A DECLARATIVE APPROACH TO INTER-SCHEMA MAPPINGS

by
Robert Amor, John Hosking, Warwick Mugridge*

ABSTRACT

The requirements for the specification of mappings between tools in an integrated and interactive design system are described in this paper. The declarative mapping language, VML, is introduced. VML allows a high level, bidirectional specification of mappings between two arbitrary schemas. To illustrate the utility of VML a demonstration system consisting of plan definition and code conformance tools is integrated via a common data model using VML mappings. The VML mappings are capable of handling both relational and object-oriented style schemas as well as interactive and batch style design tools. We illustrate the use of VML to specify correspondences between classes, conditional application of correspondences, different styles of equivalence, initialisation conditions as well as method handling for interactive systems.

INTRODUCTION

Our group has had a long term interest in the computerised interpretation of codes of practice. However, to be useful, such applications need to be integrated both together and with other tools, such as for editing plan information. In recent work we have defined a common schema for two applications concerned with checking compliance with codes of practice, as a first step in an evolutionary approach to a more complete common schema (Mugridge and Hosking, 1995). This work highlighted the need for a means to define and implement mappings between applications and the common schema. The mapping implementation needs to maintain or manage the consistency between the multiple instances of the schemas involved. In this paper, we describe VML (View Mapping Language), a declarative language for defining inter-schema mappings, and its application in a demonstration integrated building design system.

A DEMONSTRATION SYSTEM

To act as a focus for our work, we have been developing a demonstration system integrating two of our previously developed code of practice conformance tools, together with a plan drawing system, a materials editor, and a 3-D visualisation tool. The architecture of the system is shown in Fig. 1. ThermalDesigner (Amor et al, 1992) checks building designs against the insulation requirements of a draft New Zealand insulation code. WallBrace (Mugridge and Hosking, 1988) checks building

* Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand e-mail:{trebor, john, rick}@cs.auckland.ac.nz

designs against the wall loadings requirements of a New Zealand code for loadings in light timber framed buildings. These two tools were used to define the initial Integrated Data Model (IDM) as described in (Mugridge and Hosking, 1995). PlanEntry is a novel constraint-based plan drawing system (Hosking et al, 1994). The materials editor is a constraint-based tool for specifying materials and bracing to cover wall faces, and VISION-3D is a tool for visualising and manipulating 3-D models (Bourke, 1989). The IDM schema is developed in EXPRESS, using the EPE environment (Amor et al, 1995a), which supports translation to an object-oriented implementation for use in model instantiation.

The mappings between the various tools and the IDM are specified using VML. In practice the mapping is implemented in two parts. An EXPRESS schema defining the data requirements of the tool is first specified using EPE. VML is used to specify the mapping between the IDM schema and the tool schema. A very simple mapping, implemented conventionally, is used to map from the tool schema into whatever actual format is required for the design tool. For example, for VISION-3D this involves walking the hierarchical object structure to produce a flat file acceptable as input to VISION-3D. These simple mappings are not described any further in this paper. The arrows on the mapping indicate directionality of information flow. For all tools except VISION-3D, information flow is bidirectional, meaning that changes in a model at either end of the link may cause changes in the model at the other end. In the demonstration system, VISION-3D is used purely as a rendering tool, and hence has only a one-way mapping defined between it and the IDM.

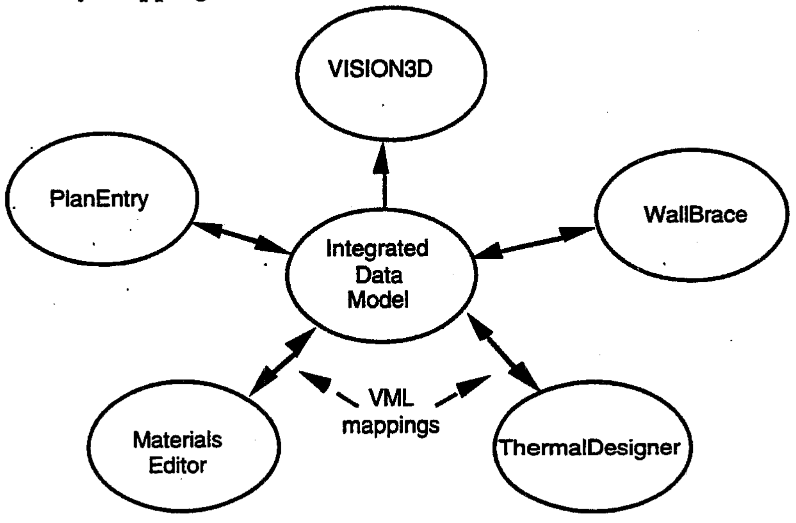

Figure 1: Architecture of the demonstration system

The nature of the tools to be integrated and the way in which we wish them to interact places some requirements on the mapping implementations. Firstly, the tools, with the exception of the VISION-3D interface, all have object-oriented data models, in contrast to the relational models assumed in many integration projects. In addition, we desire an interactive environment, so changes need to be propagated between the tools as quickly and incrementally as possible to maintain consistency between the various model views. This, again, is in contrast to the batch view taken in most other integration projects.

In this paper we detail experience in using VML to integrate the PlanEntry, materials editor, and VISION3-D tools with the IDM. The integration of WallBrace and ThermalDesigner is currently proceeding.

VML

VML (Amor 1994) is a high level declarative language (as compared to other mapping languages which are procedural in nature) for specifying inter-schema mappings. As VML is primarily declarative, a single mapping specification can often be used to map data in either direction between instances of the two models. A mapping is specified through a set of inter-class definitions. Each definition specifies correspondences between classes in the two models involved along with the conditions under which the correspondences hold. For example, Fig. 2 shows a simple inter-class definition between the idm_plane class in the IDM and the pf_plane_object class in the PlanEntry tool. The equivalences specify mappings between attributes in each of the classes involved. In this case the equivalences are direct equalities.

```
inter_class([idm_plane], [pf_plane_object],
   equivalences(
      name = planename,
      axis = axis,
      offset = offset,
      @view_plane = @select
   )
).
```

Figure 2: Simple VML inter-class definition for classes representing planes

An interesting feature of the language is that both data and method invocations may be mapped between the two schema. For example, in Fig 2 the view_plane method of idm_plane is mapped to the select method of pf_plane_object (the @ symbol indicates a method). Data mappings, such as the equality between the name attribute of the idm_plane and the planename attribute of pf_plane_object, effectively define bi-directional constraints between the attributes and objects involved. Thus a change to either attribute will propagate to the other. Method mappings trigger invocation of mapped-to methods whenever a method on either side of the mapping is invoked.

Figure 3 shows a more complex mapping between the idm_space_face class of the IDM and the fe_face and fe_face_window classes of the Materials Editor. The Materials Editor provides a two dimensional view (the face window) of a plane through the building containing a collection of space faces over which can be laid material and bracing data. This is not the only mapping involving these classes, so *invariants* specify the conditions under which this mapping applies. In this case the type_of_space attribute of idm_space_face must not be opening, and the fe_face object must be a member of the walls list attribute of the fe_face_window object.

The "dot" notation (eg fe_face_window.walls) is used to disambiguate attributes if an attribute of the same name exists in another class.

```
inter_class([idm_space_face], [fe_face, fe_face_window],
   invariants(
      type_of_face \= 'opening',
      member(fe_face, fe_face_window.walls)
   ),
   equivalences(
      min=>x = x0,
      min=>y = y0,
      max=>x = x1,
      max=>y = y1,
      plane = fe_face_window,
   ),
   initialisers(
      face_property = 'idm_space_face'
      fe_face@create( idm_space_face.plane, idm_space_face.plane,
                      'space', 0, 0, idm_space_face.min=>x,
                      idm_space_face.min=>y,idm_space_face.max=>x,
                      idm_space_face.max=>y)
   )
).
```

Figure 3: Mapping from IDM space faces to Materials Editor faces and editing windows

The equivalences in Fig. 3 are, again, direct equalities, but require some additional explanation. The arrow notation (eg min=>x) indicates an indirect attribute reference. In this case, the min attribute of the idm_space_face class is a reference to a point object, and the equivalence is with the x attribute of that point. The plane = fe_face_window mapping is another interesting one. The plane attribute of idm_space_face is a reference to the parent idm_plane object that this face is attached to. A different inter_class definition specifies a mapping between idm_plane objects and fe_face_window objects. The plane = fe_face_window equivalence then indicates that the fe_face_window object involved in this mapping is the same object that is mapped to from the idm_plane object referenced by the plane attribute. Figure 3 also shows two initialisers. The first initialises the attribute specified whenever its parent object is created. The create method initialiser specifies parameters required for object creation. In this case parameters are needed for the fe_face object creation call.

```
equivalences(
   plane=>offset = (offset - x)*scale,
   map_face_type_from_orientation(type_of_face,pe_face.orientation,
                                  pe_face, spaces)
)
```

Figure 4: Example of more complex equivalences

Figure 4 illustrates other variants of the equivalence specifications. In addition to direct equivalences, VML supports equations, such as plane=>offset = (offset - x)*scale. Equations can involve attributes from several classes and are inverted appropriately to solve for mapping in either direction. Equations can involve both invertible arithmetic operators, and a range of special operators, such as the aggregate operators (sum, count, minimum, maximum, average) to derive values from lists of values or references, which simplify the specification of correspondences. Mapping predicates may also be specified, such as map_face_type_from_orientation. These are implemented as Prolog code, which must be able to run in either direction to support bidirectional mapping. For particularly complex mappings, an equivalence may be specified as calls to two procedures, one for each direction of the mapping. None of this latter type of equivalence were required in this project.

Complex structural mappings require an additional feature of VML, illustrated in Fig. 5. This specifies a mapping between the PlanEntry pe_building and IDM idm_building classes. These classes have a number of associated attributes which are collections of references to other objects. The class pe_building has attributes: spaces, which is a collection object containing a list of references to pe_space objects; roofs, which is a collection object containing a list of references to pe_roof objects; and faces which is a list of references to pe_face objects, which specify face geometry. Class idm_building has attributes: spaces, which is a list of references to abstract spaces including roofs and spaces; and face_views, which is a list of references to different views associated with faces and which can include geometry-oriented views, materials oriented views, and bracing oriented views. There is thus a partial overlap between the sets of objects referenced by the attributes involved in each of the two classes. The idm_building spaces includes objects corresponding to both the spaces and roofs attribute of pe_building, while the faces attribute of pe_building contains objects corresponding to some of the idm_building face_views objects. This form of partitioning associated with the structural mapping requires more power than the invariant mechanism provides and necessitated the introduction of the bijection equivalences shown in Fig. 5.

```
inter_class([idm_building],  [pe_building],
   equivalences(
     bijection(spaces[]@class('idm_space'), spaces=>list[]),
     bijection(spaces[]@class('idm_roof'), roofs=>list[]),
          bijection(face_views[]@class('idm_space_face') = faces[])
   ),
   initialisers(
    name = 'PlanEntry building',
    pe_building.plan=>building = pe_building
   )
).
```

Figure 5: Example of bijections in mapping between PlanEntry and IDM buildings

Bijections provide a sophisticated set selection and manipulation capability associated with a mapping. For example consider the two bijections:

```
bijection(spaces[]@class('idm_space'), spaces=>list[]),
bijection(spaces[]@class('idm_roof'), roofs=>list[])
```

These together specify how the idm_building spaces are partitioned to the pe_building spaces and roofs attributes. Consider initially mapping from the IDM to

PlanEntry. The [] symbol indicate an iterator, so spaces[] in the first argument of the bijection indicates that the mapping needs to iterate over all of the elements in the idm_building spaces list. The part following that (@class('idm_space') or @class('idm_roof')) is a selection condition. Each object in the iteration that matches the selection condition is selected to take part in the mapping. In this case, objects of class idm_space are selected for the spaces mapping, while objects of class idm_roof are selected for the roofs mapping. The actual mapping of the individual objects involved is managed by another inter-class mapping relating the classes involved. The reverse direction mapping is specified by iterators and selectors in the second arguments. In this case, all elements of both the spaces list and the roofs list are selected (as there is no selection condition) for mapping. Changes to either of those lists then cause appropriate modifications to the idm_building spaces.

This bijection illustrates a problem unique to OO based systems. In a relational system the correspondence between a building and its spaces would be through a join of two relations, idm_building and idm_space, where the building key of the space matches that of the building and an attribute of the space object would determine whether it is a roof or not. In this OO system there is no need for a key to distinguish between spaces and roofs, the objects have behaviour which defines their 'roofness' or 'spaceness'. To distinguish between the types of space we need to access meta-information about the objects in the spaces list.

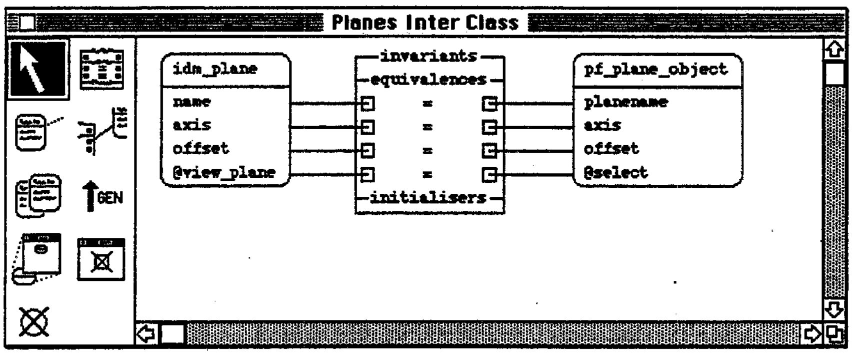

Figure 6: VML graphical form of the inter-class definition in Fig. 2

VML has both graphical and textual forms, and is supported by a specification environment that permits rapid construction of mapping definitions in both forms. The graphical form of the mapping of Fig. 2 is shown in Fig. 6. Some details of the textual specifications, such as equivalence or invariant equations, are omitted from the graphical form which just indicates the attributes involved in the relevant equivalences and mappings.

DYNAMIC BEHAVIOUR

The VML implementation supports a sophisticated transaction-based approach for managing consistency between an instance of a common schema and instances of subordinate tool schemas. The granularity of the transactions between the schema instances is under user control, and can vary from coarse, such as in updating the central model with the results of a simulation, to fine, such as a change to an individual attribute.

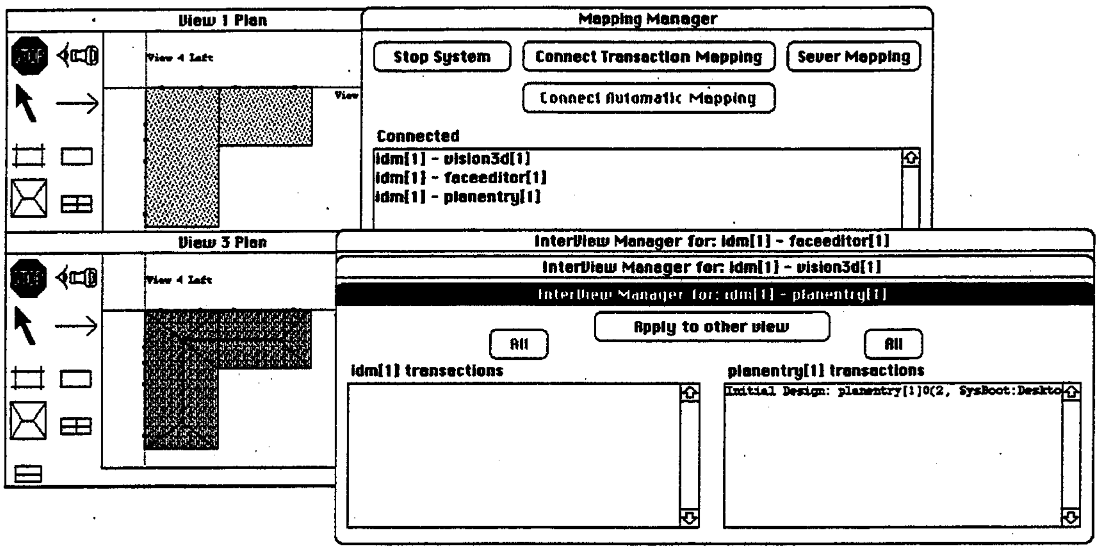
View 1 Plan
View 4 Left
Mapping Manager
Stop System
Connect Transaction Mapping
Sever Mapping
Connect Automatic Mapping
Connected
idm[1] - vision3d[1]
idm[1] - faceeditor[1]
idm[1] - planentry[1]
View 3 Plan
InterView Manager for: idm[1] - faceeditor[1]
InterView Manager for: idm[1] - vision3d[1]
InterView Manager for: idm[1] - planentry[1]
Apply to other view
All
All
idm[1] transactions
planentry[1] transactions
Initial Design: planentry[1]0(2, SysBoot:Deskto

Figure 7: Completion of transaction in PlanEntry ready to be passed to the other tools

Here we illustrate the use of this transaction system in the current implementation of our demonstration system. Figure 7 shows a building design for a simple L shaped building constructed using PlanEntry. The user is currently in the process of mapping the design (as a whole, in this case) to the IDM and then on to the other tools. The additional windows provide information about the transactions involved in the various mappings, and allow the user to instigate and control the mapping process. Fig 8 shows the result of subsequently mapping the model through to VISION3-D (at rear) and mapping one of the faces through to the materials editor to allow addition of materials and bracing information for that face.

RELATED WORK

Defining mappings for integrated design systems in the A/E/C domain is receiving a lot of attention as the ISO STEP project progresses and as integrated design systems move from the research projects out towards real world applications. There are several specialised languages that have been developed (see Verhoef et al 1995 for a review of mapping languages) for this domain. These languages are mainly procedural in nature, requiring two mappings to define a bidirectional mapping and in the main dealing with a batch oriented model of the integrated design system.

CONCLUSIONS

We have demonstrated the utility of a declarative approach to the specification of inter-schema mappings in the context of a demonstration integrated building design system. The VML language used for mapping specification provides a high level, easily maintained, description of the mappings between the various tool and integrated models used. The transaction-based mapping implementation demonstrates the viability of the language for implementing inter-tool data transfer.

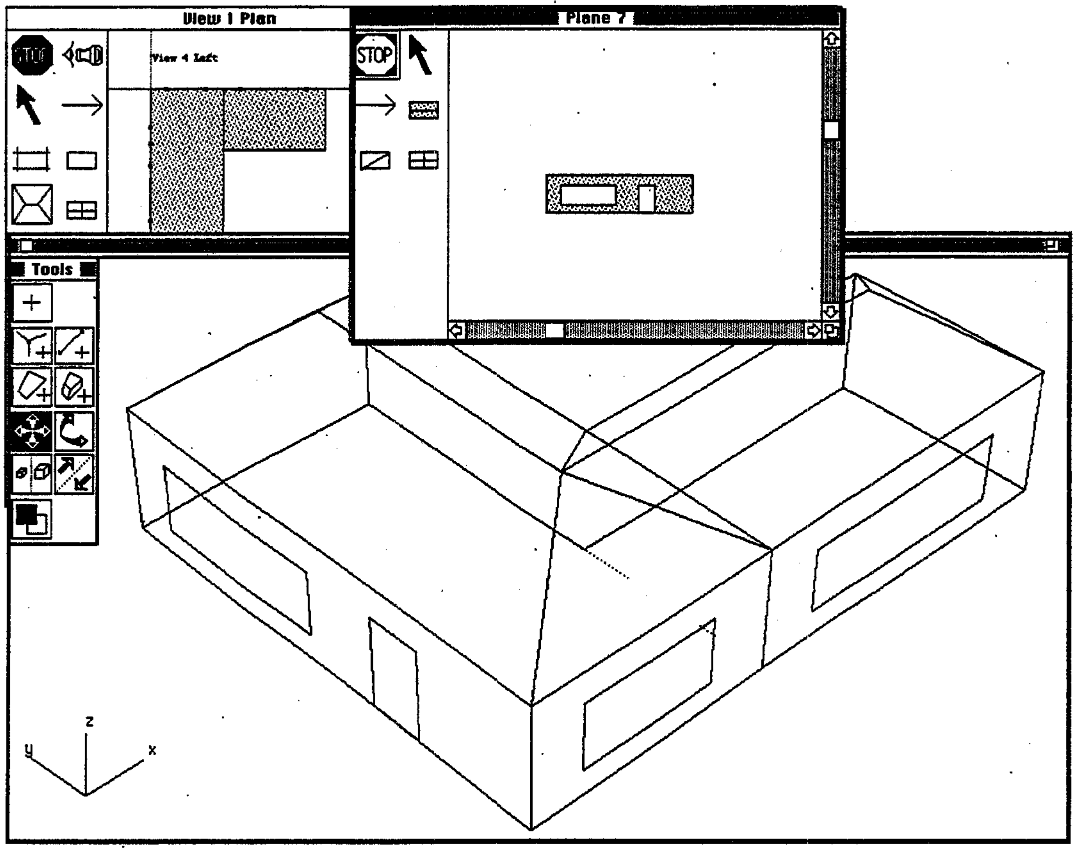

Figure 8: Initial PlanEntry design propagated through to all connected tools

Through the specification of VML mappings for the design tools in this integrated design system we noted difficulties in the specification of mappings to object-oriented systems that do not occur in relationally structured applications. This problem is to do with the notion of keys in relational database systems (RDBMS) and the use of the object ID as an identifier in OO systems. The problem surfaces when we need to modify the structural layout of objects between schemas being mapped. In a RDBMS system the structure is defined implicitly through a series of operations on keys of the various tables. When we need to create a different structure we need only order the queries to utilise keys in a different manner. In an OO system the relationships between objects is specified explicitly through references to other object ID's and in many cases key information is not held in all the objects referenced in this manner. Hence, if the structure needs to be drastically re-organised there may not be enough information in the referenced objects to perform the rearranging automatically.

In VML there are several methods to manage this problem. Bijections allow the re-arrangement of lists of pointers between two structures where there is enough information in the referenced objects. VML's group() specifier in the inter_class class specification allows sets of objects to be collected that match criteria in the invariants section and these sets of objects can be assigned to list attributes of a new class.

In the next stage of our work we plan to integrate the code conformance tools, ThermalDesigner and WallBrace, to complete the demonstration system. Initially this will be by a direct mapping from the IDM to the existing structure of these systems. However, of particular interest in this work is the amount by which an application such as ThermalDesigner could be restructured given that it no longer requires a substantial data entry component. Much of the structure was originally imposed on this application (and on WallBrace) to make the task of data entry simpler. Eliminating the need for this via direct attachment to the IDM would make a simplification of the application structure viable. Likewise, the approach to developing new applications to be integrated via the IDM is likely to be considerably different, with a more component-oriented approach taken rather than a stand-alone application view.

A further area of research is that of extensions to VML. VML currently does not map constraints, such as are used in both PlanEntry and the materials editor. This makes the management of constraint interaction across multiple applications somewhat more difficult. By including such constraints in the mappings a more effective approach to integration could be made. In particular, it would avoid the assumption that interdependencies of redundant data, such as the materials and bracing views, will be managed properly by any tool that changes part of it.

ACKNOWLEDGMENTS

The authors acknowledge the financial support of the Building Research Association of New Zealand, the New Zealand Foundation for Research Science and Technology, and the University of Auckland Research Committee in performing this research.

REFERENCES

Amor, R. (1994) A Mapping Language for Views, Department of Computer Science, University of Auckland, Internal Report, 30p.

Amor, R., Augenbroe, G., Hosking, J., Rombouts, W. and Grundy, J. (1995a) Directions in Modelling Environments, accepted for publication in *Automation in Construction*.

Amor, R.W., Hosking, J.G., and Mugridge, W.B. (1995b) A Review of Computerised Standards Support in New Zealand, submitted to International Journal of Construction Information Technology.

Bourke, P.D. (1989) *VISION-3D User Manual*, School of Architecture, University of Auckland, Auckland, New Zealand.

Hosking, J.G., Mugridge, W.B. and Blackmore, S. (1994) Objects and constraints: a constraint based approach to plan drawing, in Mingins, C. and Meyer, B. Technology of object-oriented languages and systems TOOLS 15, Prentice Hall, Sydney, pp 9-19.

Mugridge, W.B. and Hosking, J.G. (1995) Towards a lazy evolutionary common building model, Building and Environment, 30(1), pp 99-114.

Verhoef, M., Liebich, T. and Amor, R. (1995) Multi-Paradigm Mapping Method Comparison, accepted for presentation at CIB workshop on Modeling of Buildings through their Life-cycle, 21-23 August, Stanford, USA.