Chapter 5

The View Mapping Language (VML)

All the work described in the previous section tackles the problem of mapping between schemas to some extent, but none of the approaches described provide the full range of abilities described in Section 4.1.3 as being required for a general inter-schema mapping language. In this section the View Mapping Language (VML) is presented. VML overcomes many of the problems identified in the languages canvassed in Chapter 4. VML is a high-level, declarative, and bidirectional language suitable for the description of correspondences between two arbitrary schemas of a domain. VML dispenses with all notions of target and source schemas in the mapping definition. As far as practicable, a VML definition treats both schemas as equal partners in a mapping. VML also removes many distinctions between entities and attributes, to allow mappings between entities and attributes to be specified in the same way that attribute to attribute mappings are specified.

Throughout this section examples are used to illustrate each construct in the VML language. These examples are drawn from the large mapping example described in Section 1.6 and are fully specified in Appendix E. However, to complement the examples in Section 4.2, the VML specification for the example shown in Table 4.5 can be seen in Table 5.1. All of the components of this formalism are fully described later in this section, but, a brief description of the mapping in Table 5.1 is as follows. The example shows separate *inter_class* definitions for each type of class to be mapped between, along with *invariants* specifying the conditions under which the mapping holds. This combination of classes and invariants must be unique for every *inter_class* definition, and will be checked by any mapping implementation. *Equivalences* specify the mappings to be undertaken, all with implicit type conversion and implicit object type mapping (i.e., for *related* and *relating*). All *inter_class* definitions can be used in both directions depending upon where data resides that requires mapping. *Initialisers* specify initial values of attributes when they are created by the application of a mapping.

```
inter_class([component_relationship], [support_connector],
        invariants(
                quality = 'support_connection'
        ),
        equivalences(
                id = identified_by,
                related = related,
                relating = relating
        ),
        initialisers(
                type_of = 'un_known'
        )
).

inter_class([component_relationship], [element_connector],
        invariants(
                quality = 'element_connection'
        ),
        equivalences(
                id = identified_by,
                related = related,
                relating = relating
        ),
        initialisers(
                type_of = 'un_known'
        )
).
```

Table 5.1 VML mapping for example problem

In a VML environment it is assumed that all mappings are between two schemas. When it is necessary to map information between several schemas and a single schema each mapping is specified independently so that the mapping implementation can manage updates to and from each model independently.

At the top level, considering the complete mapping between schemas, the work from database views gives two possibilities for the types of mappings which can exist. These are read-only views and read-write views. This carries over to schema mappings as well: a mapping can operate in one direction, giving a read-only view; or in both directions giving read-write views.

A VML mapping consists of an introductory specification of the schemas to be mapped between, and then a set of correspondences between entities and attributes to describe how the mapping is to be achieved (see Table 5.2). The syntax of VML is similar in style to that of Prolog and Snart, the implementation languages of this thesis, but it could easily be rewritten to resemble the syntactic style of EXPRESS or other modelling languages without affecting the semantics of the language.

```
mapping = inter_view_def { inter_class_def } .
```

Table 5.2 Top level definition of a VML mapping

5.1 Mapping between schemas

A VML mapping definition commences with an *inter_view* definition, which specifies the two schemas between which a mapping is to be described, the type of view represented by the schemas in this mapping, and the completeness of mapping that is required (see Table 5.3).

```
inter_view_def = 'inter_view(' model_id ',' model_type ',' model_id ',' model_type ','
        map_type ')' '.' .
model_id = simple_id [ '{' version '}' ] .
model_type = 'integrated ' |
        'read_only ' |
        'read_write ' .
version = integer_literal |
        real_literal |
        atom_literal |
        string_literal .
map_type = 'complete' |
        'partial' .

For example:
inter_view(idm{0.09}, integrated, planentry, read_write, complete).
```

Table 5.3 Definition of an *inter_view* specification

The *model_id*s specify the names and optional version numbers of the schemas which are being mapped between in this mapping. The first *model_id* specified is treated as the left-hand side schema and the second as the right-hand side schema. The side of the schema is used to determine which entities and attributes are being referenced in a mapping. If an entity appears on the left-hand side of a specification then, by default, it belongs to the first schema specified in the *inter_view* (though this can be explicitly over-written, as described later).

The optional version number (e.g., *idm{0.09}* in Table 5.3) associated with a schema enables a mapping to be specified to a specific version of a schema. The use of version numbers also allows a mapping to be specified between different versions of the same schema, supporting schema evolution.

The *model_type* indicates the role that a schema instance plays in this mapping. As the reason for a mapping between models varies depending upon the model being connected, the *model_type* allows the specification of different roles for a model dependent upon the connection being made. There are three allowable values for *model_type*:

read_only: specifies that data can be mapped from that model to the other model, but not vice-versa. Specifying a *read_only* role does not restrict the data in either model from being modified, it just affects a model's ability to pass changes through to its connected model. The *model_type* for both models can not be set to *read_only*.

read_write: specifies that data can be mapped to this model as well as mapped out of it. Where both models are depicted as *read_write* there is no synchronisation when propagating changes between the models.

integrated: has almost the same meaning as a *read_write* model. The exception is that all changes waiting to be passed on from the *integrated* model must be accepted before a change is mapped to this model. The need for this level of synchronisation is discussed in Chapter 2. Only one of the two model types can be *integrated* to avoid deadlock situations.

The three *model_type*s cover all the connections that may be required between models. A *read_only* model is for the situation where it is necessary to provide information to a participant but not necessary for them to modify the model it is being passed from. The *read_write* and *integrated* model types are for mappings where changes from the connected model can be accepted. With the *integrated* model type it is possible to ensure the consistency of data in a connected model before allowing modifications to be made by it. Individual models can play different roles in different mapping specifications which can lead to interesting chains of connections. For example, a model which has a *read_only* connection to one model could have an *integrated* connection to another set of models. This type of connection could be used to provide a barrier between a central model and a set of users who wish to experiment with aspects of the data as well as share their modifications with each other (see Figure 5.1).

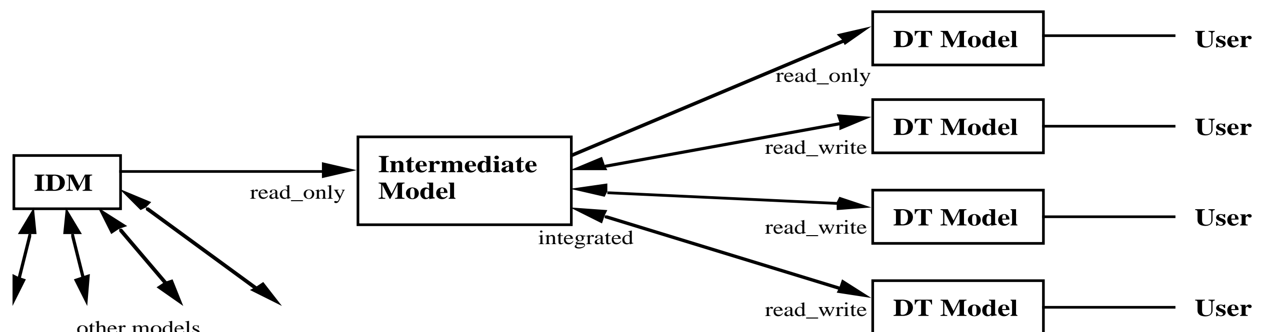

Figure 5.1 A tree of *inter_view* mappings

The *map_type* indicates how complete the movement of data needs to be between the two models. The two allowable values for *map_type* are:

complete: all objects of the entities described in the mapping must be mapped to the datastore for the other model. This is used for models representing the same type of objects (e.g., a whole building) and where a mapping only makes sense if everything can be moved across. That is, the derived model is inconsistent if it is not possible to map all objects of types mentioned in the mapping specification.

partial: is used where one schema describes a much smaller domain than the other (e.g., a room schema versus a building schema) and where it is not necessary to map all the information across to the other model (e.g., only a single room is required to be mapped rather than all the rooms in the building).

5.2 Mapping between classes

Following the *inter_view* definition is a set of *inter_class* definitions which describe the relationships between entities in the two schemas. An *inter_class* definition details: the entities from both schemas that take part in the mapping; an optional set of conditions which must hold to use the mapping; the actual relationship between data in the two entities; and, optionally, initial values for attributes when an object is created (see Table 5.4). Table 5.4 also shows an example *inter_class* specification. This example describes the mapping between *idm_space_face* objects in one schema and *v3d_polygon* objects in another schema under the condition that the *type_of_face* of the *idm_space_face* is equivalent to the value *opening*. Where this is the case there are two functional mappings specified in the equivalences section to map the object identifier and shape between representations. There are also initial values specified for attributes of the *v3d_polygon* object which provide default reflection and colour information.

```
inter_class_def = 'inter_class(' class_list ',' class_list [ ',' inherits ]
        [ ',' invariants_def ] [ ',' equivalences_def ] [ ',' initialisers_def ] ').' .
class_list = '[' [ class_key_name { ',' class_key_name } ] ']' .
class_key_name = 'group(' class_name ')' | class_name .
inherits = 'inherits(' inherit_list ')' .
invariants_def = 'invariants(' invariant_expr { or_op invariant_expr } ')' .
equivalences_def = 'equivalences(' equivalent { ',' equivalent } ')' .
initialisers_def = 'initialisers(' initialiser { ',' initialiser } ')' .

For example:
inter_class([idm_space_face],[v3d_polygon],
        invariants(
                type_of_face = 'opening'
        ),
        equivalences(
                map_id_to_num(idm_space_face, object_id),
                map_polar_rect_to_polygon(min=>x, min=>y, max=>x, max=>y, plane=>axis,
                        plane=>offset, points[1], points[2], points[3], points[4])
        ),
        initialisers(
                diffuse_reflection = 0.1,
                colour=>r = 0,
                colour=>g = 0,
                colour=>b = 0
        )
).
```

Table 5.4 Definition of an *inter_class* specification

5.2.1 Entity names and keys

The two class lists in an *inter_class* definition specify the entities which are involved in the mapping being specified. As detailed in the *inter_view* definition, the first *class_list* refers, by default, to entities from the first schema and the second to the second schema. The union of the two lists of entity names provides the key for this mapping. There can be any number of mappings with the same key. However, each specification must be distinguishable from the others by the conditions specified in the *invariants* specification. In the example shown in Table 5.4 the key is [idm_space_face, v3d_polygon] and though there are several *inter_class* definitions between these

two classes (see Appendix E) they are all uniquely identified by their invariants, in this case *type_of_face = 'opening '*.

In some mappings it is necessary, and in many other mappings very convenient, to be able to describe a mapping to temporary entities, e.g., where two mappings can reuse a partial mapping in their transformations. To be able to distinguish mappings to temporary entities from those which create real objects in a particular view we denote temporary entities by names prefixed either with an underscore symbol, e.g., _temp1, _temp2, or with a capital letter, e.g., Temp1, Temp2 (see Table 5.5 for the syntax of allowable class names). Temporary entities provide a mechanism to specify entities which do not exist in the schemas of the two systems that are being mapped between.

In Table 5.4 the *class_list* is defined as consisting of one or more *class_key_name*s. Having more than one class specified in a *class_list* allows the mapper to associate objects from a model when the objects previously had no association. A *class_list* with multiple *class_key_name*s denotes that when constructing an instance of this mapping, an object from each class in the *class_list* is required. This provides a way of associating objects of classes which may not be directly accessible from objects of the first class defined in the list, creating an effect similar to a join in a relational database system. The manner in which objects are associated is dependent upon the invariants specified in the mapping. Where no invariants are specified, the number of mappings that would be performed is equal to the cross-product of all objects for each of the named classes. Where invariants are specified, the number of mappings is restricted by application of the invariants to the cross-product of all objects of all named classes. The following small example helps illustrate how this works. The example shows five objects, two of class *a* and three of class *b*. Both classes have an attribute called *type,* and the value of *type* is shown for all five objects. Two *inter_class* definitions are shown. The first has no invariants specified, and as can be seen from the set of object pairs, displayed after the *inter_class,* this forces a complete cross-product of objects from both classes. The second *inter_class* has an invariant requiring the *type* attribute of objects of class *a* and *b* to be equivalent. This reduces the object grouping down to three sets of object pairs, rather than the six for the full cross-product.

Object ID	Class	Object.type
o1	a	1
o2	a	2
o3	b	1
o4	b	1
o5	b	2

```
inter_class([a, b], [c], .....).
        [[o1, o3], [o1, o4], [o1, o5], [o2, o3], [o2, o4], [o2, o5]]
inter_class([a, b], [c], invariants(a.type = b.type), .....).
        [[o1, o3], [o1, o4], [o2, o5]]
```

The *group()* specifier also suppresses creation of the full cross-product. It allows a collection of objects of the named class to be grouped together for the purposes of the mapping. This is commonly used to group multiple objects of a single class into a collection that is not explicitly supported in the original schema, and then to map the collection to a schema which requires this grouping. Without invariants, *group()* selects all objects of the named class; with invariants, the objects of the class are restricted by application of the invariants to each individual object being grouped. To illustrate how *group()* works consider the small example above with modified *inter_class* definitions as below. The first *inter_class* shows that all objects of class *b* are grouped with objects of class *a* in a set rather than a cross-product. The second *inter_class* shows that the objects in the grouped set can be restricted through the use of invariants.

inter_class([a, group(b)], [c], ...).
[[o1, [o3, o4, o5]], [o2, [o3, o4, o5]]]
inter_class([a, group(b)], [c], invariants(a.type = b.type), ...).
[[o1, [o3, o4]], [o2, [o5]]]

In an *inter_class* specification, one of the *class_list*s can be left empty. This allows the specification of initial conditions that must be established when a mapping between models is initiated. In most systems this would be to allow the creation of an object with initial values when a model is created (e.g., the controller of a design tool, or entities which have no representation in the schema being mapped from).

Table 5.5 shows the definition of a class name. While the default reading of a mapping is that the order of schemas in the *inter_view* definition is the order of classes in an *inter_class* definition the ordering can be overridden through the specification of the *model_id* in the class name. This allows classes from a schema to be specified in either side of an *inter_class*, and also allows for mappings between classes of a schema and temporary entities to be defined (or between temporary entities and temporary entities) in any order the mapping specifier desires.

```
class_name = [ model_id ':' ] class_id |
       variable_id .
model_id = simple_id [ '{' version '}' ] .
class_id = simple_id .
variable_id = upper_case { simple_id_char } |
       '_' ( letter | digit ) { simple_id_char } .

For example:
idm_space_face
idm{0.09}:building
_temp
```

Table 5.5 Definition of a *class_name*

5.2.2 Inheritance of *inter_class* definitions

The specification of inherited *inter_class* definitions allows *inter_class* specifications to closely model the structures that are found in object-oriented schemas. Where there are correspondences between the parent classes of a set of child classes it is more efficient and more maintainable to

specify inherited correspondences between the parent classes than to re-specify the mappings for each child class. The type of mapping where this feature will be most commonly utilised is in version mapping for object-oriented schemas. The specification of an inherited mapping, as shown in Table 5.6, utilises the key of a mapping to determine which mappings to inherit. Where the mapping specified has multiple definitions for that key (i.e., with different invariant specifications), then the combined mapping will be expanded into a set of mappings encompassing all combinations of invariants from the inherited *inter_class* definitions.

```
inherits = 'inherits(' inherit_list ')' .
inherit_list = inherit_map { ',' inherit_map } .
inherit_map = 'inter_class(' class_list ',' class_list ')' .

For example:
inherits(inter_class([person],[person]))
```

Table 5.6 Definition of inheritance

5.2.3 Invariant specification

Invariants are an optional part of an *inter_class* definition, and describe the conditions under which it is possible to use a particular *inter_class* definition. For example, defining an invariant *invariants(building.type = 'commercial')* in an *inter_class* definition would denote that it was only possible to use this *inter_class* definition for buildings which are commercial buildings, and presumably there would be other *inter_class* definitions which would specify what to do with other types of buildings. Thus, invariants are selection criteria to use when deciding which *inter_class* definition to apply to any given object. Each individual invariant expression can only reference objects and attributes from a single schema in the mapping. Invariants are therefore broken into two sets, those which apply to classes in one schema and those which apply to classes in the other. When deciding if an *inter_class* definition is to be used, all invariants which apply to the schema being mapped from must evaluate to *true* on the objects being tested.

```
invariants_def = 'invariants(' invariant_expr { or_op invariant_expr } ')' .
invariant_expr = invariant_simple_expr { and_op invariant_simple_expr } .
invariant_simple_expr = '(' invariant_expr { or_op invariant_expr } ')' |
        expression rel_op expression |
        predicate |
        function |
        method |
        'group(' attribute_name { ',' attribute_name } ')' .

For example:
invariants(
        type_of_face \= 'opening',
        pe_face.offset = pf_plane_object.offset,
        map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
        contained_in_face(pe_face, pe_opening)
)
```

Table 5.7 Definition of invariants

The invariants also create constraints on the objects that are mapped, which in some cases can be used to fill in values in an object, or in others to create constraints on an object. In the example above it can be seen that when an object is mapped back onto the building with invariant

building.type = 'commercial' then the value of *type* can be automatically set to *'commercial'* without having to use an equivalence definition to specify this.

Invariants can be thought of as boolean expressions which must equate to *true* to allow the mapping to proceed. Invariants (the syntax of which is shown in Table 5.7) can be composed of functions, predicates or object method calls which succeed or fail, or expressions containing relational operators (e.g., =, >=, <). Sets of invariant conditions can be joined together through the use of *and* operators ',' or *or* operators ';'. VML offers one special function to be used with grouped classes in an *inter_class* definition. The *group()* function can only be used with a grouped class (see Section 5.2.1). It can only be used on an attribute of a grouped class which has a finite domain (usually an enumerated type) and is used to collate all objects of the named class whose value for the named attribute is identical. For example, specifying *inter_class([group(building)], [aggregation_of_type], invariants(group(building.type)), ...)* would ensure that a separate mapping would be performed for each type of building in the first model, pooling values into a single object in the model of the second schema for every different value of *type* found in the model of the first schema.

The example in Table 5.7 provides a complex set of invariants which must be satisfied to allow a particular *inter_class* specification to be used. In this specification the *type_of_face* attribute must contain the value *'opening'*, and the *offset* object referenced by *pe_face* and *pf_plane_object* must be the same, also the functions *map_orientation_axis()* and *contained_in_face()* must evaluate to true with the supplied parameters.

```
initialisers_def = 'initialisers(' initialiser { ',' initialiser } ')' .
initialiser = expression '=' expression |
        predicate |
        method .

For example:
initialisers(
        idm_space_face.face_property = 'idm_space_face',
        idm_material_face.face_property = 'idm_material_face',
        idm_material_face.material=>type_of_material = 'idm_window_material',
        idm_material_face.material=>type_of_window = 'idm_single',
        idm_material_face.material=>window_subtype = 'clear',
        fe_opening@create(idm_space_face.plane, idm_space_face.plane, 'space', 0, 0,
                idm_space_face.min=>x, 0 - idm_space_face.min=>y,
                idm_space_face.max=>x, 0 - idm_space_face.max=>y,
                idm_material_face.material=>window_subtype)
)
```
Table 5.8 Definition of initialisers

5.2.4 Initialiser specification

The optional initialiser section allows the definition of initial values for attributes of objects created in an *inter_class* specification. For models that are object-oriented, the initialiser section also provides a location to specify the *create* method parameters for objects that may be created during

the mapping. The initialiser section (the syntax of which is defined in Table 5.8) usually comprises mainly assignment statements specifying values for attributes, though predicates and procedures can also be specified.

Any entity attribute, or referenced attribute, of any class specified in the class lists of the *inter_class* specification can be initialised in the initialiser section. Initialisers will only be applied to newly created objects (i.e., not to an existing object which is associated through an invariant specification) and may cause the creation of other objects (e.g., attribute assignment through pointer chains).

5.2.5 Equivalence specification

The equivalence section comprises the bulk of most *inter_class* definitions as it specifies the correspondences between the attributes of entities defined in the class lists of the *inter_class*. It is this section which contains all the equations, functions, and procedures which will need to be solved or executed to map between models of the schemas in the mapping. The declarative nature of VML is most evident in this section as equivalences are used to define mappings between attributes. The ordering of expressions is unimportant as their solution is dependent only on the state of the model being mapped from. The syntax of the equivalence section is shown in Table 5.9.

```
equivalences_def = 'equivalences(' equivalent { ',' equivalent } ')' .
equivalent = expression '=' expression |
        'map_to_from(' predicate ',' predicate ')' |
        'bijection(' bijection_expr ',' bijection_expr ')' |
        predicate .

For example:
equivalences(
        bijection(idm_space_face[].type_of_face \= 'opening', walls[]),
        bijection(idm_space_face[].type_of_face = 'opening', openings[]),
        idm_material_face = materials,
        idm_bracing_face = bracing,
        idm_plane.name = name,
        idm_plane@view_plane = fe_application@create_view(_, idm_plane.name)
)
```

Table 5.9 Definition of equivalences

5.2.6 Mapping equations

The style of equations that can be used to define the mapping between classes was developed to meet the mapping types identified in Section 4.1.3. The major abilities of the VML language are described below. For the full syntax of the language refer to Appendix A.

5.2.6.1 Attribute initialisation or constant value specification

A constant value can be assigned to an attribute by equating the value with the named attribute as shown in the examples below.

type_of_face = 'opening'

diffuse_reflection = 0.1

gloss_factor = 90.0

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser these equations provide the initial value for an attribute which may be modified by other equations in the equivalences section. When specified in an invariant or equivalence they specify a constant value. If the value is specified in the invariant it is used either to determine which mapping to apply to the class the attribute belongs to, or to initialise an attribute for a newly created object. If the value is specified in an equivalence, it specifies a value for the attribute which may not be modified (i.e., it will always be reset to the specified value). This specification corresponds to the Attr->Attr mapping of cardinality 0:1.

5.2.6.2 Equality

Direct equality between two attributes of a simple type (e.g., REAL, INTEGER, BOOLEAN), or named types, can be specified by equating one attribute with the other attribute as shown in the examples below.

name = planename

axis = axis

offset = offset

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser these equations provide the initial value for an attribute which may be modified by other equations in the equivalences section. If the equality is specified in the invariant it specifies attributes of entities on the same side of a schema, the values of which must match for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence then it denotes that the attributes of the respective entities must hold the same value. This specification corresponds to the Attr->Attr mapping of cardinality 1:1.

5.2.6.3 Pointer equality

Equality between pointers to objects of entities in the schemas is specified in the same manner as for equality between attributes of simple types, e.g.:

plane = fe_face_window

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser these equations provide the initial object reference for an attribute which may be modified by other equations in the equivalences section. If the equality is specified in the invariant, then it will be specifying reference attributes of entities on the same side of a schema whose object references must match for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence, then it denotes that each attribute has the object identifier of the object that was created from the *inter_class* definition between the entities of the two objects. This specification corresponds to the Attr->Attr mapping of cardinality 1:1.

5.2.6.4 Simple equations

Equations can also be used to define relationships between various attributes, as shown in the examples below. The range of algebraic and transcendental functions available is the set of functions supported by LPA Prolog.

min=>y = 0 - y0
r * sin(theta) = y_coord
r = sqrt(x_coord * x_coord + y_coord * y_coord)

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser, these equations provide the initial calculated value for an attribute, which may be modified by other equations in the equivalences section. If the equation is specified in the invariant, then it specifies attributes of entities on the same side of a schema, the calculated value of which must match for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence, then it denotes that the attributes of the respective entities must hold the value determined by the re-arrangement of the equation to solve for each particular attribute. This specification corresponds to the Attr->Attr mapping of cardinalities 0:1, 1:1, 1:C, C:1, C:B. Where there are more than two attributes in an equation (i.e., cardinalities 1:C, C:1, C:B) it is assumed the implemented mapping system will re-arrange equations to solve for unknown values, or arrange sets of equations to ensure that values calculable in other equations can be used to solve more complex equations, the method for achieving this in an implementation of VML is described in Chapter 9.

5.2.6.5 Pointer references

Following pointer chains to reference attributes of the referenced object is possible with the => operator in VML, as shown below. Pointer chains can be followed to any depth and provide a method for collapsing, or expanding, reference structures in a schema.

apex1=>x = apex2=>x

Pointer references have the same semantics wherever they are used. When solving equations which contain pointer references, it is sometimes necessary to create objects of the referenced type to have a full reference for solving the equation. For example, in the equation above, if *apex1* had no value it would be necessary to create an object of type *point*, the reference to which would become the value of *apex1*, so the attribute *x* could be set to the value of *apex2=>x*.

5.2.6.6 Functions

Functions in VML are the built-in functions of LPA Prolog, which are predicates which either fail or succeed along with the special functions *exists/1* and *var/1* which are used to determine whether an attribute has a value specified or if it has not been assigned to (see the examples below).

member(fe_face_material, fe_face_window.materials)
exists(end_point=>z)
var(end_point=>z)

Functions are used in the invariants section of an *inter_class* definition to determine whether to use the particular *inter_class* with the objects currently under consideration.

5.2.6.7 Aggregation functions

In contrast to the functions described above, aggregation functions are not invocable in both directions, or re-writable as most equations are. The set of aggregation functions (sum, maximum, minimum, average, count) provide summary details of lists of values or objects, as shown in the examples below.

```
sum(wall.windows=>(height * width)) = glazing_area
maximum(panes=>(offset=>y + height)) - minimum(panes=>offset=>y) = height
```

Aggregation functions can not be solved in both directions (e.g., for the first example, knowing the *glazing_area* does not allow the calculation of *height* and *width* of all the *windows* in a *wall*). Instead, when performing a mapping to an equation with an aggregation function, the calculated value from one side of the equation is used to specify a constraint on the values of the schema the mapping is being applied to. For instance, in the first example above, if the *glazing_area* is known, then the sum of *height * width* for all *windows* in the *wall* will be constrained to the value of *glazing_area*. Aggregation functions can be used in any part of an *inter_class* definition. The implementation of these constraints is assumed to be handled by the implemented mapping system, Chapter 10 details how this was achieved for one VML implementation.

5.2.6.8 List and array references

Individual elements of a list or an array may be referenced through the indexing operator. The examples below show equations which reference elements of a list and an element of a 3-dimensional array.

```
axes[2] = v_ref
exists(axes[2]=>direction_ratios[3])
vector[2, 2, 3] = iz
```

List and array references can be used in any part of an *inter_class* definition.

5.2.6.9 List and array iteration

A mapping between lists or arrays of the same dimensions and bounds can be specified through the iterator operator. This operator provides a short-cut notation for the specification of for-loops over the contents of lists and arrays, as long as the relationship can be specified in an equation. Mappings between lists and arrays which do not meet these conditions must be specified with predicates, procedures or methods. The example below shows a mapping between a list of strings (*classified_by*) and a list of object references (*material*) which has an attribute, *name*, of type *string*. The result of a mapping between these two attributes will be that the first item in the *classified_by* list is identical to the value of *name* of the first object reference in *material*, and so on for the number of items in *classified_by*, or in *material*, depending upon which direction the mapping is being run.

classified_by[] = material[].name

Iterators can appear in invariants, equivalences and initialisers and have slightly different semantics in each. When specified in an initialiser, an iterator provides the initial calculated values for an attribute of *list*, *set*, *bag*, or *array* type (within the specified bounds of the aggregate type), which may be modified by other equations in the equivalences section. If the equation is specified in the invariant then it specifies a condition which must hold for every value in the aggregate attribute referenced for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence, it denotes that each element of the iterated attributes of the respective entities must hold the value determined by the re-arrangement of the equation to solve for each particular attribute. This specification corresponds to Attr->Attr mapping of cardinalities 1:1, 1:C, C:1, C:B. It can also denote Attr->Entity mapping of cardinalities 1:C, C:B, 1:N, or Entity-Attr mapping of cardinalities C:1, C:B, N:1, or Entity->Entity mapping of cardinality 1:1.

5.2.6.10 Conditional list and array iteration

VML provides bijections to extend the flexibility of the iterator operator. Bijections allow the specification of conditions to be checked on each element of an aggregate attribute, or grouped set of objects (formed by a *group()* specification in a class list), before a mapping can take place. The examples below show the bijection between lists of object references where the mapping is conditional on the class type of each object in the list. In the example below the right hand schema entity has attributes: *spaces*, which is a collection object containing a list of references to space objects; *roofs*, which is a collection object containing a list of references to roof objects; and *faces,* which is a list of references to face objects, which specify face geometry. Entity *idm_building* from the left hand schema 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 first bijection specifies that only *idm_building* spaces which are really living spaces (i.e., their type is *idm_space*) should be mapped to the *spaces=>list*, though all spaces in *spaces=>list* can be mapped to the *idm_building* spaces aggregate attribute. The second bijection specifies the same conditions, except this time for roof objects, and the third bijection performs a similar mapping for geometric faces. The @ operator specifies a method call in the example bijections, in this case to the meta-method *class()* which returns the class of the referenced object. The first bijection therefore iterates through all objects in the *spaces* list, extracting those whose class is of type *idm_space*.

bijection(idm_building.spaces[]@class('idm_space'), spaces=>list[]),
bijection(idm_building.spaces[]@class('idm_roof'), roofs=>list[]),
bijection(idm_building.face_views[]@class('idm_space_face'), faces[])

Bijections may only appear in the equivalences section of an *inter_class* definition. Bijections can be run in both directions and are taken to mean: iterate over all elements in the specified attribute or group of objects, but perform a mapping only for those elements which match the conditions

which are specified. Conditions can be in the form of method calls (as in the example above), predicate calls, or conditional equations. As multiple bijections can be specified over a single attribute (e.g., *idm_building.spaces[]* in the examples above), the result of the conditional evaluation and the equation solving are unioned with values from the other bijections on the list or array being mapped to. In the example above this means that in mapping from the right-hand side to the left-hand side the *idm_building.spaces[]* will contain the union of *spaces* and *roofs* from the right-hand side schema list attributes being mapped from. This specification corresponds to Attr->Attr mapping of cardinalities 1:1, 1:C, C:1, C:B. It can also denote Attr->Entity mapping of cardinalities 1:C, C:B, 1:N, or Entity-Attr mapping of cardinalities C:1, C:B, N:1, or Entity->Entity mapping of cardinalities 1:1, 1:C, C:1, C:B, N:M.

5.2.6.11 Functions

User defined functions extend the set of built-in functions of VML. User defined functions are broken into two categories: those which reference attributes from a single schema; and those which reference attributes from both schemas (examples of the latter are shown below, as detailed in Appendix E).

```
list_splitter(vals, _temp_schedule.splitvals)
map_polar_rect_to_polygon(min=>x, min=>y, max=>x, max=>y, plane=>axis,
        plane=>offset, points[1], points[2], points[3], points[4])
```

User defined functions which reference attributes from one schema are used in the invariants and initialisers sections of the *inter_class* definition. User defined functions which reference attributes from both schemas must be invocable with the attributes from either side instantiated (i.e., able to run in either direction). When used, these functions are called with the values from the schema the mapping is coming from, and the results are used to instantiate the attributes of the schema the mapping is being applied to. A function is not invoked unless all function attributes of the schema the mapping is coming from have values. In general, functions may not manipulate objects (e.g., create new objects, delete objects, reference object attributes, call object methods), but allow structures to be manipulated and values to be computed. Functions correspond to the Attr->Attr mapping of cardinality 0:1, 1:1, 1:C, C:1, C:B.

5.2.6.12 Procedures

Where a mapping can not be specified bidirectionally using equations or functions, it is necessary to describe the mapping procedurally. In VML the *map_to_from* predicate is used to denote two procedures which perform a mapping. In a *map_to_from* definition there is a procedure for mapping in each direction (if the mapping is one-way, i.e., *read_only*, then one of the procedures is not necessary and may be replaced with the function *true*). Dependent upon the direction in which a mapping is applied, the appropriate procedure will be invoked to perform its mapping. Procedures assume full control over how to perform a mapping, and as such may create objects, reference attributes, delete objects, etc. An example of the use of procedures is shown below.

map_to_from(map_3D_rect_to_polar(x, y, z, x1, y1, z1, min, max, plane),
map_polar_to_3D_rect(min, max, plane, pe_wall))

Where *map_3D_rect_to_polar()* and *map_polar_to_3D_rect()* are Prolog predicates that map from a 3D rectangular representation (given by two points in 3D space) to a polar representation, or vice versa. As procedures may need to manipulate objects in the stores they may require information about the current status of the mappings in the system and the store managers handling the objects in the system. To allow this information to be accessed in a procedure, a special parameter may be passed to the procedure. This parameter *$mapping_system$* is replaced with the object ID of the mapping system controller when a procedure is invoked. The three most utilised services offered by the mapping system are: return of the type of mapping being handled (currently just transaction-based or interactive); return of which pass through the mappings is being executed (for implemented systems where multiple passes are made during the mapping); and return of the object ID of an object that was created in a mapping based on another object ID (see describing pointer equality above). As procedures have full control over what they do in the system, they can perform mappings between any combination of Attr and Entity in the system, and can describe mappings between any cardinality of these Attr and Entity maps.

5.2.6.13 Method invocation

As VML is designed to work with OO environments as well as in interactive environments, it is necessary to handle OO method invocation. Methods are specified as an optional class or attribute definition, followed by the @ symbol and the name of the method, with parameters if they exist. Method handling in VML covers two cases: for classes which have creation methods with parameters, the initial parameters of the *create* call can be specified in the initialisers section of an *inter_class*; in a mapping where method invocation in one model can trigger a method invocation in the model being mapped to, the parameters of the methods to call can be specified. The examples below illustrate both these cases. The first example shows a case where whenever the *view_plane* method of an *idm_plane* object is called then the *create_view* method of the corresponding *fe_application* object in the mapped model should be called, or vice versa. The second example defines the parameters required in a create call for objects of type *fe_face_window*.

idm_plane@view_plane = fe_application@create_view(_, idm_plane.name)

fe_face_window@create(idm_building, idm_plane.name, idm_plane.axis, 0, '+', [])

Method parameters which return values not required for the mapping can be specified with a _ to indicate that the parameter should be ignored (as in *create_view* above). Methods are called if all parameters from the side being mapped from are known. This needs to be kept in mind when *create* methods are specified, as all the parameters from the side being mapped from must be guaranteed to be bound at the time that the object creation takes place. This also means that object references in a *create* method must be resolvable through pointer equality at the time of the object creation. For example, in the *create* method above *idm_building* will be replaced with the object ID created from the *inter_class* mapping for *idm_building*. The level of support for mapping of methods will be dependant upon the approach taken for the implementation, as methods can have

side-effects which are only valid if methods and data are mapped in a strict sequence.

5.2.6.14 Type conversion

Simple type conversions are implicit in a mapping definition. Casting of results of equations is not required in VML, as this is implicitly defined by the type of the attributes specified in the equation. Pointer equivalence handles the type conversion of object references, as well as lists and arrays of object references. Procedures are expected to correctly cast results calculated in the procedure when setting the value of an attribute. Mappings of attributes which have complex types are expected to explicitly define the type conversion as part of the mapping specification.

5.2.6.15 Unit conversion

Unit conversion is not supported implicitly in the VML language. Any unit conversion which must be performed between attributes must be modelled explicitly. While it would have been possible to assume an implicit unit conversion in a mapping there would have been some practical difficulties in the implementation. One difficulty is that implementing a system to determine the final unit of a complex equation is no simple task. Another problem is that predicate, procedure and method definitions provide no meta-information on the units of their outputs, so it would be impossible to check that the result of a predicate, procedure or method was in the correct units.

5.2.6.16 Temporary attributes

Temporary, or local, attributes can be defined in a VML mapping. These attributes have the same syntactic definition as a temporary entity, i.e., names prefixed either with an underscore symbol, e.g., _temp1, _temp2, or with a capital letter, e.g., Temp1, Temp2. Temporary attributes allow the specification of partial computations which may be used in further equations. To this extent temporary attributes can reduce the complexity of an equation definition and can be used to improve calculation performance in an implemented mapping system (as demonstrated in the example below).

```
WallTheta = tan_1(y_offset / x_offset),
WallDist = sqrt(x_offset * x_offset + y_offset * y_offset),
wall_x = WallDist * cos(WallTheta + azimuth),
wall_y = WallDist * sin(WallTheta + azimuth)
```

5.3 A Graphical Notation for VML

To satisfy the modelling notation requirements of Section 4.1.3, and to complement the textual notation of VML described in Sections 5.1 and 5.2, a graphical notation (VML-G) was developed. The graphical notation describes a subset of the full VML language and is aimed at high-level views of the mapping specification.

As a graphical language provides fast reading and comprehension of the textual equivalent, VML-G is likely to be of use in large integration projects where the schemas of the IDM and the design tools can be very large requiring hundreds of *inter_class* specifications to detail a full mapping. To manage a mapping specification of this size, the developers of integrated design systems will require many diagrams showing parts of a mapping, from high-level design views during the initial specification phase (which can identify entities which must be mapped between), through to more detailed descriptions of the mapping between attributes heading towards the implementation stage. Graphical specifications will also provide some benefit where single entities have a large number of attributes. In these cases the modeller may wish to consider subsets of the entities attributes when specifying a mapping, requiring multiple views of the mapping between entities.

5.3.1 Graphical icons of VML-G

In VML-G there is a single graphical icon type representing an *inter_class* definition (see middle icon in Figure 5.2). This icon has three sections corresponding to the three sections in an *inter_class* definition. These three sections allow invariants, equivalences and initialisers to be grouped into localised areas in the icon and provide a visual separation of these distinct functions. The other icon type defined in VML-G denotes an entity taking part in the mapping with the *inter_class* (see the left and right hand icons in Figure 5.2).

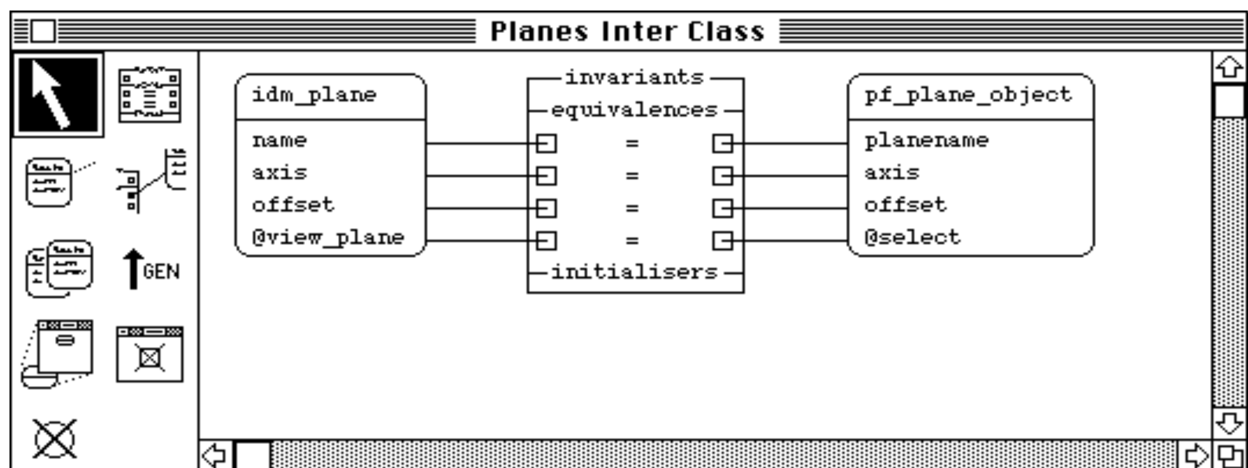

Figure 5.2 Graphical mapping specification in VPE

A graphical view of a mapping between entities from two schemas is specified by drawing an *inter_class* icon with icons for the entities on either side of it. By convention we put entity icons from the left hand schema on the left of the *inter_class* icon and the others on the right. This is not strictly necessary as the schema an entity belongs to can usually be ascertained quite simply. Entity icons specify the name of the entity represented (with optional schema and version information) and can list any attribute and method (prefixed with a '@') names defined in the entity (see Figure 5.2 for an example of direct attribute and method mappings). When there are multiple entities from a single schema participating in a mapping, the vertical order of the icons defines the ordering in the class list of the *inter_class* definition (Figure 5.3 shows the VML-G for such a mapping along with the textual equivalent). An entity which is to be grouped in the class list of the mapping

specification is drawn with a double line around the outside (like a stack of entities), also shown in Figure 5.3.

In Figure 5.2 we see that the icon for an *inter_class* definition consists of three parts. At the top is the invariants section which specifies conditions which must hold for this mapping to take place. Below that is the equivalences section which contains all mappings between attributes and entities. At the bottom is the initialisers section which holds the definition of initial values for attributes.

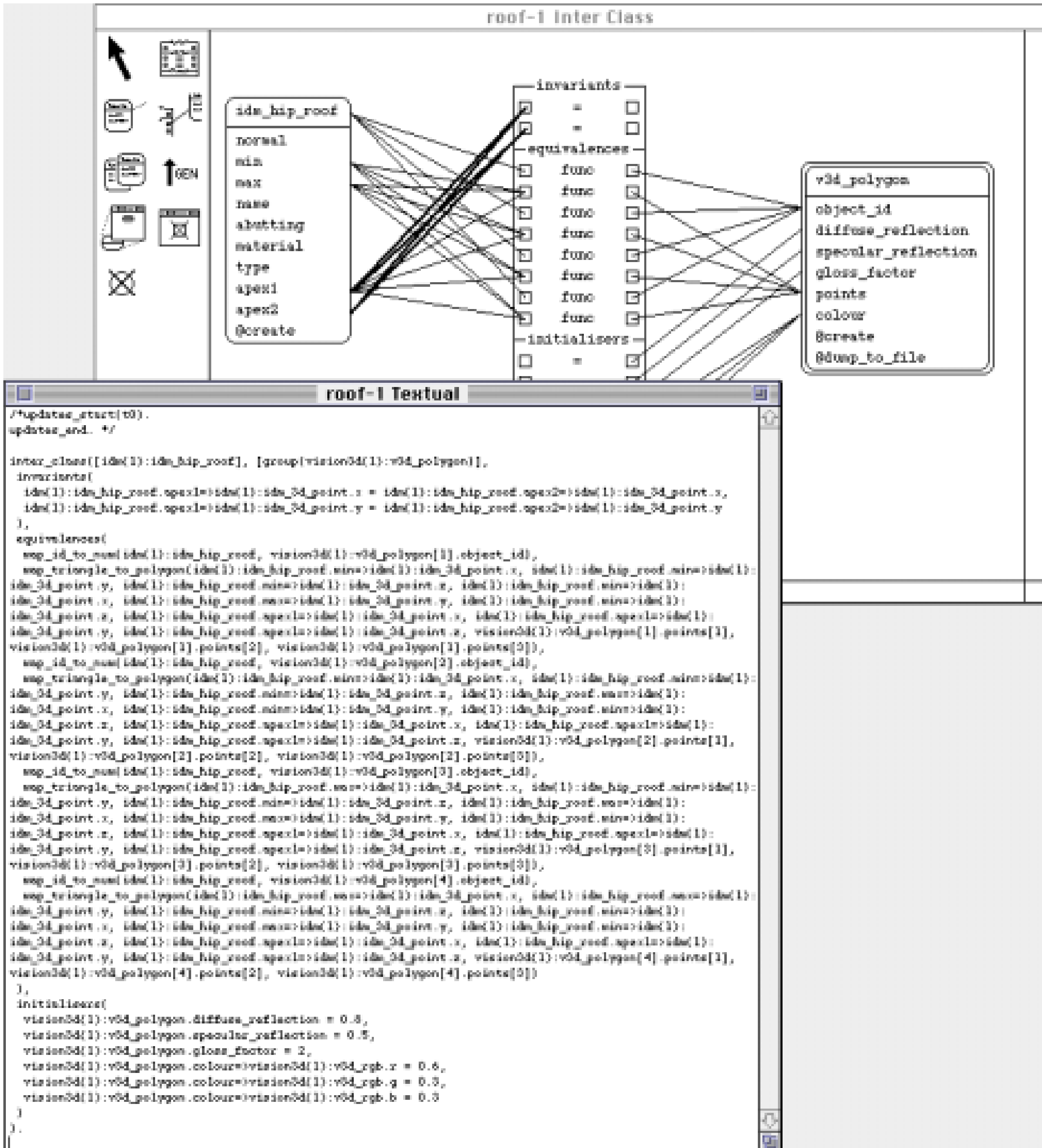

Figure 5.3 Complex VML-G specification with textual equivalent

Each individual equation, function, or procedure is given a single row in the *inter_class* icon. At each end of the row is a box to which the attributes and entities involved in the particular equation,

function, or procedure are connected. In the middle of the row there can be one of four symbols which are used to define the type of mapping being defined between attributes and entities. The four symbols are:

= denotes that there is a direct equivalence between the attribute (or entity) in one schema and the attribute (or entity) in the other. These one-to-one mappings are distinguished as a special case as they occur frequently in mapping specifications.

eqn denotes that the attributes or entities in one schema are related to attributes or entities in the other schema through an equation that is not a one-to-one equivalence.

func denotes that the attributes or entities wired together in this row are mapped through the use of a functional mapping specification.

proc denotes that procedural code is required to map between the specified attributes or entities. In the textual notation this has to be specified with different procedures to map in each direction.

5.4 Appraisal of VML

VML and VML-G define notations which are capable of modelling all known types of mappings between two schemas. They provide the ability to describe high-level views between entities in schemas, through to low level detailed views of mapping equations. They are supported by an environment (described in Chapter 6) which provides for multi-view graphical and textual specification, along with a global consistency mechanism. Mapping checkers and a generic mapping definition (also described in Chapter 6) provide a means to describe the required mappings between schemas for any implementation paradigm. In the remainder of this section, we review how these new notations meet the requirements laid out in Section 4.1, and what further work remains.

VML satisfies the requirements of Sections 4.1.1 and 4.1.2 in the following manner:

Structural mapping types: the class lists of an *inter_class* definition, along with invariant specifications, provide the means of describing any of the cardinalities of object to object mappings that can occur. Equivalence specifications provide a means of describing attribute to attribute mappings. In most cases, relationships between attributes and objects can also be defined in the equivalences section. The creation of objects during mapping initialisation (corresponding to 0:1 mappings) can be handled through an *inter_class* with one class defined in one class list and none in the other. Other types of new object or attribute creations are able to be specified through the initialisers section of the *inter_class* definition. VML has no explicit notion of object deletion or attribute value removal, therefore 1:0 mappings are not possible to encode explicitly. However, instead of explicit deletion, a notion of constrained creation is utilised. For example, if there is a condition under which an attribute should have no value then this would be specified in an invariant condition and

the following mapping would not map a value across.

Semantic mapping types: the majority of these conflicts can be resolved through explicit definition of mappings between attributes in schemas. For example, homonym and synonym conflicts from Table 4.3 can be resolved by specifying a mapping between the correct attributes. However, the recognition of these conflicts is the responsibility of the mapper and is not detected in the VPE support environment. The same is true for most of the schema and data conflicts of Table 4.4. Equivalences and initialisers provide the mechanism for resolving the conflicts; their recognition is the responsibility of the mapper. Structural conflicts of Table 4.3 are resolved through specification of classes in the class lists and the invariants which tie together objects of these classes only under specified conditions.

An appraisal of VML with regard to the more mapping-specific requirements of Section 4.1.3, yields the following:

Language level: the use of VML-G allows rapid and concise specification of which entities and attributes are related, without having to provide exact details of the connections. The declarative style of the VML notation allows for simple specification of exact relationships between entities and attributes, without having to detail specific methods to implement the relationships, and without having to rewrite the relationships to solve for all referenced attributes. In VML there is an underlying assumption that the mapping system will determine how to perform the mapping, and rearrange equations as required. However, it is recognised that a declarative approach can only be used for a certain set of correspondences, and in some cases a procedural specification will be necessary. VML provides for this as well. Some of the surveyed languages provide a more concise notation than VML for special cases (e.g., a Transformr 'COPY' to copy between identical objects during a version update). VML, however, provides a consistent level of specification for any type of mapping.

Language notation and modelling environment: VML is supported by a graphical notation (VML-G) to allow high-level views of mappings, as well as VML's lower level implementation views. The VPE environment demonstrates some of the power that can be gained from a multi-view graphical and textual specification environment with consistency maintained between all views. For example, select portions of a mapping can be described in different views, and entity icons can be expanded and contracted to show as much or as little of the entity interface as required.

Language style: the declarative style of VML ensures that it is easily translated into implementation environments of many paradigms. For example, translating VML into a procedural, batch operated mapping system would be possible by transforming VML mappings (re-arranged to solve for the required attributes) to procedural statements. This approach has been explored at BRANZ (Price 1995) to tie together several in-house design and analysis tools. As an example of the other implementation paradigm extreme, an interpreted on-line

implementation of VML mappings is illustrated in Chapter 10.

Bidirectional: all VML mappings are implicitly bidirectional, though this can be overridden in the *inter_view* definition. Any VML mapping is specified with the understanding that the implementation system will use the mapping specification, re-arranged as necessary to solve mappings, dependent on the direction in which data is mapped.

Conditional mapping: the invariants section of a VML mapping makes explicit the conditions under which a particular mapping can be utilised. These conditions are specified independently of the mapping definition.

Aggregation: functions to perform aggregation are included in the VML language. It is assumed that the implemented mapping system will determine which aggregate equations can be solved in both directions, and handle the consistency problems of those that can not.

Relationship handling: VML provides an operator to navigate the various structures found in different schemas. Through this operator, the compression of complex structures and the telescoping of simple structures can be defined.

Initialisers: the initialisers section of a VML mapping allows for the specification of initial values for entities and attributes, as well as specifying parameters for the creation of objects in an object-oriented environment. VML also allows the specification of an *inter_class* mapping with only one entity in one of the class lists as a way of defining objects which must be created at the start of a mapping process.

Unit handling: VML provides no implicit mapping for attributes of different units. Any unit conversions required between attributes must be identified by the mapper, and the conversion detailed as part of the mapping. This decision was made as a result of the difficulty of determining the resultant unit of the re-arrangement of any arbitrary equation, and the lack of unit information in the specification of functions, procedures and methods which can have serious flow-on effects (e.g., if a method returns a result used in a later calculation).

Type handling: simple type conversion is implicit in VML, so an implementation mapping system is required to implement type conversion for all simple types (e.g., float to integer). For simple types, the required conversion can be ascertained from the schema definition. Conversion of complex types must be detailed as part of the mapping.

VML provides the fundamental mapping representational requirement for schemas that need to share information with an IDM in an integrated design system. However, there are some aspects of the VML language which require further work:

Mapping language requirements: the list of mapping types around which the mapping language analysis is based is not known to be complete. The semantic mapping types are drawn from a slightly different domain and the mapping language requirements are drawn mainly from experience with mappings over the course of this project. For example, the requirement for bijections did not become clear until the large mapping problem illustrated in Appendix E was attempted. Early work on this thesis attempted to determine mapping

types from analysis and classification of known mapping problems based around the types specified in Section 4.1. However, further analysis of combinations of these types to determine high-level mapping types foundered in the combinatorial explosion of low-level combinations to be analysed. There appears to be no definitive classification of requirements in a mapping and further work looking at requirements would provide a sounder basis to compare mapping languages and evaluate their descriptive ability.

Greater control specification in mappings: the level of control specification in VML is too general. The *inter_view* specification is the only place where the level of mapping between schemas is defined. In the case where there is a *partial* mapping between the schemas there is no methodology available to determine how to calculate which part of which schema should be mapped. The underlying assumption is that a selection tool will be invoked at the start of the mapping, in order to select the subset of the model which will be mapped. It should be possible to specify the constraints on partial mappings in a more formal manner. There is also no control on individual *inter_class* specifications, so one-way mappings can not be explicitly defined on an *inter_class* by *inter_class* basis. However, this is not a problem for design tools (e.g., an IDM mapping to the output values of a design tool), as the project specification of Chapter 7 defines the input and output subschemas for all design tools in terms of the IDM's schema.

Greater micro-level control in an *inter_class* definition: Though the invariants section provides the conditions under which a mapping may take place, there are mappings which are almost identical, but which require separate *inter_class* definitions because a single equation or method call needs to be different. The main example of this problem is with create methods for objects, in these calls, where all parameters must be instantiated, it is necessary to supply a default value if an attribute is uninstantiated. However, with VML this requires two separate *inter_class* definitions, one which checks that the attribute is instantiated, and the other which checks that it isn't. Simple conditional wrappers around equations would provide a solution to most of these problems.

Generic function and procedure definition: though the majority of a VML specification is generic, and able to be mapped to almost any language paradigm, function and procedure definitions are not. Currently, functions and procedures are defined in Prolog and Snart, as these are the underlying languages in which the existing mapping implementation is written. It is unlikely that a totally generic procedure and function definition language could be defined, but work on the Neutral Model Format (Sahlin et al. 1995) and the Java language (Gosling and McGilton 1995) would suggest that some degree of generality could be supported.

Unit converter: VML makes the assumption that there is no equation re-arranger available that can calculate the final unit of any arbitrarily re-arranged equation. This is mainly due to the lack of unit information available from function, method and procedure definitions, which may calculate values for attributes used in later equations. Providing a notation for the specification of units on all parameters of functions, methods and procedures, and hence to

the design of a unit calculator for arbitrary equations, would prove an interesting challenge.

Explicit specification of type conversion: it would be useful to be able to define a mapping between complex types. This type mapping could be assumed to be used automatically, in the same way that simple types are assumed to be automatically converted. This functionality is available in mapping languages such as EXPRESS-M, and syntactic changes to VML to support the definition of type mappings would be trivial.

Method mapping: the VML specification of method invocation is limited to one-to-one correspondences. There is currently no way of specifying combinations of method invocations due to their temporal nature. For example, the set of method calls required to invoke a method call in the mapped to schema could be spread over several transactions. While data values are constant over multiple transactions, methods are only seen at the time that they occur. A more sophisticated conditional language specification would be required to fully model methods allowing combinations of several method calls to be required to invoke a mapping and also to enable mappings based on the parameters used when invoking the method. This would also impact on the type of system required to handle mappings between schemas as the whole history of the use of the particular schema would have to be available to be scrutinised.

In summary, this chapter introduces VML, a language that can be used to describe bidirectional mappings between two schemas. The language is shown to provide the functionality to meet the requirements for a mapping language. An environment which supports the specification of VML mappings is described in Chapter 6. The specification of mappings is the penultimate step in the specification of an integrated design system for a particular project. The final step is to model the project specification, which utilises the schemas for design functions from Chapter 3, and assumes that there is a mapping between the schemas for each design function and the IDM for the integrated design system. Chapter 7 introduces a formalism for project specification which details the flow of control between design functions to build upon the mappings to the IDM specified in this chapter.