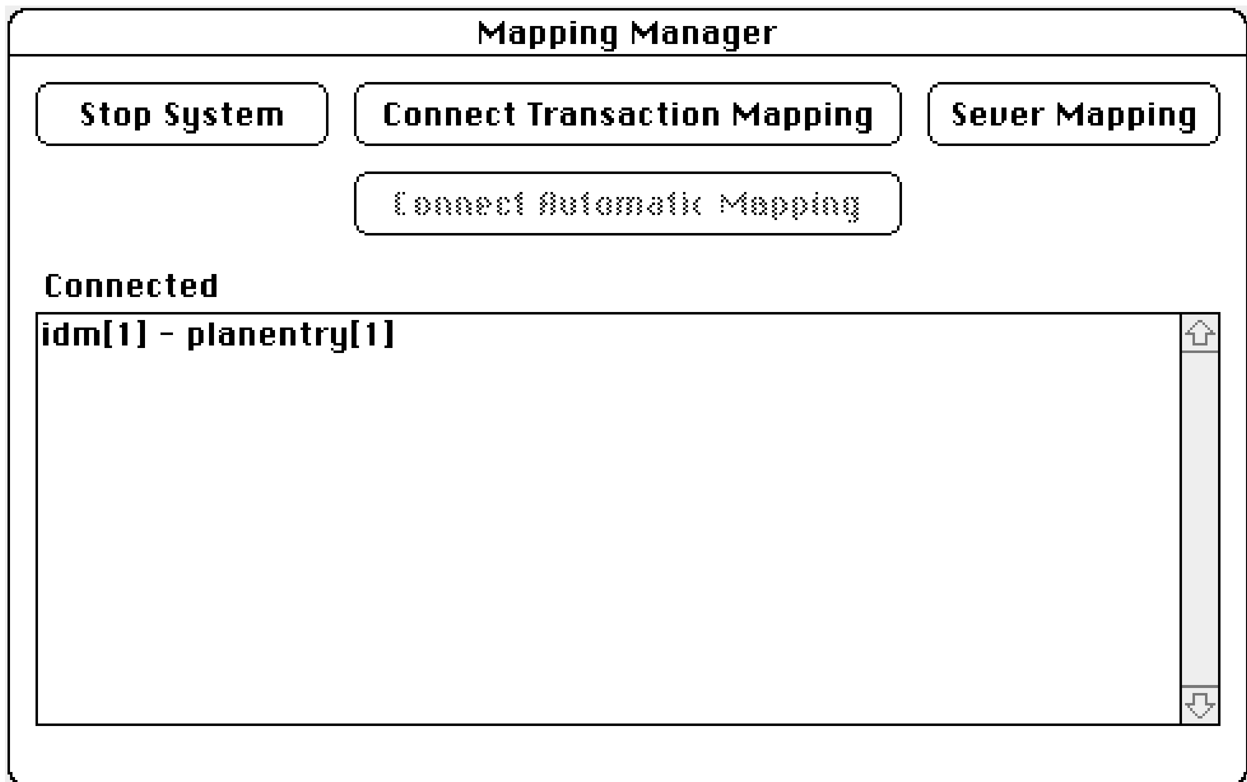

Figure 10.2 An actor's mapping controller interface

10.2 The Mapping Controller

The actor interface to the mapping system must allow the actor to manage all mappings between design tools that they require to perform their design role in a project window. To enable this each actor has a mapping controller interface as shown in Figure 10.2. This interface allows the actor to manage a design session where data is to be transferred between the IDM and a range of design tools. The range of functions available through the interface are:

Start System: initialises the mapping controller and allows the specification of a persistent data-store for the storage of information about the mappings. If an existing data-store is specified, the mapping state that was saved in it is restored and actors can continue from where they left off. If a new data-store is specified no mappings are loaded initially. When the mapping controller has been initialised the *Start System* button changes into the *Stop System* button as described below.

Stop System: allows the design session to be temporarily suspended. All data pertaining to the current set of mappings (and any which were previously connected and then severed) are saved to the persistent data-store.

Connect Transaction Mapping: starts up a transaction-based mapping between two data-stores which contain data models as specified during the creation of the mapping database as described in Chapter 6. The actor identifies the two data-stores to be used in the mapping. These can be existing data-stores or new data-stores, in which case a new file is created for the data-store. A mapping manager is created for each mapping which is connected into the

system. See Figure 10.3 for a transaction-based mapping manager interface.

Connect Automatic Mapping: starts up an automatic mapping between two data-stores which contain data models as specified during the creation of the mapping database as described in Chapter 6. The actor identifies the two data-stores to be used in the mapping. These can be existing data-stores or new data-stores, in which case a new file is created for the data-store. A mapping manager is created for each mapping which is connected into the system. See Figure 10.4 for an automatic mapping manager.

Sever Mapping: the actor can select one of the mappings which has been started up and remove it from the set of active mappings. When a mapping is severed the current mapping state is stored in the mapping controller data-store in case the mapping is ever required again.

When a mapping manager is established it informs the two data-stores it is connecting that it is interested in all transactions that are completed, and for the automatic manager all modifications as well. The data-store records the details of the interested mapping manager and adds it to its list of objects interested in its state. Several mappings may reference the same data-store and see all the changes made to that data-store, thus establishing a central IDM with multiple views for the IBDS. A collaborative environment with multiple concurrent actors is envisaged for the IBDS. However, because a Macintosh environment lacking a client-server-based object-oriented database has been utilised, the effort put into considerations of concurrency has been minimal. In the single machine environment, data-stores are locked only when a mapping is being applied to them. It is envisaged that this scheme could be used by an object-oriented database serving several actors on different machines. The only other capability the object-oriented database would require is the ability to pass a message through to the mapping manager objects on each machine which need to be informed of new transactions or modifications (a CORBA-like environment (Otte et al. 1996) would support this).

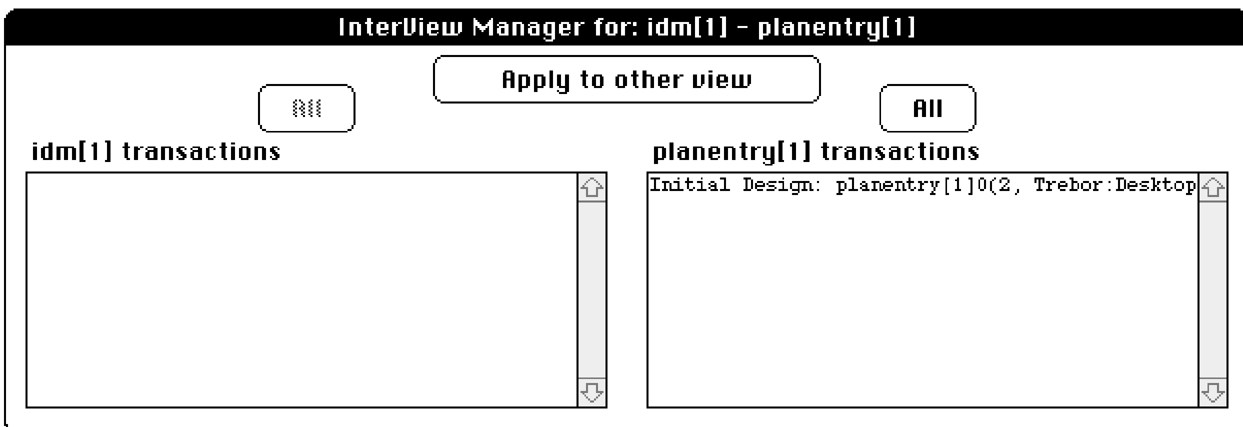

Figure 10.3 A transaction-based mapping manager

The mapping controller operates in its own persistent data-store, along with all the mapping managers that it oversees. Therefore, the set of loaded mapping managers and their state can be retained between sessions with the mapping controller. Having a persistent data-store for the mapping managers also means that all previous mappings between the two data-stores it manages

are retained and all the dynamically created indices, described in detail in later sections, can be saved between sessions. This ensures that all the work performed in determining previous mappings does not have to be duplicated when a new transaction is mapped in a later session.

10.2.1 Transaction-based mapping manager

When a transaction-based mapping manager is connected, it interrogates the two data-stores it is connecting to ascertain what transactions they have recorded in them. A difference list is constructed, and presented to the actor in the interface shown in Figure 10.3. The data-store's list of transactions can be used to ascertain which common transactions have been applied to each data-store as when a mapping is performed the full transaction name (actor label, data-store name and number) of the originator of the transaction is used to label the changes made in the opposing data-store. Using the two lists of outstanding transactions the actor determines which transactions to map between the two data-stores, when they should be mapped, and, within certain constraints, in which order they are mapped. As new transactions are completed in each data-store they are notified to the mapping manager and are added to the list of outstanding transactions for the particular data-store.

If one of the schemas in the mapping being used to maintain the connections between the data-stores is specified as *read_only* (see Chapter 5 for the different types of mapping available) none of the transactions in that data-store will be displayed in the mapping manager. When an actor selects one or more transactions to be mapped through to the other data-store, the mapping type is examined to check whether the other store is categorised as *integrated*, in which case a check is made to ensure that all outstanding transactions from the *integrated* data-store have been applied. This ensures that data-stores labelled as *integrated* are only updated by mappings from data-stores which are consistent with the *integrated* data-store's state. If all outstanding transactions from the *integrated* data-store have been applied then the mapping is allowed to proceed to the next stage of consideration, otherwise it is aborted with an appropriate message to the actor. When a set of transactions are organised to be mapped they are examined to ensure that there is no other outstanding transaction from the same data-store which must be mapped through before the selected one; for example, where a selected transaction depends upon a modification in an earlier transaction, such as the creation of an object for which an attribute has been specified. If there is no conflict the transaction mappings are started, otherwise an appropriate message is presented to the actor and the mapping aborted.

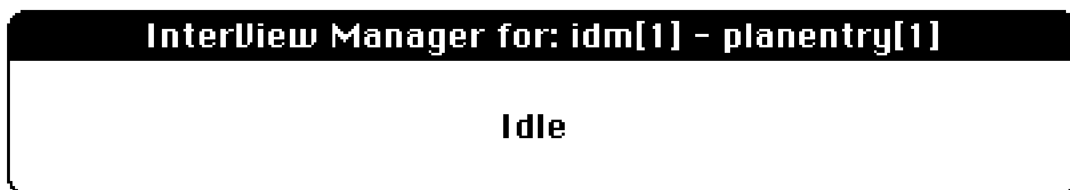

Figure 10.4 An automatic mapping manager

10.2.2 Automatic mapping manager

An automatic mapping manager can only operate between two data-stores which have had the same transactions applied to them. When an automatic mapping manager is connected, it interrogates the two data-stores it is connecting to ascertain what transactions they have recorded in them. If the two lists are different it will not allow a connection. When the conditions for an automatic mapping manager are met, an interface as in Figure 10.4 is created. This provides no functionality to the actor, but provides information about when data is being mapped between the two data-stores. As well as mapping individual modifications between the two data-stores, the automatic mapper recognises signals for a completed transaction and collates the corresponding modifications in the mapped store to become the modifications corresponding to the newly named transaction.

The mapping controller may be used to start up a mix of mapping managers, as transaction-based and automatic mapping managers readily coexist. However, the changes propagated by one type of mapping manager may be seen slightly differently by the other type of mapping manager. For example, if an IDM is connected to both transaction-based design tool models and automatically mapped design tools, all automatically mapped modifications made to the IDM will not be seen by the transaction-based design tools until an end of transaction is mapped through, at which time it will see the collated set of atomic modifications. The case of a full transaction mapped to the IDM will be seen by an automatic mapping manager as a large set of individual modifications queued for processing.

10.3 Performing a Mapping

The sections on data-stores and the mapping controller have laid the groundwork required to describe the actual mapping process. The reader should now have an idea of how modifications are gathered in a data-store and the compressed form in which they are presented to the mapping system. The reader should also understand the control options available to the actor and under what conditions they allow a mapping to proceed. In the following sections the process followed for a transaction-based mapping is described. It is almost identical to that followed for an automatic mapping.

To describe how a mapping is performed, a top down approach is adopted, looking at some general setup work, then the various cases presented by different types of objects, leading down to how individual equations are treated in the mapping system.

10.3.1 In preparation to map

Before a mapping can proceed it is necessary to ensure that the data-stores are in the state specified by the transactions being mapped (see Table 10.1 for the pseudo-code). To ensure this, all current work on the data-store and all outstanding transactions which are not being mapped must be reversed. This is possible because the persistent data-store controller tracks all transactions, both *working* and *finished*, and the transaction records all modifications (and the order in which they were performed). To bring the data-store into its correct state all *working* and unmapped transactions are rolled-back. Transactions to be mapped are checked against working and unmapped transactions to ensure that no dependencies exist between them. If the transactions to be mapped are dependant upon other unmapped data then the user in informed of the dependant transactions and the mapping is terminated. The roll-back leaves the data-store in the state it was at the completion of the specified transactions. As the data-store is locked during the mapping, rolling-back the data-store does not affect any running applications, apart from stalling them. At the end of the mapping the data-store is returned to its previous working state by rolling-forward all the rolled-back modifications.

```
performing a mapping
        determine outstanding transactions (working and unmapped)
        calculate dependencies between outstanding transactions and transactions to map
        if dependencies exist
        then
                cancel mapping and inform user
        else
                roll-back all outstanding transactions
                perform mappings
                roll-forward all rolled-back transactions
```

Table 10.1 Pseudo-code for performing a mapping

```
establish a mapping
        determine all inter_class definitions purely for the store being mapped to
        for each inter_class found above
                if object of specified class exists matching the invariants
                then
                        do nothing
                else
                        create object of named type
                        solve all initialisers
```

Table 10.2 Pseudo-code for establishing a mapping

10.3.2 The first mapping between two stores

The first time a mapping is performed to a particular data-store, some objects may need to be created in the data-store (see Table 10.2 for the pseudo-code). In a VML mapping this is specified by an *inter_class* which only lists classes for one of the schemas being mapped between. If this is the first known mapping to the particular data-store, the mapping system searches the mapping database for any *inter_class* definitions of this form. If any are found the data-store is searched for

objects of the class specified. If none are found new objects of the specified type are created and the equations, usually initialisers, calculated as defined later in this chapter.

```
perform mappings
        for each aggregated modification to be mapped
                case aggregated mapping of
                create:
                        find inter_class definitions referencing class of created object
                        for each inter_class identified
                                determine initial object groups
                                generate object combinations for the inter_class
                                for each remaining combination
                                        if created object is grouped in inter_class header
                                        then
                                                match with existing mappings of this type
                                                re-evaluate grouped objects
                                                re-evaluate invariants if object's class involved
                                                if invariants are violated
                                                then
                                                        dissolve existing mapping
                                                else
                                                        re-calculate affected equivalences
                                        else
                                                create required objects in other data store
                                                apply initialisers
                                                solve equivalences
                modify:
                        { see Table 10.11 }
                delete:
                        determine all existing affected mappings
                        for each affected mapping
                        if object's class is grouped in inter_class header
                        then
                                re-evaluate grouped objects
                                re-evaluate invariants if object's class involved
                                if invariants are violated
                                then
                                        dissolve existing mapping
                                else
                                        re-calculate affected equivalences
                        else
                                { see Table 10.10 }
                        find all other inter_class definitions with object's class grouped
                        for each inter_class identified
                                { perform mapping as for create above }
```

Table 10.3 Pseudo-code for performing mappings

10.3.3 Consideration of modification types

The three classified forms of object modification obtained from the data-store (create, modify and delete) are handled separately in the mapping system, though the create and modify forms are very similar. A general outline of what happens for each of these types of object modification is presented initially and then more detail will be presented on important aspects of their handling (see Table 10.3 for the pseudo-code).

create: all *inter_class* definitions which reference the class of the new object are identified. All combinations of the classes in the header of the *inter_class* specifications are created from objects in the data-store being mapped from, but using the created object as a placeholder for the class in the header. Then all the invariants pertaining to that side of the mapping are applied to the combinations to determine if any of them are applicable. If any pass the invariants then all the initialisers and equations are solved as described later in this section.

modify: all existing mappings which use the modified object are notified of a change to the object and must re-calculate any changed equations. If the modification affects invariants of *inter_class* definitions then they must be re-evaluated as for a newly created object, i.e., it may be possible to create a mapping with the new values, or it may be necessary to dissolve a mapping as for delete below.

delete: all mappings referencing the deleted object are notified and must re-calculate their equations. If the object was part of the header of an *inter_class* then the mapping is dissolved and any objects created in the mapped to data-store due to the now deleted object are themselves deleted.

10.3.4 Determining combinations of objects from an *inter_class* header

When instigating a mapping between two data-stores, or propagating changes between data-stores, it is necessary to determine which objects are going to be used with each *inter_class* definition. The first step towards this is determining the combinations of objects to be tested against the invariants for each *inter_class* definition (see Table 10.5 for the pseudo-code). This is calculated by examining each class in the header of an *inter_class* and returning a list of possibly matching objects. The only exception is for the class of the newly created, or modified, object where the returned list contains just the newly created, or modified, object (unless it is denoted as grouped). If the class of the newly created, or modified, object is in the header as grouped then the returned list is a list of a list of all objects matching the class defined. Table 10.4 shows some combinations of headers and the resulting lists of objects returned. CK is the class of the known (newly created or modified) object, C1 and C2 are arbitrary classes. The resultant object list is either a single object or a number of objects (shown as 1..n, or m or p), the *O* denotes an object and is subscripted with the class type from the *inter_class* header.

As the combinations of all objects in these lists grows exponentially based on the number of objects in each list, a pruning algorithm is employed to help minimise the number of combinations produced (see Table 10.6 for the pseudo-code). Note that in the worst case, where there are no invariants, the number of combinations will equal the cross-product of all objects in each list. The pruning algorithm about to be presented is not optimal computationally (the author is aware of optimal methods from relational database work, Ullman 1982), but is used to prototype the mapping manager implementation and is a workable tradeoff between programming time available

(minimal), program intricacy (simple), and efficiency (moderate).

inter_class header	Resultant object lists
[CK]	[[O_{CK}]]
[CK, C1]	[[O_{CK}], [$O1_{C1}$..On_{C1}]]
[CK, C1, C2]	[[O_{CK}], [$O1_{C1}$..On_{C1}], [$O1_{C2}$..Om_{C2}]]
[C1, CK]	[[$O1_{C1}$..On_{C1}], [O_{CK}]]
[CK, group(C1)]	[[O_{CK}], [[$O1_{C1}$..On_{C1}]]]
[CK, group(C1), C2]	[[O_{CK}], [[$O1_{C1}$..On_{C1}]], [$O1_{C2}$..Om_{C2}]]
[group(CK)]	[[[$O1_{CK}$..Op_{CK}]]]
[C1, group(CK)]	[[$O1_{C1}$..On_{C1}], [[$O1_{CK}$..Op_{CK}]]]

Table 10.4 Examples of *inter_class* headers and resultant object lists

```
determine initial object groups
        for each class in inter_class header
                if class is grouped
                then
                        find all objects of named type
                        return as a list of a list
                else if class is of key object
                        then
                                return the key object in a list
                        else
                                find all objects of the named type, return in a list
```

Table 10.5 Pseudo-code for determining initial object groups

To prune the number of object combinations at each stage the following algorithm is used. Starting with the last list of objects, the list is reduced by applying all invariants which apply purely to those objects. The next set of objects is similarly reduced, then the cross-product of the two sets is obtained. This combination is reduced by applying any invariants which pertain just to the two classes in the cross-product. Then the next set of objects is reduced by applying all invariants which apply purely to those objects. Then the cross-product of the reduced set and the previous cross-product is obtained. This combination is reduced by applying any invariants which pertain just to the classes in the new cross-product. This is repeated until the first set of objects has been incorporated into the cumulative cross-product. The result is a list of lists of objects which match all the invariants in the *inter_class* definition. Each of these lists can be used to create a mapping to the other data-store. In the worst case, where there are no invariants, the number of combinations is equal to the cross-product of all objects of all classes. However, assuming this is a valid specification of a mapping then it must be handled by the mapping system.

generate object combinations for the *inter_class*
initialise current result set to empty
for each set of objects relating to a class in the *inter_class* header
determine invariants relating purely to this class
apply selected invariants to the set
create cross-product of reduced set with current result set
determine invariants applying to classes incorporated in current result set
apply selected invariants to the result set

Table 10.6 Pseudo-code for generating object combinations for an *inter_class*

The way in which an invariant is applied to objects of a class varies, dependent upon whether the class is grouped or not. If a class is not grouped, and an object fails an invariant, the whole combination is removed from the list of combinations to be further considered. If a class is grouped, then the invariant is applied to each object in the group and is used to reduce the number of objects in the group. The application of this algorithm is demonstrated in the following example based upon an *inter_class* definition for mapping between the IDM and PlanEntry. In this example, a mapping from PlanEntry to the IDM is being undertaken. The *pe_face* object is the newly created object, and there are six *pf_plane_object* objects and five *pe_opening* objects. The values of selected attributes and functions of the example objects are shown below the header and invariants section of the *inter_class* definition. Note that in the values column for objects of class *pf_plane_object* the result of calling the function *map_orientation_axis* is shown, rather than an attribute value, and for the objects of class *pe_opening* the result of calling the function *contained_in_face* is shown.

```
inter_class([idm_space_face], [pe_face, pf_plane_object, group(pe_opening)],
        invariants(
                type_of_face \= 'opening',
                member(pe_face.orientation, ['n', 's']),
                pe_face.offset = pf_plane_object.offset,
                map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
                contained_in_face(pe_face, pe_opening),
                pf_plane_object.axis \= 'z'
        ),
……).
```

ObjID	Type	Attributes / Functions	Values / Results
o1	pe_face	[orientation, offset]	['n', 12.5]
o2	pf_plane_object	[axis, offset, map_orientation_axis('n', 'x')]	['x', -7, false]
o3	pf_plane_object	[axis, offset, map_orientation_axis('n', 'x')]	['x', 12.5, false]
o4	pf_plane_object	[axis, offset, map_orientation_axis('n', 'y')]	['y', -11, true]
o5	pf_plane_object	[axis, offset, map_orientation_axis('n', 'y')]	['y', 12.5, true]
o6	pe_opening	[contained_in_face(o1, o6)]	[true]
o7	pe_opening	[contained_in_face(o1, o7)]	[false]
o8	pe_opening	[contained_in_face(o1, o8)]	[false]
o9	pe_opening	[contained_in_face(o1, o9)]	[true]
o10	pe_opening	[contained_in_face(o1, o10)]	[false]
o11	pf_plane_object	[axis, offset, map_orientation_axis('n', 'z')]	['z', 0, false]
o12	pf_plane_object	[axis, offset, map_orientation_axis('n', 'z')]	['z', 2.3, false]

The initial list of list of objects for the header [pe_face, pf_plane_object, group(pe_opening)] is [[o1], [o2, o3, o4, o5, o11, o12], [[o6, o7, o8, o9, o10]]]. The first step is to try to reduce the grouped list [[o6, o7, o8, o9, o10]]. However, there are no invariants which apply purely to these objects, so the next list is selected, [o2, o3, o4, o5, o11, o12] and it is also reduced. In this case the invariant checking that the axis is not *z* can be applied, producing the list [o2, o3, o4, o5]. The two resultant lists are combined to create the following cross-product:

[[o2, [o6, o7, o8, o9, o10]],
 [o3, [o6, o7, o8, o9, o10]],
 [o4, [o6, o7, o8, o9, o10]],
 [o5, [o6, o7, o8, o9, o10]]]

There are no invariants affecting only *pe_opening* and *pf_plane_object* items so this initial cross-product can not be further reduced at this time. Then the next list is selected, [o1]. There is one invariant which applies purely to this object (member(pe_face.orientation, ['n', 's'])), this is checked against the object in the list to produce the same list, [o1]. This list is then combined with the other lists to create the cross-product below:

[[o1, o2, [o6, o7, o8, o9, o10]],
 [o1, o3, [o6, o7, o8, o9, o10]],
 [o1, o4, [o6, o7, o8, o9, o10]],
 [o1, o5, [o6, o7, o8, o9, o10]]]

Now the remaining three invariants on the PlanEntry side can be applied to each combination. The first list, [o1, o2, [o6, o7, o8, o9, o10]], is rejected as ‘pe_face.offset = pf_plane_object.offset’ is false, the second list, [o1, o3, [o6, o7, o8, o9, o10]], passes that invariant but fails the ‘map_orientation_axis(pe_face.orientation, pf_plane_object.axis)’ invariant. The third list, [o1, o4, [o6, o7, o8, o9, o10]], is rejected as ‘pe_face.offset = pf_plane_object.offset’ is false, the fourth list, [o1, o5, [o6, o7, o8, o9, o10]], passes that invariant and the ‘map_orientation_axis(pe_face.orientation, pf_plane_object.axis)’ invariant. The last invariant,

‘contained_in_face(pe_face, pe_opening)’ is used to reduce the objects in [o6, o7, o8, o9, o10] to the final list of [o6, o9]. Therefore the reduced list of lists from the given set of objects being applied to the invariants of this *inter_class* is [[o1, o5, [o6, o9]]], meaning that one mapping is performed across to the IDM for the newly created object o1.

10.3.5 Four pass mapping process

In recognition of the fact that the order in which objects are created in one data-store may not match the order objects should be created in another data-store, the mapping process is performed in four passes (see Table 10.7 for the pseudo-code). Using a four pass system obviates problems that may occur in transaction-based mapping due to the collating of all object modifications to one point, rather than the actual order they occurred. The work performed in each pass is detailed below:

First pass: determine all combinations to be mapped from the initial data-store. Where necessary, create new objects in the data-store being mapped to as described in Section 10.3.6. At this point not all objects, which could be connected together to match classes in the *inter_class* header, are created, so only specify values for the attributes of the newly created object. At the end of pass one it is guaranteed that the minimum state of the mapping is that all combinations to be mapped have been identified, and the object matching the first class of the *inter_class* header for the data store being mapped to has been identified. It is also guaranteed that all values that are directly identifiable for that first object have been calculated.

Second pass: determine all objects matching the header class specification. In mappings where there is more than one class specified in the header of the *inter_class* for the data-store being mapped to, try to connect to existing objects for the other classes rather than create new objects. At the end of pass two it is guaranteed that all objects matching the *inter_class* header for the data store being mapped to have been identified or created. It is also guaranteed that all values that are directly identifiable for these objects (i.e., specified in an *inter_class* where the object’s class is specified in the header) have been calculated, this includes initialisers, equivalences and invariants.

Third pass: attempt to solve equivalences for each affected object. Some equations with values that are accessed down a pointer chain may not be solvable at this time, as the values for the referenced object may not have been calculated. At the end of pass three it is guaranteed that all indirectly referenced objects (i.e., those specified through a pointer chain in equivalences) have either been associated to the *inter_class* (where they existed already), or created and initialisers solved. Due to the lack of control over the order that these indirectly referenced objects get created, and their initial values specified, there is a requirement for a further pass to solve outstanding equivalences. For example, the following equation o_{1l}=>o_{2l}=>o_{3l}=>a_{1l} = o_{1r}=>o_{2r}=>o_{3r}=>a_{1r} where each of the object references is solvable by another mapping (e.g., if mapping from left to right, the object ID of o_{1r} is determined

from the mapping of o_{11}) is not solvable until all associations have been completed. This is because the attribute references are not able to be determined until phase two is complete, and depending upon the order of solving mappings this equation might be attempted before all other attribute references have been matched.

Fourth pass: reassess all previously unsolvable equations involving referenced values to determine whether they are now calculable. At the end of pass four it is guaranteed that all equivalences which can be solved from the data available in the data store being mapped from, through the *inter_class* definitions, have been calculated.

```
four-pass inter_class resolution
        case pass number of
        one:
                determine all combinations to be mapped from data store
                create objects for first class in inter_class header specification
                apply appropriate initialisers, equivalences and invariants
        two:
                attach or create objects for all other classes in inter_class header
                apply appropriate initialisers, equivalences and invariants
        three:
                for each mapping combination apply all equivalences
        four:
                determine unsolved equivalences affected by pass three resolutions
                re-calculate determined equivalences
```

Table 10.7 Pseudo-code for the four-pass *inter_class* resolution

10.3.6 Mapping a new combination to the other data-store

When preparing to map a combination of objects it is checked against object combinations that have already been mapped to the other data-store (see Table 10.8 for the pseudo-code). If the combination of objects, except for those in grouped classes, is unique, then a new mapping is initiated. If the combination matches, except perhaps for objects in grouped classes, then all affected equations of the existing mapping are re-evaluated.

When a new mapping is created, an object whose class is that of the first class in the header for the side being mapped to is created. In the example above, an object of class *idm_space_face* would be created. Then all initialisers, equivalences, and invariants which determine values for the new object are solved. If the *inter_class* header for the side being mapped to contains more than one class, then a combination determining procedure, as outlined in Section 10.3.4, is followed in the data-store that is being mapped to during the second pass through the mappings. This provides a way of linking up with existing objects in the data-store being mapped to, rather than creating superfluous objects during the mapping. If a matching combination is found, then all equivalences which apply to the found objects are solved. If no matching combination is found, then a new object is created for the next class along in the header, all initialisers, equivalences, and invariants which determine values for the new object are solved and the matching process is retried. This process is duplicated until either a matching combination of existing objects is found for the rest of

the header classes, or until new objects have been created for every class defined in the header of the *inter_class*.

```
mapping all combinations of objects
    for each combination of objects for an inter_class header
        if combination matches an existing mapping combination (except grouped)
        then
            re-calculate affected equivalences
        else
            create object for first class in inter_class of side being mapped to
            apply initialisers, equivalences and invariants purely for this object
            for each following class in inter_class header (in pass 2)
                search for matching objects
                if no objects found
                then
                    create object for class no object could be found for
                    apply initialisers, equivalences and invariants
            solve all equivalences and invariants for the mapping (pass 3 and 4)
```

Table 10.8 Pseudo-code for mapping a new combination

10.3.7 Procedures followed when a new object is created

Whenever a new object is created during a mapping, the initialisers section of the *inter_class* definition being used is examined and any initialisers which apply to objects of that class are solved, setting values for attributes, or calling methods of the newly created object (see Table 10.9 for the pseudo-code).

Whenever an object is created in a mapping it is cross-referenced against the mapping in which it was created. In this manner the list of objects created in each individual mapping can be ascertained, and this information used when objects need to be deleted.

```
a new object has been created
    if an object clash was reported
    then
        find existing object which matches known criteria
        delete newly created object (if not already discarded by system)
        utilise the found object's ID as the new object ID
    else
        find all initialisers applying to the object
        for each identified initialiser
            resolve initialiser
        cross-reference object creation against this mapping
```

Table 10.9 Pseudo-code for creating a new object

As the mapping manager tries to distance itself from the underlying implementation of the data-stores, it does not duplicate the functionality which would be provided by e.g., relational databases. To this extent, determining object uniqueness (i.e., key violations in a relational database) is not handled explicitly by the mapping manager. Instead, if an object is created which conflicts with existing objects, the mapping manager expects to be informed of this fact by the

underlying data-store. If such a clash is detected, the mapping manager searches the existing data-store to find the previous object. The object whose newly asserted value caused the clash is deleted and the existing object used for all further references. In this manner the mapping manager can reuse existing data-store objects, rather than creating multiple instances of objects containing the same data. However, this is not demonstrated here as the Snart persistent data-store used in the demonstrations of the mapping manager does not yet have an implemented key uniqueness verification system for objects created inside it. This remains for future work.

10.3.8 Tracking objects created and referenced in mappings

The fact that an object is referenced in a mapping is also tracked. This tracking encompasses more than just the objects collated together to match the header classes of the *inter_class*, but includes all objects referenced through the equations in the *inter_class*. Objects that can be included in this manner are those that are used to follow reference chains. In this way a list of mapping managers is compiled for every object used in a mapping. Then when an object is modified the set of mapping managers which may have to perform recalculations is quickly identified. These mapping managers are informed of the modified object and affected attributes, and using the pre-computed lists of objects and attributes in each equation determine which equations have to be re-computed.

```
an object was deleted requiring a mapping to be dissolved
    delete primary object from first class of header in the inter_class
    determine all other objects matching the header
    for each object
        if this mapping is only reference to the object
        then
            delete object
        else
            update mapping references to the object, deleting dissolved mapping
    determine all objects created when solving equivalences and initialisers
    for each object
        if this mapping is only reference to the object
        then
            delete object
        else
            update mapping references to the object, deleting dissolved mapping
```

Table 10.10 Pseudo-code for mapping an object deletion

10.3.9 Mapping the deletion of an object

The mapping of an object deletion in one data-store is handled using a type of garbage collection scheme (see Table 10.10 for the pseudo-code). If the deletion of an object causes a mapping to become non-viable (e.g., not enough objects for the classes in the header, or an invariant is violated) then all objects which were necessarily created when setting up that mapping (e.g., the object for the first class in the *inter_class* definition) are deleted. All the other objects which were created during the mapping are examined. If their only point of reference is the mapping which is being dissolved then they are deleted as well. Also, all objects referenced by equations in the

mapping are examined. If their only point of reference is the mapping which is being dissolved then they are deleted as well.

10.3.10 Mapping the modification of an object

Objects whose attributes have been modified use the lookup table (an AVL tree structure) to determine which mapping controllers need to be informed (see Table 10.11 for the pseudo-code). When the mapping controllers learn of the modification they initially check whether the modifications affect any of the invariants of the *inter_class*. If invariants are affected then they are re-evaluated to ensure that they still hold. If the invariants are violated the mapping is dissolved with deletions of objects as defined in Section 10.3.9. If the invariants are not affected, or if they still hold, then all affected equations are identified and re-calculated.

```
an object has been modified
    determine affected mappings
    for each affected mapping
        if modification affects an invariant specification
        then
            re-evaluate object combination
            if new combination results (except for grouped classes)
            then
                dissolve existing mapping
                create required objects in other data store
                apply initialisers
                solve equivalences
            else
                re-calculate affected equivalences
        else
            re-calculate all affected equivalences
    find inter_class definitions referencing class of modified object and also in invariants
    for each inter_class identified
        determine initial object groups
        generate object combinations for the inter_class
        for each remaining combination
            if combination matches existing mapping, or class is grouped in header
            then
                do nothing as will have been solved above
            else
                create required objects in other data store
                apply initialisers
                solve equivalences
```

Table 10.11 Pseudo-code for mapping an object modification

All *inter_class* definitions which reference the modified object in their class headers are also identified. They are checked to see if the modified attributes affect any of the invariants of these *inter_class* definitions. If they do, then the object combinations are computed as in Section 10.3.4 and any new combinations are used to initiate new mappings as detailed in Section 10.3.6.

10.3.11 Evaluating, or re-evaluating affected equations

All equivalences in an *inter_class* are passed through to the equation solver, along with all attributes affected by the object creation or modification. Prepended to all the equivalences are the invariants which apply to objects in the data-store being mapped to (i.e., what has to hold for the objects that are being created or modified). As each equation is pre-processed and all objects and attributes used in the equation identified, the first step in solving an equation is to identify values for all necessary object and attribute references in the equation (see Table 10.12 for the pseudo-code).

The result of this search for values is used to determine whether the equation can be solved currently or not. When performing a mapping, the controller is not allowed to make changes to the data-store the mapping is coming from. The assumption is that the transaction is not modifiable as it may have already been mapped to other data-stores. Therefore, any equation with unknowns on the side being mapped from is classed as unsolvable. Given that all values are known for the data-store being mapped from, the three different types of equation are handled slightly differently:

procedure: all values from the data-store being mapped from are matched to the procedure parameters. If there are any object references for the store-being mapped to, then either the existing object reference is supplied, or a new object is created to be passed in. If the parameter *$mapping_system$* is specified then the object ID of the mapping system is passed as a parameter. Any attribute unknowns are recorded and the procedure invoked. When it terminates, any of the unknowns which were calculated during the procedures calculations, and passed out in the parameters, are assigned to the specified attributes.

function: all values from the data-store being mapped from are matched to the function parameters. If there are any object references for the store being mapped to, then either the existing object reference is supplied, or a new object is created to be passed in (on the assumption that a function will not know how to create an object in a data-store). Any attribute unknowns are recorded and the function invoked. When it terminates, any of the unknowns which were calculated during the function's calculations, and passed out in the parameters, are assigned to the specified attributes.

equation: the mapping system contains an equation rearranger which can take most equations and rearrange to solve for a particular attribute. Therefore it is only necessary to identify the attribute that needs to be solved in the equation. If there is one unknown in the set of attributes of the data-store being mapped to, then this is the unknown attribute for which the equation is solved. If there are several unknown attributes, then the equation can not be solved at the current time, though it may be solvable after other equations have been solved (i.e., some of the unknowns are calculated by other equations), and is placed in a queue to be re-examined. If there are no unknowns, an alternate approach must be taken. Part of the meta-data recorded for each attribute, by way of facets, is a facet determining who supplied the data for a particular attribute. Where all attributes are known, this facet information is used to provide a ranking of the attributes in terms of the importance of where their data

was asserted, from default data through to actor defined data. This ranking is applied to try to identify the least significant piece of data known. The equation is then solved for this attribute. If no attribute can be identified to be solved then the equation is placed on a queue to be examined when the other equations have been solved, to see if they provide any further assistance in solving the difficult equation. After all equations have been attempted, the queue of unsolvable equations is re-examined, checking whether anything has changed which makes these equations solvable. This queue is iterated over until there is no change in the status of the equations in the queue, i.e., nothing was solved in a particular iteration. Any equations which can not be solved, after all the procedures outlined above, are rewritten as constraints and asserted against the particular object involved. In this way the values that should hold for a particular object, or set of objects, are recorded and are seen in the data-store even after the mapping has completed. Asserted constraints are specified in the constraint specification language implemented in Snart (Mugridge et al. 1995). The Snart constraint specification language has a similar syntax to the VML notation, enabling VML equations to be easily rewritten and asserted in Snart.

```
identify values for an equation
    for each reference in the equation to data-store being mapped from
        find value for the reference
    if any missing references in the equation to data-store being mapped from
    then
        terminate solving of equation
    else
        for each reference in the equation to data-store being mapped to
            if value is known for the reference
            then
                return known value
            else
                case equation type of
                procedure:
                    case reference type of
                    object:
                        { see Table 10.9 }
                    $mapping_system$:
                        return object ID of mapping system
                    attribute:
                        record as unknown
                function:
                    case reference type of
                    object:
                        { see Table 10.9 }
                    attribute:
                        record as unknown
                equation:
                    record as unknown
```

Table 10.12 Pseudo-code for identifying values for an equation

The ranking of importance of values assigned to an attribute is managed by the mapping system. The mapping system ensures that no low ranked values are able to overwrite high ranked attributes (e.g., an attribute defined through a default value being mapped over an attribute previously

defined by an actor). In most cases, values at the same ranking may overwrite each other if specified later in time (e.g., a design tool calculated value can overwrite a previous design tool calculated value). This is not automatically true for one case, that of actors. In this case, a dialogue is initiated to ensure that the new actor-specified data is allowed to overwrite the previous actor-specified data. If the overwriting is allowed, then that permission is maintained for the whole mapping of the transaction, before being rescinded again. The full set of update authorities is shown in Table 10.13.

New / Previous	User	Design Tool	Constraint	Default	Unknown
User	negotiate	no	no	no	no
Design Tool	yes	yes	yes	no	no
Constraint	yes	no	merge	no	no
Default	yes	yes	yes	yes	no
Unknown	yes	yes	yes	yes	yes

Table 10.13 Update authorities for attributes derived from different sources

When an equation is solved and a value instantiated for an attribute, it is necessary to instantiate the facet recording who specified the calculated value. In some cases this is easily calculated. Initialisers and invariants which specify a constant value for an attribute allow the value to be categorised as default. Equations which equate single attributes or pointers can be categorised utilising the same categorisation as the attribute from which the value or reference is mapped. Equations which involve several attribute values which may come from very different sources (e.g., default values and actor-specified) require a more sophisticated method of determination. In the mapping system the highest level specifier is used for complex equations. For example, in an equation involving default values, design tool calculated values, and actor-specified values the final attribute value will be defined as coming from an actor. Using this method, the values in a mapped system tend towards being actor-specified, seemingly imparting a high level of confidence in the data in the system. However, this also means that an actor must intervene more often during the mapping process to assert the right to overwrite values specified by other actors (assuming they are within the actor's schema specifying modification permission).

10.4 Appraisal of Mapping Controller

The mapping system described in this chapter provides an implementation of the VML language capable of both transaction-based and interactive mappings between data-stores. The implemented

system allows multiple data-stores to be connected to an IDM, with full maintenance of the consistency of the IDM at all times, as well as reflecting changes made in any connected data-store through to all affected data-stores. The mapping system has been tested with a wide set of mappings, including many small mappings drawn from other work in the area (see Appendix D and Clark 1992; Bailey 1994; Hardwick 1994), and a large example showing the integration of four design tools connected through an IDM (see Appendix E; Hosking et al. 1995; Mugridge et al. 1996). The large example described in Appendix E shows the use of the mapping system to maintain information updates and consistency back and forth between the interactive tools being integrated, as well as being output to a visualisation tool. The use of the different tools is controlled by a process model for the task being attempted, which establishes when actors can perform their design functions, and manages the mappings which are allowed. This example in Appendix E provides the main demonstration of how all the components of the thesis work together to create an integrated design system.

Although the mapping system is capable of maintaining the correspondences between two data-stores, it is not capable of linking together two independently developed data-stores (e.g., the same building specified independently in two different design tools). This is mainly due to the assumption that a new object is created in the data-store being mapped to for every *inter_class* mapping which is initiated with objects from the other data-store. The class matching algorithm could be extended to attempt a match of the initial class rather than creating one, but this may not provide the required solution in all cases. The problem is partially that there is no syntax to specify whether an object should be created or searched for in a mapping, and partially that there are cases when it is not possible to decide whether an object should be created or searched for. Part of the problem comes from object-oriented systems where there are not necessarily keys defining which objects must be unique. Instead it is assumed that the object-oriented system ensures the consistency of the model by the way in which it is developed (semantics of the application). These semantics may not be able to be specified in a mapping without implementing a large amount of the object-oriented application within the mapping.

The mapping system requires more work in the section which decides which attribute of an equation to solve for. The current method works well where there are small numbers of attributes in an equation, or where there is a previously asserted value which is a default or other low level of specification. However, where several attributes were all previously asserted at the same level there is no good way of deciding which to solve for. The addition of reasons, as implemented in the Snart language (Hosking et al. 1994), may provide a way of deciding which attribute to solve for depending upon which attribute, or attributes, changed in the object being mapped from. This could, however, make the equation definition much larger and more difficult to maintain, though the use of default directions would alleviate this problem. The real problem is dealing with an actor's intent in specifying information. If this intent could be captured, perhaps partly through the process specification, then it would be easier to determine what data could change. For example,

information derived from trialling a design variation should not take precedence over previous information asserted for a legally binding project sign-off stage. Capturing the level of intent would extend Table 10.13 to include classifications such as actor assumptions, derived information, informed specification, and constraints due to standards, codes of practice, best practice, etc.

Several speed improvements could be made to this system. Mapping of small building models (a building consisting of two spaces with doors and windows) between an application and the IDM takes twenty minutes, or more, on the Macintosh on which this was developed. Two sections of the mapping system are known to be inefficient.

- The persistent data-store which traces and records all modifications and method calls adds a very large burden on all applications in which it is utilised. This is especially true for the tracking of method calls as the calls in the running mapping system (written in Snart) are trapped as well as those in the running design tools (also written in Snart). However, the method calls of the running mapping system are discarded immediately. Rewriting the data and method call tracing in a lower level language would provide large speed improvements to design tools and the mapping system.
- The generation of combinations matching *inter_class* headers is very slow when large numbers of objects are involved. Two improvements are considered for this problem.
  - The first is to implement the more efficient combination generation algorithm from relational database work (Ullman 1982). Though the worst case combination generation is a polynomial of the same order as the number of classes referenced in a header (as in a relational database system), this is unlikely to ever be a practical problem. The reasons for this are that usually only a very small number of classes are grouped through a header, and where a larger number of classes are referenced it is usually to associate information from a single uniquely identifiable object. The relational database algorithms can ensure that the most efficient combinations are computed first and the resultant sets are reduced as soon as possible.
  - The second is to ensure that combinations are only generated once for any set of new, modified, or deleted objects. In the current system this is not the case and each new, modified, or deleted object forces the generation of its own set of combinations, before checking that the generated combinations are not already being used.

Other speed improvements are possible from the use of a low-level programming language like C rather than the LPA Prolog used in this thesis. Trials with a rewritten Snart have shown two orders of magnitude speed improvement for some operations (in the constraint processing). However, Prolog was the correct choice for prototype implementation for this PhD thesis in that more functionality was programmed than would have been possible in the same time with other languages.