INTRODUCTION TO EXPRESS

By Chuck Eastman

Express is the most current language or meta-representation used in the development of STEP. It has been under development since about 1990. It is general and quite like an object-oriented programming language. Here, it is presented from the bottom up, showing its elements and how they can be put together.

1. SCHEMAS

The unit of definition is called a SCHEMA. Unless noted otherwise, all names are global to a SCHEMA. There are special mechanisms to cross references across SCHEMAS, but this requires a special syntax in relation to references within the same SCHEMA. SCHEMAS will be defined in detail later.

2. NAMING

In general, the convention is followed that all user defined names are lower case, while all reserved words are capitalized. Names may have embedded underbars(_) but no other special characters.

3. BASIC TYPES:

EXPRESS provides a set of basic types, that are predefined and available to use in the definition of higher level types.

The Basic Types are:

```
NUMBER, REAL, INTEGER, STRING, LOGICAL, BOOLEAN and BINARY.
```

`NUMBER` is a generalization of `INTEGER` and `REAL.` `STRING` is a list of characters. `LOGICAL` can have the values (`TRUE, FALSE, UNKNOWN`) while `BOOLEAN` only has (`TRUE,FALSE`). `BINARY` is a vector on bit values.

Basic types can be used to define higher level types, eg.

```
TYPE area : REAL;
TYPE name : STRING;
```

Types also may be used to define attributes within higher level entities, eg.

```
ENTITY part;
      -
      -
      weight : REAL;
      -
END_ENTITY;
```

Both STRING and BINARY are variable length vectors. They may have a length that is specified or not. When not specified, the length is variable. When a length is defined, up to that many may be assigned. When the length is specified and appended by FIXED, then all assignments must be of exactly this length. For example:

```
     s1 : STRING;          (* variable length string *)
     s2 : STRING(10);      (* variable number up to 10
characters*)
     s3 : STRING(10) FIXED;     (* exactly 10 characters *)
```

EXPRESS also has an enumeration type, where the possible values are explicitly defined. For example:

```
     TYPE compass_direction = ENUMERATION OF
       (south, north, east, west );
     END_TYPE;
```

4. CONSTRUCTORS

In addition to single types, variables and higher level structures can be aggregated into larger groupings. This includes arrays, bags, lists and sets. They are presented in order.

ARRAY is used to define a vector of elements of fixed size and these may be concatenated, eg.:

```
     matrix : ARRAY[1:4] OF ARRAY[1:4] OF REAL;
```

Both the lower and upper bound must be defined in arrays. It must be that lower_bound. < upper_bound.

BAG is an unordered collection of. like elements. Its lower bound and upper bound may or may not be specified. If the lower bound is specified, there must be at least the lower bound n assigned. If the upper bound is assigned, this a the maximum number of elements. If the bounds are not defined, it is assumed that the bounds are [0:?]. As an example,

```
     bag_of_points : BAG[1:?] OF point;
```

means that there must be at least one point in `bag_of_points`.

LIST is an ordered collection of like elements, similar to the ARRAY, but LIST may be variable length. Thus:

```
     list_of_points : LIST[0:?] OF points;
```

means that there may any number of points in the list, which remain ordered.

SET is an unordered collection of like elements. Duplicates are not allowed. SETs may be of fixed size or not. An example might be:

```
set_of_names : SET OF [1:100] OF name;
```

`set_of_names` must have a membership of at least 1 and not more than 100.

There is one other type definition, to define a type that may be selected from among a set of types, similar to a UNION type in C. EXPRESS calls this a `SELECT TYPE`. A `SELECT` type is also a generalization of other types. Some examples follow:

```
TYPE NUMBER = SELECT(REAL,INTEGER);
END_TYPE;

TYPE connection = SELECT(nail,screw,bolt);
END_TYPE;
```

In all cases, the elements of the set selected from must be types and already defined.

Types may be used to define higher level types, so that a large number of types may be defined in terms of simpler ones. Notice that the limitation is that all higher level uses of a type have the same domain as the lower level ones, unless rules are used to constrain them (see Section 5.2.)

5. ENTITIES

The general object type, that allows definition of unlike elements, is called the ENTITY. It supports a wide set of complex elements to be defined within it, which we will consider incrementally.

The basic `ENTITY` definition has the format:

```
ENTITY  point;
    x,y,z : REAL;
    ref_coord: cartesian_coordinate_system;
END_ENTITY;
```

The point is composed of three attributes of type REAL and a relation with `cartesian_coordinate_system`.

ENTITYs can be inherited or specialized into other ENTITYs. For example:

```
ENTITY homogeneous_point
    SUBTYPE OF (point;)
    SUPERTYPE OF (colored_point);
    w : REAL;
END_ENTITY;

ENTITY colored_point;
       color : enumeration of (red, yellow, blue, green);
END_ENTITY;
```

An ENTITY may define both SUBTYPE and SUPERTYPE relations with other types. The implications are that all the elements of the SUPERTYPE are copied into the SUBTYPE.

Access to supertype elements is by way of pathnames. For example the variables available in homogeneous_point are:

```
SELF\point.x;
SELF\point.y;
SELF\point.z;
w;
```

In the above pathnames, SELF refers to the current ENTITY and '\' is the supertype relation. These are the pathnames that would be used to access these attributes from within homogeneous_point. (w is not accessible from point.)

Either the SUPERTYPE or the SUBTYPE may define a relation between the two ENTITYs. However, all SUPERTYPE and SUBTYPE declarations, taken together, must be consistent with a directed acyclic graph. That is, there must be no cases where an ENTITY is both a SUBTYPE and a SUPERTYPE of the same ENTITY. There can be no cycles in the graph.

5.1 Derived Attributes

In addition to explicit attributes, that are assigned values, EXPRESS also supports derived attributes. These are not explicitly loaded as data, but are computed at assignment time from other values carried within an ENTITY. A fairly complex example (not of my style but illustrative) follows. It includes constructs that we will get to eventually. Derived attributes are identified by DERIVE:

```
ENTITY circle;
    center ; point;
    radius : REAL;
    axis : vector:
DERIVE
    area : REAL := pi * radius ** 2;
END_ENTITY;
```

Thus the attribute area can be accessed just like other attributes, but is computed when it is accessed. The scope of attributes accessed within a DERIVE expression is the ENTITY in which the DERIVE expression is located.

5.2 Domain Rules

Restrictions on the possible values allowed for different attributes can be defined, using EXPRESS domain rules. These are a clause within the ENTITY specification, initiated by WHERE. An example WHERE might be:

```
ENTITY vector;
   a, b :  REAL;
   c :     OPTIONAL REAL;
WHERE
   length1 : a**2 + b**2 + c**2 = 1.0;
END_ENTITY;
```

All domain rules, such as `length1,` are integrity constraints which carry a value of type LOGICAL. When accessed, it supposedly evaluates the expression and returns one of the values TRUE, FALSE or UNKNOWN. UNKNOWN is used when some attributes are missing. The OPTIONAL modifier on the `c` allows this value to optionally exist or not exist.

EXPRESS supports the definition of subroutines, called FUNCTIONS, that can be used to defined complex WHERE clause or DERIVE rules.

6. RELATIONS

Object models are basically defined as a collection of objects (In EXPRESS, called Entities) and relations. While there has been great attention in object oriented systems in the defining of entities, there has been less effort to define relations well.

At their most basic, relations may be one-to-one, one-to-many, or many-to-many. One-to-one relations are the easiest to define, where one entity has an attribute that refers to an instance of another entity. In the example of circle above, the entity circle refers to two other entities, point and vector these are both one-to-one relations. One-to-many relations are easily defined by making the referencing attribute a constructor.

6.1 Defining Many-to-Many Relations

Most languages allow a user to define a relation in one direction, but not both. This makes the definition of many-to-many relations difficult. That is, in the definition of a polygon made up of a list of edges, it is straightforward to define relations from polygon to edges, for example. It is less common to have ready ways to define relations in the reverse direction. Such relations are required when an edge may be part of multiple polygons and a polygon is composed of multiple edges.

EXPRESS solves this problem by providing the INVERSE clause. We use the polygon example:

```
ENTITY line;
   pt1 : point;
   pt2 : point;
WHERE
   not_concident : (SELF\pt1.x<>SELF\pts.x) OR
         (SELF\pt1.x<>SELF\pts.x) OR
         (SELF\pt1.x<>SELF\pts.x);
INVERSE
   loops : SET [0:2] OF polygon FOR edges;
END_ENTITY;


ENTITY polygon;
   edges : LIST [2 : ?] OF lines;
WHERE
   2_connected : (* check that all of the line's endpoints
                    are all two-connected *)
END_ENTITY;
```

The polygon references a set of lines that comprise it. The INVERSE relation in the lines identifies a SET of polygon references, thus allowing one to interrogate how many and which polygons a line a part of. The polygon ENTITY specifies that it must have at least two lines within it, but a liner may be part of zero to two polygons An INVERSE relation references the corresponding relation it mirrors in the opposite direction. A constraint is implied that for every relation in one direction, there is a corresponding relation in the other direction..

7. SUPERTYPE CONSTRAINTS

In the general case, when SUBTYPEs are defined of a SUPERTYPE, they may be responding to a variety of conditions. A person, for example may be specialized into male and female. In this case, it is not usually possible for an instance to be both of these types. On the other hand, a room may have SUBTYPEs of west_direct, east_directed, south_directed and north_directed. A particular room instance may be both east_directed and south_directed.

To make these different cases explicit, EXPRESS provides constraints for the SUBTYPE clause. They are:

ONEOF -- defines that the following set of SUBTYPEs are mutually exclusive. An instance may be of only one SUBTYPE.

AND -- defines that the following set of SUBTYPEs are always included in all instances of the SUPERTYPE.

ANDOR -- defines that there is no rule and that the SUBTYPEs may be defined into instances or not.

An example making use of these constraints follows:

```
ENTITY mechanical_part
SUPERTYPE OF (AND (power_part,handling_part));
...
END_ENTITY;

ENTITY power_part
SUPERTYPE                                                    OF
(ONEOF(fluid_powered,electrical_powered,powerless));
...
END_ENTITY;

ENTITY handling_part;
SUPERTYPE                                                    OF
(ONEOF(air_handling,liquid_handling,communication));
...
END_ENTITY;
```

In the above example, mechanical_part is defined as always being an instance composed of two subtypes, a power_part and a handling_part. There are several subtypes of power_part and also several subtypes of handling_part. Some example possible mechanical_part instances would be

powerless+communication, fluid_powered+air_handling, liquid_powered+liquid_handling. In all, instances can be defined of any member of the cartesian product of power_part and handling_part.

8. INSTANCES

EXPRESS assumes that all ENTITYs may have instances. That is, there may be many copies of each ENTITY. In some cases, however, one wishes to define an ENTITY for the purposes of defining other ENTITYs, but for which there will never be any instances. In this case, the entity is defined ABSTRACT ENTITY.

9. EXAMPLE

```
SCHEMA example;

TYPE date = ARRAY [1:3] OF INTEGER;
END_TYPE;

FUNCTION years(d : date) : INTEGER;
(* computes an age to the current date from d *)
END_FUNCTION;

TYPE hair_type = ENUMERATION OF
  (brown,
  black,
  blonde,
  redhead,
  gray,
  white,
  bald);
END_TYPE;

ENTITY  person;
   SUPERTYPE OF  (ONEOF(male, female));
   first_name     : STRING;
   last_name : STRING;
   nickname  : OPTIONAL STRING;
   birth_date     : date;
   children  : SET [0 : ?] OF person;
DERIVE
   age  :  INTEGER  := years(birth_date);
INVERSE
   parents : SET [0 : 2] OF person FOR children;
END_ENTITY;

ENTITY female;
   SUBTYPE OF (person);
   husband   : OPTIONAL male;
  maiden_name     : OPTIONAL STRING;
WHERE
  WI : (exists(maiden_name) AND EXISTS(husband)) OR
       NOT EXISTS(maiden_name);
```

```
END_ENTITY;

ENTITY male;
   SUBTYPE OF (person);
   wife           : OPTIONAL female;
END_ENTITY;


END_SCHEMA;
```

REFERENCES:

Danner, W.F., D.T. Sanford and Y. Yang [1991], "STEP (Standard for the Exchange of Product Model Data) Resource Integration: Semantic and Syntactic Rules", Building and Fire Research Laboratory, Report NISTIR 4528, March, Gaithersburg, Md.

Eastman, C.M. [1999] *Building Product Models*, CRC Press, Boca Raton FL.

Haas, Wolfgang, draft [1991] STEP Part 201 *Explicit Draughting*, TC 184/SC4 N104, N.I.S.T. 11 October.

Kramer, T.R., M. Palmer and A.B. Finney, [1992], "Issues and Recommendations for a STEP protocol Framework, *NISTIR 4755*, National Institute of Standards and Technology, January 17.

Schenk, Doug, [1991], EXPRESS Language Reference Manual: External Representation of Product Definition Data, ISO TC184/SC4/WG5, Document N14 (29 April, 1991)

Schenk, D.A. and P.R, Wilson [1994], Information Modeling the EXPRESS Way, Oxford U. Press, N.Y.