

## Chapter 4

### Inter-Schema Relationship Modelling

The mapping of data between models is a vital process in an integrated design system. To enable the correct specification of a mapping between schemas, a mapping language and specification support environment is required. In a similar manner to the schema specification languages, a mapping language provides an abstract specification of a portion of the problem domain. The definition of a mapping between two schemas is likely to be a very large piece of work, often involving several domain experts for both models. These experts must maintain the consistency of the mapping specification. Because of this necessary involvement of domain experts, it is clear that a high-level specification language is needed to provide a specification close in semantics to the problem domain and well removed from the implementation. The mapping language should also support multiple levels of specification in a similar manner to schema modelling languages.

As well as providing the basis for mapping of data between them, a specification of the mapping between two schemas provides modelling benefits over those that are usually obtained just from the schema definition. For example, the specification of an inter-schema mapping makes explicit those constraints that are implicit in a design tool (e.g., a design tool which assumes vertical walls needs to explicitly model this requirement when specifying a mapping with a schema that does not have this assumption).

If a formal mapping language is used to specify the mappings, mapping specification environments offering consistency management between various modellers are possible. Tools can then be created to: check statically whether the mapping references valid schema properties; check types in the conversion; check units in the conversion; identify affected mappings when components in the schema are modified; and ensure that all attributes of entities are mapped between the two schemas.

However, until recently, the mapping between the IDM and a design tool schema has not been modelled in any way in integration projects. All mappings for design tool schemas have typically been hand-coded in the programming language of the project. This approach is now changing as several projects have reached the stage of development where they recognise the utility of having mapping languages to formally model mappings between schemas (ATLAS 1993; Staub et al. 1994). To satisfy the demand for mapping languages a number of new languages have been developed.

In this chapter the types of mappings which are required to describe correspondences between schemas are examined. Languages for specifying mappings are surveyed, including formalisms recently developed for mapping in the ISO-STEP standard. An informal set of requirements for a mapping language is collated from previous work and from an analysis of mapping requirements for a range of actual schema. These requirements are then used to measure the power of the surveyed languages, and enable comparisons between the languages to be performed.

## 4.1 Mapping Types

Analyses of the problems posed by mappings and integration have previously been conducted in the database area. To gain an understanding of what is required to map between two schemas, the analyses performed in the field of database integration are examined and presented here in two categories. One category investigates the types of structural mapping that can be expected between two schemas, the other presents a more semantic description of the types of conflict found when integrating schemas.

### 4.1.1 Structural mapping types

Structural mapping types provide a key to the complexity of mappings that can be expected in a particular domain. Two evaluations of these type of mapping are those of van Horssen et al. (1994), which details a table of mapping types (see Table 4.1), and Bijnen (1994), which provides a more general specification of mapping types. Both of these evaluations are drawn from considerations of requirements of a mapping language: in van Horssen's case as a precursor to evaluating several mapping languages; and in Bijnen's case to show what his mapping language should provide. In Table 4.1 the asterisks denote the requirement for mappings of the specified type, blank cells indicate no mappings of the specified type are required. The labels on the table below are slightly misleading, as where they specify *entity* they actually mean *object*. For example an N:1 mapping for Entity->Entity denotes the mapping of N objects of one entity type to a single object of another entity type.

	Entity->Entity	Attr->Attr	Entity->Attr	Attr->Entity
1:0	*	*	*	*
0:1	*	*	*	*
1:1	*	*	*	*
1:C	*	*	*	*
C:1	*	*	*	*
1:N	*			*
N:1	*		*	
N:M				

**Table 4.1** Mapping types from van Horssen et al. 1994, the left axis specifies the cardinality of mappings which may be required

In Table 4.1 the cardinalities of the various types of mapping are considered with a distinction being made between zero (0), singular (1), constant (C) and variable numbers (N, M) in the mapping. The lack of an asterisk for the many to many mapping for entities in this table is surprising as some are known to exist (for example some design tools use a set of objects to represent a schedule, with each object specifying a fixed number of hour-value pairs. Where a mapping is required between two such tools which have different numbers of hour-value pairs in each object there can be different numbers of objects in each model, hence an N:M mapping). It is also surprising that all combinations of constant values (C in Table 4.1) are not included. If the breakdown includes 1:C, etc. then a rigorous evaluation should also evaluate N:C, etc., though 0:C is not required as it can be constructed through multiple 0:1 mappings.

	Object->Object	Attr->Attr	Object->Attr	Attr->Object
1:0	*	*	*	*
0:1	*	*	*	*
1:1	*	*	*	*
1:C	*	*	*	*
C:1	*	*	*	*
C:B	*	*	*	*
1:N	*			*
N:1	*		*	
C:N	*			*
N:C	*		*	
N:M	*			

**Table 4.2** Full set of structural mapping types

Following the examination of structural mapping types above, the table in Table 4.1 has been extended to provide a more rigorous definition of the possible combinations of mapping cardinalities that can occur. This is shown in Table 4.2, where all combinations of constant to multiple, and constant to constant (C:B) are provided.

### 4.1.2 Semantic mapping types

The semantics of individual mappings have also been examined. Tables 4.3 (Batini et al. 1986) and 4.4 (Kim and Seo 1991) are examples from their analysis of the schema integration process which examines various categorisations for integration. While these tables display conflict types found when integrating schemas, most of them hold equally well for mappings. This is because the definition of a mapping between two schemas can be seen as almost the same task as specifying the integration of one schema into another. The main difference is that in a mapping the schemas being mapped between do not have to be completely unified. The mapping need only be sufficient to create valid instances of a building in either schema.

<b>Naming conflicts</b>	
homonyms	same name for different concepts
synonyms	same concept described by different names
<b>Structural conflicts</b>	
type conflicts	same concept represented by different modelling constructs
dependency conflicts	group of concepts are related with different dependencies in different schemas, e.g., 1-1 versus n-m
key conflicts	different keys assigned to the same concept in different schemas
behavioral conflicts	different insertion or deletion policies associated with the same class of object in different schemas
<b>Conflict categories</b>	
identical	everything is the same.
equivalent	where different but equivalent modelling constructs have been applied but the perceptions are still the same and are coherent. These are further subdivided into:
	behavioral if the same set of answers to any given query can be obtained from all representations.
	mapping instances can be put on a one-to-one correspondence.
	transformational if a representation can be obtained by applying a set of atomic transformations that by definition preserve equivalence.
compatible	not identical or equivalent, but modelling constructs, designer perception and integrity constraints are not contradictory.
incompatible	contradictory because of the incoherence of the specification.

**Table 4.3** Schema integration conflict types from Batini et al. 1986

Tables 4.3 and 4.4 provide a checklist of problems that could be encountered and a means of categorising the difficulty of a particular problem. They do not, however, provide many clues as to what would be required in a mapping language to handle these types of problems. Some of the interesting points which come out of these two categorisations are: the need to update schemas for missing attributes and entities; the handling of unit conversion; and the handling of type conversion.

<p><b>Schema conflicts</b></p> <p>Table versus table conflicts</p> <ul style="list-style-type: none"> <li>one-to-one table conflicts <ul style="list-style-type: none"> <li>table name conflicts <ul style="list-style-type: none"> <li>different names for equivalent tables</li> <li>same name for different tables</li> </ul> </li> <li>table structure conflicts <ul style="list-style-type: none"> <li>missing attributes</li> <li>missing but implicit attributes</li> </ul> </li> <li>table constraint conflicts (keys and check conditions)</li> </ul> </li> <li>many-to-many table conflicts (as in one-to-one)</li> </ul> <p>Attribute versus attribute conflicts</p> <ul style="list-style-type: none"> <li>one-to-one attribute conflicts <ul style="list-style-type: none"> <li>attribute name conflicts <ul style="list-style-type: none"> <li>different names for equivalent attributes</li> <li>same name for different attributes</li> </ul> </li> <li>default value conflicts</li> <li>attribute constraint conflicts <ul style="list-style-type: none"> <li>data type conflicts</li> <li>attribute integrity-constraint conflicts</li> </ul> </li> </ul> </li> <li>many-to-many attribute conflicts (as in one-to-one)</li> </ul> <p>Table versus attribute conflicts</p> <p><b>Data conflicts</b></p> <p>Wrong data</p> <ul style="list-style-type: none"> <li>incorrect-entry data</li> <li>obsolete data</li> </ul> <p>Different representations for the same data or same representation for different data</p> <ul style="list-style-type: none"> <li>different expressions</li> <li>different units</li> <li>different precisions</li> </ul>
---

**Table 4.4** Schema integration conflict types from Kim and Seo 1991

### 4.1.3 Mapping language requirements

An examination of the mappings required between the design tool schemas used in this thesis, based on the classifications in Sections 4.1.1 and 4.1.2, leads to a range of pragmatic requirements for an ideal mapping language:

Language level: To enable rapid and concise specification, a language's notation should closely mirror the domain in which it is being used. A language to describe a mapping needs to be able to represent relationships between entities, attributes, references, and, in an object-oriented environment, methods, using a high-level notation. Using low-level notations will require the mapping specifier to concentrate on the practicalities of how to implement the mapping rather than on the specification of the actual mapping.

Language notation and modelling environment: The bulk of a mapping specification is concerned with relationships between attributes. Experience shows that many of these relationships need to be described in an equational form which is basically textual in nature. This tends to indicate that mapping languages will be based around a textual notation. However, given the potentially large size of the schemas involved in a mapping, a graphical notation allowing high-level views of the mapping will prove of benefit in the early definition of the mapping. A graphical notation for high-level views is likely to provide faster cognition of

relationships between entities in the schemas, in contrast to reading a purely textual specification, as relationships will be explicitly depicted rather than having to be determined by the user parsing a textual expression. A graphical notation supported by a modelling environment is also likely to prove a useful checking tool in ensuring that mappings are completely specified for particular entities. This is because a graphical notation will show the entities and their attributes in the same view as the mapping, allowing easy determination of what has or has not been mapped. As some of the class definitions are large and contain various types of data (attributes, relationships and methods), it will be useful to allow several views of a mapping specification, with different views concentrating on different aspects of the mapping. The specification of a mapping is also likely to require a modelling tool of a similar magnitude to that used for managing schema definition to maintain consistency between various mapping specifiers as well as between various views of the mappings.

**Language style:** A mapping language needs to support many levels of correspondence specification, from simple equivalences between attributes through to complicated programs to extract and manipulate data into the required form. The language should not presuppose a single style of implementation (e.g., batch mode; full model translation; or automatic incremental mapping). Rather, any implementation style should be able to be implemented from the specified mapping. This requirement would tend to favour a declarative style of mapping specification over a procedural style, as a procedural style is less amenable to many implementation styles (e.g., to an incremental update model). A declarative notation is likely to provide the highest level of specification and most closely satisfy the modelling level requirement defined above. This is because declarative languages do not specify a particular method for performing a certain function, but specify more what is required. Although a declarative approach may provide the highest level of definition, it is recognised that not all mappings will be able to be specified in a declarative manner. A procedural form may also need to be supported.

**Bidirectionality:** Many connections between tools require the same structures to be mapped in both directions. Where bidirectional mappings are required, the mapping language should support their specification without the need to duplicate information on the correspondences. Where only unidirectional mappings are required this should also be able to be specified.

**Conditional mapping:** Dependent upon the state of a model, or portions of a model, the data in the model may need to be mapped in different ways. To enable this in a mapping, it must be possible to specify conditions which must be satisfied prior to the application of a particular type of mapping. Such conditions provide one way of making explicit assumptions that are only implicitly specified in a schema.

**Aggregation:** The level of detail used to represent entities in a schema will vary enormously. When mapping between schemas with very different levels of detail it is necessary to aggregate information in very detailed schemas to fill higher level (more abstract or less detailed)

schemas. The reverse process will also be required, but it is often impossible to do this unambiguously (e.g., given a total glazing area it may be impossible to work out the area of each individual window). In this case it may be possible to constrain the constituents in the detailed model to match the values of the aggregated model.

**Relationship handling:** Similar to the problem of various aggregation levels is the problem of relationship structures. Different schemas of a domain are likely to choose different structures to represent the relationship between entities in their schemas. A mapping language needs to be able to restructure relationships in a schema, compressing long pointer chains, telescoping down into deep structures, and moving relationships between entities in different schemas.

**Initialisers:** During the mapping process new objects must be created and, in some cases, initial values set for attributes. Having a method to specify initial values independent of mappings, which may calculate values for these attributes as data becomes available, is important. This provides another way of making explicit assumptions that are only implicitly specified in a schema. It also provides a mechanism to ensure that a minimum set of data is created in a mapping, for example, to ensure that a design tool could run after any amount of data is transferred through in a mapping.

**Unit handling:** Attributes in different schemas often use different units to represent their quantities. Whether due to the country in which the schema was developed (i.e., imperial or metric units), the equations used, or just the magnitude of the result presented to the user, the ability to convert attributes between different units is required of a mapping language.

**Type handling:** Different schemas may use different precisions of types to represent their values, depending upon the accuracy required or the time and space limitations in the calculations. Different schemas may also use different structures to represent the same information (e.g., tree versus list) and a mapping language needs to map between the various types to ensure the model is in a valid state after a mapping has occurred.

## **4.2 Mapping Definition Languages**

A wide range of languages are being developed (or have been developed over the last few years) to specify the mapping between schemas, or to enable the integration of schemas, or to provide a view of a schema. In this section, a representative set of these languages is examined to highlight the styles available and their expressive power.

To help illustrate the style and form of these languages, an example from a survey of mapping languages (Verhoef, Liebich and Amor 1995) will be used. This is one of five example mappings used in the paper. The mapping is between two schemas drawn from existing applications requiring data transfer in an integrated environment. The schema fragments for this example are shown in Table 4.5.

The schema fragment shown on the left-hand side describes building components which are either structural (i.e., columns or beams) or a connector between structural components. The relationship component has two forms, defined by the *quality* attribute, which are either support or element connections. The right-hand side schema fragment describes structural components, one specialisation of which is a structural connector. This structural connector has two specialisations of either a support or element connector. The mapping problem is to map between the component relationship in the left-hand side schema and either the support or element connector in the right-hand side schema. Performing the mapping in this example illustrates: a conditional mapping between entities; a type conversion (rather contrived) between two attributes; and attribute mappings which require the results of other mappings between entities.

<pre> TYPE    connection_type =     ENUMERATION OF         ( support_connection,           element_connection ) ; END_TYPE ;  ENTITY building_component     ABSTRACT SUPERTYPE OF ( ONEOF         ( structural_component,           component_relationship ) ) ;     id          : REAL ;     UNIQUE ul   : id ; END_ENTITY ;  ENTITY structural_component     ABSTRACT SUPERTYPE OF ( ONEOF         ( column, beam ) )     SUBTYPE OF ( building_component ) ;     specified_by : SET [1:?] OF         product_characteristic ;     represented_by :         geometric_representation_item ; END_ENTITY ;  ENTITY component_relationship     SUBTYPE OF ( building_component ) ;     related      : structural_component ;     relating     : structural_component ;     quality      : connection_type ; END_ENTITY ; </pre>	<pre> TYPE    support_connection = ENUMERATION OF     ( free_support, restrained_support,       un_known ) ; END_TYPE ;  TYPE    element_connection = ENUMERATION OF     ( joint_connection, rigid_connection,       un_known ) ; END_TYPE ;  ENTITY structural_component     ABSTRACT SUPERTYPE OF ( ONEOF         ( structural_assembly,           structural_element,           structural_connector ) ) ;     identified_by      : INTEGER ;     UNIQUE ul         : identified_by ; END_ENTITY ;  ENTITY structural_connector     ABSTRACT SUPERTYPE OF ( ONEOF         ( support_connector,           element_connector ) )     SUBTYPE OF ( structural_component ) ;     related, relating : structural_element ; END_ENTITY ;  ENTITY support_connector     SUBTYPE OF ( structural_connector ) ;     type_of      : support_connection ; END_ENTITY ;  ENTITY element_connector     SUBTYPE OF ( structural_connector ) ;     type_of      : element_connection ; END_ENTITY ; </pre>
--	--

**Table 4.5** Schema fragments for the two schemas in the mapping example

### 4.2.1 EXPRESS-M

EXPRESS-M (Bailey 1994) is an evolving language being developed to solve the problem of application protocol inter-operability in the STEP standard. As the language is intended for use in STEP it has been designed to look very similar to the EXPRESS language. EXPRESS-M mappings are unidirectional and map a whole model at a time (no partial updates of models). The

EXPRESS-M language has the following major components:

**SCHEMA\_MAP:** specifies the EXPRESS schemas which are the source for the mapping and the schemas which are targets for the mapping. In most cases there is a one-to-one mapping from a single source schema to a single target schema.

**MAP:** specifies the mapping between entities in the source schema(s) and entities in the target schema(s). There can be only one MAP for a particular target entity so all conditional clauses for a mapping must be handled inside one MAP. The actual mapping of data between attributes is done through assignment statements with a large range of functions available to calculate the value to assign. Iteration constructs may be used to specify values for aggregate attributes, and user defined functions are accessible in the mapping. Type conversion is specified in the mapping by means of casting from the type of the source value to that required in the target. All conditional mapping, including determination of which type of entity to create as well as what equations to apply to calculate a value for an attribute, must be handled in the single MAP definition.

**TYPE\_MAP:** specifies the mapping required to instantiate an attribute of one type from an attribute of a second type. This is for non-simple types (simple types have a default casting regime) and can handle enumerations, lists, sets, bags, and other structures. The TYPE\_MAP is also used to map between attributes of differing units. In EXPRESS there is an overlap between the unit and type specification of an attribute which can lead to the case of two attributes in two schemas being of type 'litre' with one being a real number while the other is an integer. TYPE\_MAP allows these problems to be tackled, as well as handling more usual type mapping.

```
MAP ONEOF(support_connector, element_connector) <- component_relationship;
  IF quality = support_connection THEN
    MAP support_connector <- component_relationship;
      identified_by := {INTEGER}id;
      related := {structural_element}related;
      relating := {structural_element}relating;
      type_of := un_known;
    END_MAP;
  ELSE
    MAP element_connector <- component_relationship;
      identified_by := {INTEGER}id;
      related := {structural_element}related;
      relating := {structural_element}relating;
      type_of := un_known;
    END_MAP;
  END_IF;
END_MAP;
```

**Table 4.6** EXPRESS-M mapping for example problem

**PRUNE:** specifies entities which might be created twice in a SCHEMA\_MAP and which should be culled to one instance. Multiple instances of an entity occur when there is a MAP for an entity and a reference to the entity attributes from an associated entity (i.e., the entity is also created by reference). EXPRESS-M is able to determine what to prune by monitoring separately those instances created in a MAP and those created by reference.

Manual entity instantiation: instances of entities can be created without a mapping from source entities through a simple manual instantiation specification which follows that used in STEP exchange files.

Table 4.6 shows the EXPRESS-M mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as EXPRESS-M is a unidirectional mapping language. The top level mapping specification specifies that a *component\_relationship* is mapped to either a *support\_connector* or an *element\_connector*. The conditions which decide which type of mapping is performed are specified in the if-then-else statement. Type conversion must be specified explicitly when required as seen with the casting of *id*, which is a real, to an integer, as well as the *related* and *relating* object references.

Some of the drawbacks of EXPRESS-M are:

- mapping specifications are unidirectional, therefore it requires two mapping specifications to map data in both directions between models. For schemas where a bidirectional mapping is required, having two separate mappings which are inverses of each other is problematic. Apart from the extra time required to create the two mappings, it provides more chance for inconsistencies between the mapping specifications as schemas change and mappings are refined.
- only one MAP can exist for a particular entity combination. This can make the conditional mapping specification very convoluted and difficult to decipher. Table 4.6, where there are two mapping types to choose between, shows this in a small way.
- although parts of the language are declarative there are many procedural components (all handling of aggregate components) which interfere with the readability of mapping specifications.
- no graphical formalism is offered with the textual notation, allowing only the very low level implementation view of any mapping to be manipulated.
- the ordering imposed on evaluation (i.e., as it appears in the mapping) means very careful consideration of ordering is required when specifying mappings.

#### 4.2.2 EXPRESS-V

EXPRESS-V (Hardwick et al. 1994, Hardwick 1994) is similar to EXPRESS-M in approach but with a limited ability to support bidirectional mapping between models. EXPRESS-V is intended to be an extension to EXPRESS to support database views in an ISO-STEP environment. EXPRESS-V definitions are specified along with EXPRESS definitions for a schema. The major components of EXPRESS-V are:

VIEW: specifies that an entity or entities are to be viewed as another entity. This can be a conditional specification using a WHEN clause to specify the conditions under which this VIEW may hold true. The actual mapping of data between attributes is done either in a VIEW\_ASSIGN section, to map from source to target, or in an UPDATE section, to map

from target to source. Mappings are performed through an assignment statement with a large range of functions available to calculate the value to assign.

**VIEW\_ASSIGN:** is defined in the VIEW clause and specifies what attributes in the target schema need to be updated when source entities are modified. The VIEW\_ASSIGN section uses equations to perform the mapping of attributes in the source entities as described in the VIEW section. The VIEW\_ASSIGN can be conditional through the use of a WHEN clause. There can be multiple VIEW\_ASSIGN clauses in a VIEW specification.

**UPDATE:** is defined in the VIEW clause and specifies what attributes in the source schema need to be updated when target entities are modified. The UPDATE section uses equations to perform the mapping of attributes in the target entities as described in the VIEW section. The UPDATE can be conditional through the use of a WHEN clause. There can be multiple UPDATE clauses in a VIEW specification.

**CREATE:** is defined in the VIEW clause and specifies attribute values for source entities which have to be created by the creation of target entities. The CREATE can be conditional through the use of a WHEN clause. There can be multiple CREATE clauses in a VIEW specification.

**DELETE:** is defined in the VIEW clause and specifies which attributes and objects need to be deleted in the source entities upon the deletion of a target object. The DELETE can be conditional through the use of a WHEN clause. There can be multiple DELETE clauses in a VIEW specification.

```
VIEW support_connector
FROM (component_relationship)
WHEN (component_relationship.quality = 'support_connection');
VIEW_ASSIGN
    identified_by := component_relationship.id;
    type_of := 'un_known';
    related := component_relationship.related;
    relating := component_relationship.relying;
UPDATE
    id := support_connector.identified_by;
    related := support_connector.related;
    relating := support_connector.relying;
    quality := 'support_connection';
END_VIEW;

VIEW element_connector
FROM (component_relationship)
WHEN (component_relationship.quality = 'element_connection');
VIEW_ASSIGN
    identified_by := component_relationship.id;
    type_of := 'un_known';
    related := component_relationship.related;
    relating := component_relationship.relying;
UPDATE
    id := element_connector.identified_by;
    related := element_connector.related;
    relating := element_connector.relying;
    quality := 'element_connector';
END_VIEW;
```

**Table 4.7** EXPRESS-V mapping for example problem

Table 4.7 shows the EXPRESS-V mapping for the example in Table 4.5. This example highlights the relational database approach underlying the language definition. New views must be defined for each class to be mapped, in this case views of a *component\_relationship* as either a *support\_connector* or an *element\_connector*. The WHEN statement provides the conditions under which the view can be supported. The VIEW\_ASSIGN statements describe the mappings required, and include implicit type conversion. The UPDATE section describes what can be modified in the original model when a view element is modified, this highlights a limitation of the language as it assumes one of the schemas to be a master schema from which views are created. In effect this limits views to what can be derived from one schema and does not allow creation of new objects from the view schema.

Some of the drawbacks of EXPRESS-V are:

- the mapping specification is associated with a particular schema (as in standard RDBMS views) and does not directly specify the other schema being accessed (although it is specified in a USES clause). This could make a schema specification for an IDM very convoluted as it could contain all mappings to design tools along with the schema definition. This approach also requires a flat name-space, as all entity definitions in all views are visible at the same time. This is an unreal expectation in a situation where existing applications (with predefined entity definitions) are to be integrated with an IDM.
- although the language allows for bidirectional mappings (though this is through separate VIEW\_ASSIGN and UPDATE sections in the VIEW definition), the mapping in each direction needs to be specified independently, due to the procedural specification of a mapping. Although this allows the mapping specifications to be described at one point, having a section for each direction leads to the duplication of information in the mapping specification.
- the language assumes the source schema is always much more sophisticated than the target schema as CREATE and DELETE blocks are only available for mappings in one direction. Although in a generalised integrated design system it is likely that some target schemas would require CREATE and DELETE blocks as well.
- no graphical formalism is offered with the textual notation, allowing only the very low level implementation view of any mapping to be manipulated.

### 4.2.3 EXPRESS-C

EXPRESS-C (Staub et al. 1994) was developed to extend and enhance the capabilities of EXPRESS by enabling the modelling of both static and dynamic properties of a domain. It is considered as a first step towards a fully object-oriented version of EXPRESS as was suggested within the EXPRESS v2.0 development targets. The major mapping component of EXPRESS-C is:

TRANSACTION: a named transaction can be used to specify a unidirectional mapping between sets of objects accessed from the current model. As transactions describe a procedural

mapping between their referenced objects, two transactions would be required to describe a bidirectional mapping.

```

TRANSACTION t_map_component_relationship;
LOCAL
    socr : SET OF component_relationship;
    sosc : SET OF structural_connector := [];
END_LOCAL;
    socr := POPULATION('BSSC.COMPONENT_RELATIONSHIP');
    REPEAT i := 1 TO HIINDEX(socr);
        sosc := sosc + map_component_relationship(socr[i]);
    END_REPEAT;
END_TRANSACTION;

FUNCTION map_component_relationship
    (cr : component_relationship) : structural_connector;
LOCAL
    sc : structural_connector;
END_LOCAL;
    IF (cr.quality = support_connection) THEN
        sc := compare (support_connector(support_connection.un_known) ||
            structural_connector (map_structural_component(cr.related),
            map_structural_component(cr.relater)) || structural_component(cr.id));
    ELSE
        sc := compare (element_connector(element_connection.un_known) ||
            structural_connector (map_structural_component(cr.related),
            map_structural_component(cr.relater)) || structural_component(cr.id));
    END_IF;
    make_instances_persistent([sc]);
    RETURN(sc);
END_FUNCTION;

```

**Table 4.8** EXPRESS-C mapping for example problem

Table 4.8 shows the EXPRESS-C mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as EXPRESS-C is a unidirectional mapping language. The transaction specification shows the very low level implementational approach taken with this language. The transaction definition identifies all *component\_relationship* objects in a model and creates the corresponding set of mapped objects using the function *map\_component\_relationship()*. This function utilises standard EXPRESS statements to describe a conditional creation of objects and calls other functions to map the two object references.

EXPRESS-C has the same set of drawbacks as described in EXPRESS-V along with the lack of explicit support for bidirectional mappings.

#### 4.2.4 Transformr

Transformr (Clark 1992) was designed to be able to propagate instances between different versions of a model. As with EXPRESS-M and EXPRESS-V, Transformr was developed for use in the ISO-STEP environment. The major components of Transformr are:

**COPY:** specifies the copying of an entity to another entity. In its simplest form it simply names an entity and all attributes of this entity are copied across. COPY can move all objects between entities with different names, or, through the use of derived attributes, add, or drop attributes from the new entity.

**BUILD:** specifies the creation of a new entity from a set of entities in the source. This is a conditional creation, where all attributes that need to appear in the target entities must be described, unlike **COPY** which maps everything unless the user specifies otherwise.

```

BUILD support_connector FROM component_relationship
  WHERE
    component_relationship.quality = support_connection;
  DERIVE
    identified_by := REAL_TO_INT(component_relationship.id);
    related := component_relationship.related;
    relating := component_relationship.relying;
    type_of := un_known;

BUILD element_connector FROM component_relationship
  WHERE
    component_relationship.quality = element_connection;
  DERIVE
    identified_by := REAL_TO_INT(component_relationship.id);
    related := component_relationship.related;
    relating := component_relationship.relying;
    type_of := un_known;

```

**Table 4.9** Transformr mapping for example problem

Table 4.9 shows the Transformr mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as Transformr is a unidirectional mapping language. Separate BUILD statements are used for each type of object to be created with the WHERE statement defining the condition under which the BUILD can be used. Mappings in the DERIVE section are simple statements, though explicit type conversion through user defined functions is required for simple types, as shown with the mapping of *id*.

Transformr is a simple language which has an elegant style and appears well suited to the process of propagating data between versions of the same schema. It is, however, very much a unidirectional mapping language, with limited functionality in terms of the combinations of entities which can be clustered and created. Transformr is also limited in the types of equivalences that can be specified between attributes.

#### 4.2.5 EDM-2

EDM-2 is a novel language, database system and environment developed at UCLA for use in the A/E/C domains (Eastman et al. 1995). EDM-2 incorporates three major features not found in traditional database systems, but which are of key importance in the development of an integrated design environment. These are:

**Dynamic schema specification and evolution:** a central schema in an integrated environment can be modified at any time to take account of new applications to be utilised in the system, or to incorporate new views of the existing schema for specific user needs.

**In-built integrity management:** through the use of constraints specified in the schema, or defined “on the fly” as a model is developed, the integrity of a model can be determined at any stage. Constraints with parameters which have been modified can be rechecked at any

time, and in this manner the global consistency of a model can be monitored as a design progresses.

Explicit translation definition: high-level support for translation between applications (bidirectionally) is incorporated into EDM-2.

```
CREATE DE bssc_component_relation KEYNAME
    DESC "BSSC component_relation class";

CREATE DE pss_structural_connector KEYNAME
    DESC "PSS structural_connector class";

CREATE DE gen_part KEYNAME
    ATTR(bssc: bssc_component_relation, pss: pss_structural_connector)
    DESC "Generalized beam class";

CREATE MAP components
    (bssc_component_relation)
    RETURN (pss_structural_connector)
    IMPL $MAP_METHODS/components.so
    DESC "Mapping from BSSC model to PSS model";

CREATE MAPCALL component_mapping
    MAP components
    (bssc)
    RETURN (pss)
    REF gen_part
    DESC "Map call with reference to generalized object";
```

**Table 4.10** EDM-2 mapping for example problem

The definition of mapping is supported by two constructs in the EDM-2 language:

MAP: defines a process through which entities of a particular type can be translated from one type to the other.

MAPCALL: describes the use of a MAP for particular instances in the model.

Table 4.10 shows the EDM-2 mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema. The MAP definition describes the inputs and outputs of the mapping whose implementation is hidden in the file referenced by the IMPL statement (the C code of this function is not shown here for brevity). The MAPCALL definition lists the MAP statements which are to be used when mapping from one schema model to another, in this case from *bssc* to *pss*.

Mapping constructs have only recently been added into EDM-2, so details of their incorporation are still under review. Currently the maps provide unidirectional mapping between entities, with the maps associated with the central schema (though separate schemas tend to be merged in the EDM-2 implementation). Mapping definitions are not seen explicitly in the mapping definition, as currently the implementation is as C programs which are invisible to the user browsing the schema.

## 4.2.6 KIF

KIF (The Knowledge Interchange Format: Genesereth and Fikes 1992; Khedro et al. 1994) was originally developed in the ARPA knowledge sharing initiative as a means to exchange information between applications. KIF provides a mechanism for agents to communicate messages to inter-operating applications. In KIF each agent has the responsibility of translating messages received from other agents from their native format to the format required locally. To achieve this, each agent must define a translation for messages they wish to handle. To use KIF in a standard IDM-type system would require an IDM where translations for every attached schema are written into the IDM. Each application would require translations from the IDM structures into their own internal format. This places the onus of translation on every application working in the integrated system. However, it also allows for a system where individual modifications can be propagated to all interested applications as they occur and where incremental consistency of the whole integrated system can be maintained. The main drawback of KIF is the requirement that each application be aware of the IDM and be able to translate information from the IDM whilst performing its own application tasks.

```
(<= (pss!support_connector ?ent)
    (bssc!component_relationship ?ent)
    (= (bssc!component_relationship.quality ?ent) support_connection))
(<= (pss!element_connector ?ent)
    (bssc!component_relationship ?ent)
    (= (bssc!component_relationship.quality ?ent) element_connection))
(<= (= (pss!support_connector.identified_by ?ent) ?id)
    (= (bssc!component_relationship.id ?ent) ?id)
    (= (bssc!component_relationship.quality ?ent) support_connection))
(<= (= (pss!element_connector.identified_by ?ent) ?id)
    (= (bssc!component_relationship.id ?ent) ?id)
    (= (bssc!component_relationship.quality ?ent) element_connection))
(<= (= (pss!support_connector.type_of ?ent) ?type)
    (= (bssc!component_relationship.quality ?ent) ?type)
    (= ?type support_connection))
(<= (= (pss!element_connector.type_of ?ent) ?type)
    (= (bssc!component_relationship.quality ?ent) ?type)
    (= ?type element_connection))
```

**Table 4.11** KIF mapping for example problem

Table 4.11 shows the KIF mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as KIF is a unidirectional mapping language. The mapping reflects the blackboard architecture that KIF is implemented upon, with every item mapped individually. The first two definitions create either a *support\_connector* or an *element\_connector* depending upon the value of the *quality* attribute. The second two definitions create the *identified\_by* attribute for the objects and the last two create the *type\_of* attribute for the objects. The mapping of object references (i.e., *related* and *relating*) are not attempted in this mapping.

## 4.2.7 Superviews

Superviews (Motro 1987) describes a method for the virtual integration of multiple databases. The method is founded on a set of ten operators with which the integrator defines the requisite transformations to the base schemas to produce the superview. The product of this method is a

superview of the base schemas and the set of mappings (reversible) required to move information back and forth between the superview and base schemas. The ten operations available are:

**Meet:** produces a common generalisation of two entities. This is only possible when two entities have a common key. A *meet* also introduces a consistency constraint, that values in both entities with the same key must agree on shared attributes.

**Join:** the resultant type is the union of the types of two entities.

**Fold:** allows a generalisation to absorb a more specific entity. The system must use a null value for attribute values of the specific entity that were not in the original generalised entity.

**Rename:** renames an entity.

**Combine:** joins two entities which have identical types, and creates a new entity with a new entity name.

**Connect:** joins two entities which have identical types, and the new entity has the name of one of the original entities.

**Aggregate:** creates an intermediate entity between an existing entity and a subset of its attributes. *Aggregate* can be used to normalise the schema to a form where non-key attributes of each entity are fully dependant on the key. *Aggregate* allows the relocation of an attribute on the schema.

**Telescope:** removes an entity by assigning its attributes directly to its ancestor entity. *Telescope* allows the relocation of an attribute on the schema.

**Add:** allows the addition of implicit attributes to an entity, along with a constant function to assert the value.

**Delete:** removes a portion of the database not relevant to the application.

Superviews appears well suited to a specific class of problems, those that can be described with these ten operators. For this class of mapping, data can be moved between any schema and the superview with guaranteed consistency. For any schemas which fall outside that which can be described with these operators it provides no help. Such schemas include those which do not have simple one-to-one correspondences between attributes in a schema and those in the superview. Therefore, if attributes are derived from an equation, or aggregated values, Superviews can not be used. This limits the utility of Superviews in large model mapping systems, as most schemas in this domain require sophisticated manipulation of attributes as well as manipulation of object references, which are also not handled in this RDBMS-based integration scheme.

#### **4.2.8 RDBMS views**

In relational database systems, views of the conceptual database are a well established concept. A view provides an abstract schema of a portion of the conceptual database. This notion differs from the notion of mappings between schemas, in that the mapping may not be to a schema which is a portion of a conceptual database, the mapping may be to a schema which is a superset of the conceptual schema. RDBMS views may, under certain conditions, be updateable, thus providing the effect of a bidirectional mapping as discussed in this thesis. However, views that are

updateable have severe restrictions on their definition, which limit them to what amounts to a one-to-one mapping between tables with direct equivalences between attributes (i.e., no equations or functions in the view definition).

<b>Requirement</b>	<b>E X P R E S S - M</b>	<b>E X P R E S S - V</b>	<b>E X P R E S S - C</b>	<b>T r a n s f o r m</b>	<b>E D M - 2</b>	<b>K I F</b>	<b>S u p e r v i e w s</b>	<b>R D B M S</b>
Language level	M	M	M	M	M	M	L	L
Declarative / Procedural specification	P	P	P	P	D	D	P	P
Object-oriented language support	L	L	L	-	M	M	-	-
Object-oriented method support	-	-	-	-	-	L	-	-
Bidirectional mapping support	-	L	-	-	L	L	-	-
Conditional mapping invocation	M	M	M	L	H	H	L	M
Initial values for attributes and object creation	M	M	M	L	M	L	L	L
Aggregate detail over objects and attributes	M	M	M	L	H	M	M	H
Relationship handling (expanding and truncating pointer chains)	M	M	M	M	M	M	M	M
Class graph-based model (Yes / No)	Y	Y	Y	Y	Y	Y	Y	Y
Class and attribute graph-based model (Yes / No)	N	N	N	N	Y	Y	Y	Y
Unit handling is Implicit / Explicit	E	E	E	E	E	I	E	E
Type handling is Implicit / Explicit	E	E	E	E	E	I	E	E
Temporary structures (objects and attributes) available (Yes / No)	N	N	N	N	Y	Y	Y	Y
Graphical notation available (Yes / No)	N	N	N	N	Y	N	N	N

**Table 4.12** Comparison of mapping languages (H=High, M=Medium, L=Low, -=None)

### 4.3 Summary of Inter-Schema Relationship Modelling

Table 4.12 provides a summary relating the surveyed mapping languages to the requirements detailed in Section 4.1.3. As shown in the table, none of the surveyed languages provides for the full range of requirements necessary. Of particular concern is the low level of support for bidirectional mapping, given that this type of mapping is the norm in the domains of these languages. There is also a need for object-oriented support, especially as the modelling notations that many of the mapping languages support are themselves object-oriented. Missing from many of the EXPRESS-based languages (which are promoted for this domain) are notions of classes and attributes forming the graph representing a mapping, as well as the ability to define temporary structures for partial mappings or reusable states. Almost all of the languages have no graphical

notation, and even EDM-2's notation does not provide a good overview of what takes place in an individual mapping.

To address the deficiencies in the existing languages, a new view mapping language (VML) is proposed in Chapter 5. It has both a textual and graphical notation, and a modelling environment for VML is described in Chapter 6.