Structural Information Mapping with Express-X

Peter O. Denno
Manufacturing Engineering Laboratory
National Institute of Standards and Technology

Donald B. Sanderson
National Institute of Standards and Technology
East Tennessee State University

ABSTRACT

This paper provides an overview of the Express-X language, a language that provides structural information mapping of information modeled in EXPRESS schema. The paper presents the design rationale for Express-X. The paper also serves as a tutorial in information mapping of EXPRESS-based information models and its relation to SQL mapping capabilities.

1.0 Introduction

An information model is an abstraction of a perceived reality. Not surprisingly, any two efforts to codify this perceived reality will differ, depending on the context in which the information is perceived. In terms of database schema, these differences manifest themselves as differences in scope of application, selection of concepts, meaning of values, and domain of values [Wiederhold]. Some of these differences are merely structural, others are semantic.

Express-X [Express-X] is a structural information mapping language. Structural difference in information can be distinguished from semantic differences: two assertions differ in semantics if acting on the information may result in two different behaviors. If the information only differs with respect to structure (encoding) then the two structures must produce identical behavior. The principle purpose of a structural mapping engine is to reconcile the structural differences between information in different schemas. Typically this reconciliation takes the form of mapping (re-writing) structures from a :source" form into a "target" form.

Express-X provides two basic forms of mapping: mapping information between structures in two Express schema, and viewing information. An Express-X mapping engine is a program implementing the Express-X language. An Express-X engine may take many forms. For example, a *push* implementation maps all of the information from a source data set to a target data set. Another implementation may allow selective *pull* of information, that is,

one selects the target entity to be created and the engine identifies the source data instantiating that target entity or entity type extent.

In Express-X, viewing differs from mapping in that viewing does not involve a target schema nor target data set. In viewing, the view declaration itself defines the structure of the information that is the ‘target’ of the computation.

Viewing is ephemeral; once the source information underlying the view is changed, the view is potentially invalid and should be recomputed. Mapping may be ephemeral or permanent, depending on the implementation of the mapping engine. Mapping is often used to translate data from one structural form to another; once such mapping is performed, whatever relationships existed between the source and target data sets are lost and the two data sets then evolve independently.

Express-X accommodates a form of persistent relationship between source data and views or target data. An Express-X engine may implement views or target entities that are ‘updatable,’ that is, view instances or target entities which, when modified propagate the modification of values back to the source data from which they were computed. Typically updatable views are implemented as accessors on source data; the view itself is not a copy of source data values but a readable / writable perspective on the source values themselves.

2.0 Why Express-X ?

Express-X is based on EXPRESS (ISO 10303-11) [Express]. EXPRESS allows one to specify a domain of data in the form of constraints (membership predicates). Here “domain of data” refers to EXPRESS’s ability to specify whether a data value, a named entity type instance (similar to a C language struct) or an entire data set satisfies criteria that correspond to the semantic intent of the schema. These criteria are constraints written as value range limits, enumerations of possible values, constraints on permissible combinations of named entity types, and existence constraints (e.g., “All ‘person_in_department’ entities must refer to a ‘person’ entity through a given path of relationship). EXPRESS therefore is an abstract data type language for communicating complex constraint and relationship information on data. Such capability is necessary to encode the complex relationships and constraints that are typically found in product data. (EXPRESS is used in the STEP [STEP] standards for product data of ISO TC184/SC4). One answer to the question “Why Express-X?” then is that the mapping language must be powerful enough to cope with EXPRESS-based data. Express-X, being based on EXPRESS, is such a language.

Express-X is as powerful as EXPRESS but that fact does not in itself justify the invention of a new language; general programming languages might well provide the same capability. Why then not use a programming language rather than a new mapping language? There are a few reasons. First, Express-X is, as much as possible, declarative. Rather than reading as procedural code, Express-X code looks like an assertion of the relationship between information elements. Express-X is generally more readable than procedural

code and far less apt to be influenced by one's idiosyncratic software design choices. Secondly Express-X has a published execution model [Express-X]. The execution model ensures that the mapping behavior can be well understood. Further, the execution model is a relatively efficient one and affords opportunities for optimization. The means by which mapped information is identified, for example, encourages implementations which reuse computational results rather than re-compute. That is, a mapping declaration may call other mapping declarations for results computed previously. Finally, Express-X provides a language for documenting the relationships between standardized information models. Examples of this are: (1) documenting the relationship between a standardized Express information model and its revisions; and, (2) documenting the relationship between Application Reference Models (ARMs) and Application Interpreted Models (AIMs) in STEP interpreted data. [Valois]

3.0 Fundamental Concepts

This section introduces the basic concepts of Express-X, the execution model and the identification of mapped structures. The VIEW construct is discussed first. Much of what is possible in VIEWs is also possible in MAPs, as will be discussed later.

An example view declaration:

```
VIEW_SCHEMA example;
REFERENCE FROM source_schema;

VIEW person_in_organization;
FROM (p : source_schema.person;
    r : source_schema.person_in_department);
WHERE (p.name = r.person);
SELECT
  name : STRING := p.name;
  employee_number : STRING := p.employee_number;
  department : STRING := r.department_name;
END_VIEW;
END_VIEW_SCHEMA;
```

The view declaration above describes a mapping of information from data in a source schema (called source_schema) to the view structure. The source schema, an EXPRESS schema, might look like the following:

```
SCHEMA source_schema;
ENTITY person;
  name : STRING;
  employee_number : STRING;
END_ENTITY;

ENTITY person_in_department;
  department_name : STRING;
  department_number : INTEGER;
  person : STRING;
END_ENTITY;
END_SCHEMA;
```

The purpose of the view is to associate with each person the department in which he or she works. An SQL [SQL] view performing a similar task is provided below. The definition of the relational tables used in the SQL example below and the instance data of the EXPRESS example are provided in Appendix A.

```
CREATE VIEW source_schema.person_in_organization AS
    SELECT ( p.name AS name,
             p.employee_number AS employee_number,
             r.department_name AS department
    FROM source_schema.person AS p,
        source_schema.person_in_department AS r,
     WHERE p.name = r.person);
```

3.1 Identification and Enumeration of Instances

Most essential to understanding the view (or map) declaration is understanding how it identifies and enumerates instances of its type when the view is applied to a source data set. The view provides this information in its FROM clause (and IDENTIFIED_BY clause if one is provided). TheExpress-X FROM clause above is:

```
FROM (p : source_schema.person;
      r : source_schema.person_in_department);
```

The FROM clause defines a *binding type* and, when applied to data set, a *binding extent*. The binding type is notional; it cannot be directly accessed through the language. It can be thought of as the type of tuples of the cartesian product of the sets (extents or instances of types) listed in the FROM clause. In this example, the tuples are *<(instance of person), (instance of person_in_department)>*. The binding extent is the collection of such tuples in the cartesian product of the entity extents (sets) of types person and person_in_department of some data set. Instances in the binding extent are called *binding instances*. An example entity population and binding extent for this example can be found in Appendix A.

The FROM clause defines the set of binding instances over which the body of the view or map is iterated. Because the binding extent is built as a cartesian product, for practical reasons, it is best to keep the order of the tuples (the number of types in the FROM clause) to a minimum (ideally just one or two).

In a view, a binding instance may identify an instance of the view. In a map it may identify a collection of target entities. The body of the view is evaluated in the context of the binding instance, that is, the variables in the FROM clause (‘p’ and ‘r’ above) are bound to the corresponding elements in the binding instance tuple. (e.g., ‘p’ is bound to a person and ‘r’ to a person_in_department). The instance of person and instance of person_in_department (their object IDs) are therefore a key (concatenated key in this case) to the view instance the view creates. This key can be used elsewhere in the Express-X schema to access the view or map target entities. For example, the person_in_organization view above could be called as though it were a function with two arguments, a particular person and particular

person_in_department. This function would return the corresponding person_in_organization view instance.

The FROM clause, therefore, defines an identification scheme for instances of the view or collections of target entities in a mapping. View instances can be accessed by calling the view (as a function) with values of the types defined in the FROM clause. This use of the view as a function is called an *explicit binding.*

There is an alternative identification schema to using values of the types defined in the FROM clause. This alternative is provided by the IDENTIFIED_BY clause and is mutually exclusive with the FROM clause as a means of identification. The IDENTIFIED_BY clause allows one to define a key composed of arbitrary expressions based on instances of the types defined in the FROM clause. For example, one might add the clause IDENTIFIED_BY (p.employee_number) to the view above. Instances of person_in_organization would then be identified and enumerated by the employee_number attribute of a person entity, rather than the concatenation of person and person_in_department object IDs. The explicit binding function would likewise take just one argument, an employee_number.

The IDENTIFIED_BY construct raises an issue as to what exactly is being identified. In the case that no IDENTIFIED_BY clause is stated, there is a 1-1 correspondence between binding instances in the qualified binding extent and view instances (the sense of ‘qualified’ will be explained later). Were the view above to define IDENTIFIED_BY (p.name) that is, identify the view instances by a person’s name attribute, and were there to be multiple persons with name “Smith” the correspondence would not be 1-1 but rather many binding instances to one view instance. The question then is, what attribute values would this one view instance take on, given that it must represent a number of binding instances. For example, given that there might be two persons “Smith” what is the view instances employee_number attribute? The (pragmatic) rule adopted by the designers of Express-X is that if all binding instances mapping to a view instance evaluate to the same value for a given attribute, then that value is given to the view instance. However, if two or more binding instances have different values for the attribute, the attribute’s value is set to the indeterminate value defined by EXPRESS.

The discussion so far has not touched upon a key notion of the Express-X execution model. The set of bindings on which the view or map is evaluated is not directly that set constructed from the FROM clause (*i.e.* the binding extent) it is rather a subset of the binding extent called the *qualified binding extent.* The qualified binding extent is the subset of binding instances that satisfy a selection criteria defined by WHERE clauses (there may be more than one WHERE clause in a declaration). In the example above the WHERE clause is: WHERE (p.name = r.person); the qualified binding extent is that subset of binding instances whose person instance have a name attribute equal to the binding instance person_in_department person attribute. (In this example, the WHERE clause functions as a relational join).

3.2 Map

Everything said above regarding binding types, binding instances, FROM, WHERE and IDENTIFIED_BY clauses applies equally to view and map declarations. The principle difference between view and map declarations is that the view defines a target structure and the map references target structures as entity type definitions in a target schema.

An Express-X map declaration similar to the view declaration above is:

```
SCHEMA_MAP example_map;
 TARGET target_schema;
 SOURCE source_schema;
MAP person_in_organization as
      t:target_schema.target_person_in_department
  FROM (p : source_schema.person;
       r : source_schema.person_in_department);
 WHERE (p.name = r.person);
 SELECT
   t.name := p.name
   t.employee_number := p.employee_number
   t.department := r.department_name;
END_MAP;
END_SCHEMA_MAP;
```

The target schema might be as below.

```
SCHEMA target_schema;
ENTITY target_person_in_department;
  name : STRING;
  employee_number : STRING;
  department : STRING;
END_ENTITY;
END_SCHEMA;
```

The key difference in the map is that it declares the entity types in the target schema to be created:

```
MAP person_in_organization as
      t:target_schema.target_person_in_department;
```

The example creates only one entity (of type target_person_in_department) however a map declaration may create multiple entities of various types in multiple schema. Because the map may create multiple target entities, it raises the issue of how the individual instances being created can be identified. In the example above, one may use an explicit binding to obtain an instance of the target entity type target_person_in_department using the expression below:

```
t@person_in_organization(a_person, a_person_in_department);
```

That is, the explicit binding above is provided a person (signified by the a_person variable, and a person_in_department signified by the a_person_in_department variable). Conceptually the explicit binding generates a collection of target entities, and the ‘t@’

notation identifies that the expression should return the entity target_person_in_department of that collection that was bound to ‘t’.

SQL similar in intent is provided below. To create the table and populate it so that it is independent from updates in the data source, an SQL-based system first creates the table and then populates it as shown below. Note: assume that target_schema contains a table called target_person_in_department whose column definitions correspond to the view person_in_organization in the prior example. The following SQL would populate the table with information from the person and person_in_department tables in the source schema:

```
INSERT INTO target_schema.target_person_in_department
  (name,employee_number,department)
  SELECT (p.name, p.employee_number, r.department_name
  FROM    source_schema.person AS p,
        source_schema.department AS r
   WHERE p.name = r.person);
```

3.3 View Extents

The collection of view instances corresponding to the qualified binding extent is called a *view extent*. The FROM clause of a view or map declaration may reference a view extent as it would an entity extent from a source data set. When the FROM clause references a view extent, the mapping engine must ensure that the referenced view extent is available (by evaluating the view if necessary) before evaluating the referencing view. The person_in_organization view is referenced in the FROM clause of the Express-X example view below:

```
VIEW approval_department
FROM (po : person_in_organization;  -- a view
      a  : source_schema.approval); -- an entity
WHERE (a.person = po.name)
SELECT
  department : STRING := po.department;
  action : STRING := a.item;
END_VIEW;
```

Because view extents are accessible within the language, it is possible to use them to record intermediate results or commonly accessed information. Accessible view extents used in these ways help to manage the complexity of mapping.

3.4 Partitions

The view as described above is a mapping (“mapping” in the mathematical sense) from a binding extent to a view extent. The notion of view partitions allows one to map several different binding extents into a single view extent. Said another way, a view extent is the union of the mappings of the partition binding extents into the view extent. In a partitioned view (see example below) each partition defines its own FROM clause and thus defines its

own binding extent. The body of each partition must contain identical attributes, because, of course, the instances in the view extent must all be of one type.

The example below collects source ‘male’ and ‘female’ entity instances into a view extent called ‘person’.

```
VIEW person
  PARTITION male :
  FROM (m : source_schema.male);
    name : STRING := m.name;
    employee_number :STRING := m.number;
  PARTITION female:
  FROM (f : source_schema.female);
    name : STRING := f.name;
    employee_number :STRING := f.number;
END_VIEW;
```

The value of view partitioning is that it allows one to define a view extent as a union of ‘sub extents’. The concatenated extent may be used (for example, in the binding type of another view declaration) without regard to it being the union of anything. An explicit binding function is defined for each partition, thus in the example above one may call:

```
person.male(a_male);
person.female(a_female);
```

Partitions in maps are similar to partitions in views, they allow one to specify various binding extents, each mapping (in the mathematical sense) to collections of target entity instances. Each partition must create identical types of instances in the same numbers. A map that ranges over source_schema ‘male’ and ‘female’ entity types is shown below:

```
MAP person_map as p.target_schema.person
  PARTITION male :
  FROM (m : source_schema.male);
    p.name := m.name;
    p.employee_number := m.number;
  PARTITION female:
  FROM (f : source_schema.female);
    p.name := f.name;
    p.employee_number := m.number;
END_VIEW;
```

3.5 Subtyping

Subtyping, particularly in maps, provides a sophisticated approach to code reuse and the specification of the created network of target types. Subtyping allows one to distribute the specification of created types and selection criteria throughout a hierarchy of map declarations. Subtype declarations may extend the collection of instances created by their supertypes, subtype those instances created and require additional selection criteria beyond those specified in their supertypes. In the example below, a binding instance is mapped to a target ’t_project’ entity which may additionally be one of its sub-

types,’t_in_house_project’ or ’t_external_project’ depending on the selection criteria (whether the source instance project type is ’in house’ or ’external’).

```
SCHEMA source_schema;
ENTITY s_project;
  name : STRING;
  cost : INTEGER;
  project_type : STRING;
  vendor : OPTIONAL STRING;
END_ENTITY;
END_SCHEMA;

SCHEMA target_schema;
ENTITY t_project;
  name : STRING;
  cost : INTEGER;
  management : inside_or_out;
END_ENTITY;

ENTITY in_house_project;
 SUBTYPE OF (t_project);
END_ENTITY;
ENTITY external_project;
 SUBTYPE OF (t_project);
END_ENTITY;
END_SCHEMA;


MAP project_map AS tp : target_schema.t_project;
FROM p : source_schema.s_project;
SELECT
    tp.name := p.name;
    tp.cost := p.cost;
END_MAP;

MAP in_house_map AS tp : target_schema.in_house_project;
SUBTYPE OF project_map;
WHERE (p.project_type = ’in house’);
SELECT
  tp.management := CHOICE (cost < 50000) THEN ’small accts’
                ELSE ’large accts’ ENDIF;
END_MAP;

MAP ext_map AS tp : target_schema.external_project;
SUBTYPE OF project_map;
WHERE (p.project_type = ’external’);
SELECT
   tp.management = p.vendor;
END_MAP;
```

In the example above, the supertype specifies a FROM clause identifying the binding extent (when applied to data) on which the bodies of person_map and its subtypes are executed. If a supertype map defines a FROM clause, none of its subtypes may do so. The execution model, identifying what target instances are created is as follows:

- If a map’s selection criteria and that of all its supertype maps is satisfied, the map may execute.
- A subtype map may declare a target parameter id that is also declared in any of its supertype maps (same identifier). The type created is the composition of types identified by the subtype map target parameter and all supertype maps declaring a target parameter with this target parameter id.
- A subtype map may introduce a target_parameter_id that is not defined in any of the supertype maps. In this case a new target entity of the type defined by the target parameter is created.

In the example above "tp" is a target parameter id. Because this parameter is defined in "project_map" and "in_house_map" an entity that is both type "project" and "in_house_project" may be created. Because this parameter is defined in "project_map" and "external_project_map" an entity that is both type "project" and "external_project" may be created. Whether the created instance includes type "in_house_project" or "external_project" or is just "project" depends on the satisfaction of the corresponding selection criteria.

3.6 Distribution of Target Entities

Sometimes what is desired is to create several target entities from a single source entity. This can be performed by a FOR loop wrapping the select clause, as the following example illustrates:

```
SCHEMA source;
  ENTITY parent;
    children : INTEGER;
  END_ENTITY;
END_SCHEMA;

SCHEMA target;
  ENTITY parent;
  END_ENTITY;
  ENTITY child;
     parent : parent;
  END_ENTITY;
END_SCHEMA;

MAP parent_and_child AS tp : target.parent, c: LIST [0:?] OF target.child;
FROM sp: source.parent;
FOR i:= 1 to p.children
SELECT
  parent := sp@parent_and_child(source.parent);
END_ENTITY;
```

In this example, execution of the map declaration for each parent source entity instance creates a target parent entity instance and zero or more target child entity instances. The number of child entity instances created for each parent is equal to the value of the source attribute, "children", in the source parent instance. The mapping uses an explicit binding to set the child entity's "parent" attribute.

4.0 Conclusion

Express-X, as an extension to the EXPRESS language, enables the re-use of existing concepts and terms, thus providing EXPRESS-like capabilities in a minimal extension to EXPRESS. Its declarative style furthers this goal by allowing mappings to read like constraints and function calls.

Express-X supports the definition of dynamic views of existing data, as well as the mapping of data between defined formats. It has simple mechanisms for materializing relationships between entities, and for dealing with normalizing and de-normalizing structures in mappings.

The language is currently under development as a New Work Item under ISO TC184/SC4. Tools that implement draft versions of the language are now available, including some in the public domain [Denno].

Appendix A

This appendix provides SQL Tables and ISO 10303-21 instance data [Part 21] used in the examples.

TABLE 1. - PERSON

name	employee_number
Smith	601
Jones	203
Miller	604

TABLE 2. - person_in_department

department_name	department_number	person
Engineering	6	Smith
Warehouse	2	Jones
Engineering	6	Miller

Similar Express-oriented data (in ISO10303-21 form).

#1 = person(‘Smith’,’601);
#2 = person(‘Jones’,’203);
#3 = person(‘Miller’,’604’);

#11 = person_in_department(‘Engineering’,6,’Smith’);
#12 = person_in_department(‘Warehouse’,3,’Jones’);
#13 = person_in_department(‘Engineering’,6,’Miller’);

The binding extent corresponding to FROM (p : person; r: person_in_department) for this data:
<#1=person(‘Smith’,’601’),#11=person_in_department(‘Engineering’,6,’Smith’)>
<#1=person(‘Smith’,’601’),#12=person_in_department(‘Warehouse’,6,’Jones’)>
<#1=person(‘Smith’,’601’),#13=person_in_department(‘Engineering’,6,’Miller’)>
<#2=person(‘Jones’,’601’),#11=person_in_department(‘Engineering’,6,’Smith’)>
<#2=person(‘Jones’,’601’),#12=person_in_department(‘Warehouse’,6,’Jones’)>
<#2=person(‘Jones’,’601’),#13=person_in_department(‘Engineering’,6,’Miller’)>
<#3=person(‘Smith’,’601’),#11=person_in_department(‘Engineering’,6,’Smith’)>
<#3=person(‘Miller’,’601’),#12=person_in_department(‘Warehouse’,6,’Jones’)>
<#3=person(‘Miller’,’601’),#13=person_in_department(‘Engineering’,6,’Miller’)>

The qualified binding extent corresponding to the above binding extent and the selection criteria WHERE (p.name = r.department_name) is:

<#1=person(‘Smith’,’601’),#11=person_in_department(‘Engineering’,6,’Smith’)>
<#2=person(‘Jones’,’601’),#12=person_in_department(‘Warehouse’,6,’Jones’)>
<#3=person(‘Miller’,’601’),#13=person_in_department(‘Engineering’,6,’Miller’)>

References

Date, C.J and Darwen, H., *A Guide to the SQL Standard, Third Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

Denno, P.O., *NIST Expresso Home Page*, http://www.nist.gov/expresso, July, 1999.

Express, *ISO 10303-11:1994 Industrial Automation Systems and Integration --- Product Data Representation and Exchange --- Description Methods: The EXPRESS Language Reference Manual*.

Express-X, *The Express-X Language Reference Manual*, http://www.nist.gov/sc4/wg_qc/wg11/n078/, March 18, 1999.

Part-21, *ISO 10303-21:1994 Industrial Automation Systems and Integration --- Product Data Representation and Exchange --- Description Methods: Clear text encoding of the exchange structure*.

Valois, J., *Capturing the Information in AP Mapping Tables*, http://www.nist.gov/sc4/wg_qc/wg11/n054, 1999.

Wiederhold, G., *Mediators in the Architecture of Future Information Systems*, IEEE Computer, pp 38-49, March, 1992.