EXIST: A Language to Support Data Modelling and Data Model Integration

Matthew West & Chris Angus – Shell Services International, UK

1. Introduction

EXIST (**EX**pression of **I**nformation based on logic and **S**et **T**heory) is a new language that is being developed as part of the SC4 Data Architecture Preliminary Work Item project in Work Group 10 – Technical Architecture, of ISO TC184/SC4 – Industrial Data. This paper:

- Describes the motivation for developing the language,
- Provides an outline of the key elements of the language,
- Gives some simple examples of use of the language,
- Gives an outline of the behaviour of an implementation of the language.

2. Background

Following the initial release of ISO 10303 (STEP) in 1995, SC4 established a new Working Group, WG10, to investigate, and attempt to resolve, a number of issues that arose from the initial release. These included:

- STEP Application Protocols are not necessarily interoperable (i.e. they are not subsets of a single integrated data model).
- Not all requirements can be met elegantly by interpreting the Integrated Resources, or upwards compatible extensions of them. This is because of constraints in the Integrated Resources.
- STEP, Plib, Mandate, and ISO 15926 - Integration of life-cycle data for oil and gas production facilities, are not wholly compatible with each other (i.e. they are not integrated).

WG10 established three projects to address some different aspects of these problems:

- SC4 Framework, to look at the scope and overlap of different SC4 projects, to identify potential overlap and consequent potential conflict or synergy,
- STEP Modularisation, to look at relatively short term improvements to the STEP development process that would improve interoperability between Application Protocols,
- SC4 Data Architecture, to take a long term and fundamental approach to resolving the issues outlined above.

EXIST is being developed on an experimental basis as a part of this last project. It may become:

- A statement of requirements for improvements to EXPRESS,
- A separate language from EXPRESS,
- An interesting intellectual exercise.

3. Motivation

Entity Relationship models developed with a language like EXPRESS provide a strong capability for specifying information requirements that can be satisfied in a number of environments, e.g. SDAI, relational database, object oriented database. However, historically Entity Relationship modelling languages arose from the desire to represent the relationships (and their cardinality constraints) between tables in databases. As the desire to create more semantically precise models developed, some limitations to this were noted, and the basic Entity Relationship paradigm was extended to support subtype/supertype relationships. EXPRESS is further extended to support relatively complex constraint

However, there is a basic divide in Entity Relationship models, between the model and the instances of the model and in what can be said at these different levels. Here are some limitations on what can be

Between entities in the model:

- Whilst a sub-type/super-type relationship can be shown, a class-member relationship between entities in the model cannot.

  Instances of the model are shown to be members of one or more entities in the model. However, an

  of showing this.

Between entities in the model, and instances of the model:

An entity and an instance (of another entity) may represent the same class. There is not way of

These constraints on what a model can say may be unimportant for data models that specify the data requirements for a specific application in a specific context. However, they do create difficulty when trying

-
-

  purposes.

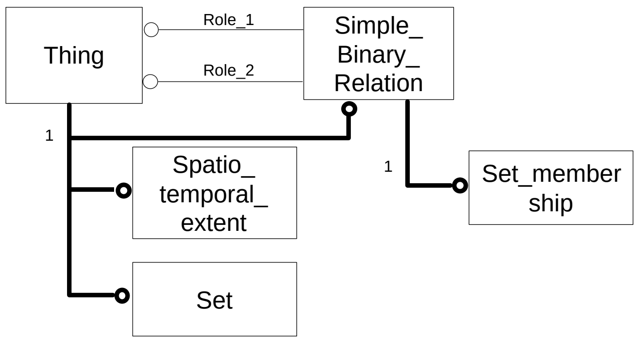

Figure 1: Even an abstract Entity Relationship model has limitations

This causes integration models to become ever more abstract as the range of models they integrate becomes more diverse, and the types of things and relationships become more varied. For example, if two overlapping models have differing levels of abstraction in different parts of the area of overlap, then the higher level of abstraction will need to be taken in each case, if the expressive capability of each model is to be retained.

However, even the most abstract of models, such as that shown in Figure 1, has limitations on what can be said. For example, this model cannot say:

- The entity Thing is a member of the entity Set (I really mean the entity thing itself here, not its members).
- Similarly, the entity Set is a member of the entity Thing (the sub-type/super-type statement says that a member of Set is a member of Thing).

- The entity Set is a member of Set.
- Even at this level of abstraction, some things cannot be said satisfactorily, e.g. ‘there are no unicorns’.

4. Language Objectives

The objectives of the language can be simply stated:

- To be able to express any sort of information, i.e. make any kind of statement. This includes:
    - States of affairs,
    - Rules and constraints,
    - Different levels of abstraction,
    - Mappings between different ways of saying things (inferences),
    - Negative statements.
- To be declarative in nature.
- To be extensible.
- The language should be defined for the following forms (at least):
    - Lexical,
    - Graphical,
    - XML,
    - Meta-model,
    - Interface specification.

5. Key elements of EXIST

The language is based around being able to support logic and set theory. An informal description of some of the main elements is given below, together with examples in the lexical form.

5.1 Objects

Objects can be given a name. In the case of real world objects or classes, the name refers directly to the object. An object name may have a namespace. Within the namespace the name may only refer to one object. However, different names may refer to the same object, and this can be recognised. The character set allowed for EXIST names is restricted, but does include the space character.

Names for EXIST objects use “$” as the namespace.

Example: UK DVLA == UK Drivers & Vehicles Licensing Authority

The “==” indicates that both names are references to the same object.

5.2 Sets and Classes

Some objects may be sets. These fall into two types, those that are defined explicitly by their extent, and those that are named and defined by the common properties of their members, known as classes. A set extent may be assigned a name. For classes members are identified by classification.

Example: $namespace: UK DVLA
car: UK DVLA/N199FOW

The ":" indicates that the object on the right is a member of the class on the left.

Example: rainbow colour := {violet, indigo, blue, green, yellow, orange, red}

The "{ … }" indicates the definition of the structure of a set (the order is not significant). "," is the element separator. The ":=" indicates the assignment of a name to the set extent.

5.3 Tuples

A tuple is an ordered set. That is the order of the elements in the set is significant and an element may appear more than once.

Example: <Matthew West, UK DVLA/N199FOW>

The "< …>" indicate the boundaries of the tuple, "," is the element separator. Note that this tuple has no meaning.

5.4 Relations

A relation is what one thing has to do with another. It takes the form of the classification of a tuple.

Example: ownership: <Matthew West, UK DVLA/N199FOW>

Note: We do not know yet whether the owner is Matthew West, or the car.

5.5 Variables

For the names we have used so far the name always refers to the same object. On the other hand a variable can refer to different objects. In EXIST there are two type of variables:

1. Variables for which different occurrences of the same name refer to the same object for one substitution, e.g. in ($y = x^2 +x$) whilst you can put any number you like in place of "x", you put the same number in each place where "x" occurs. The expression is true for whatever value x takes.

   Example: (vehicle: %x & red: %x)

   The "%" indicates that the name is a variable name, "&" is the logical and operator. This expression could be a condition or constraint for the membership of a class and could be interpreted as "when x is a vehicle and x is red"

2. Variables that only indicate some member of the class can go here. In this case multiple occurrences do not necessarily refer to the same object.

   Example: ownership: <%owner%, %owned%>

   The "%…%" indicates an object that is a member of the class named. Thus we now know that for an ownership relation the first place is for the owning object, and the second place is for the owned object.

5.6 Expressions

There have already been several expressions given as examples in the descriptions above. Expressions in EXIST so far are of three types:

- Logical expressions that evaluate to a true/false value,
- Set expressions that evaluate to a set,
- Arithmetic expressions that evaluate to a number.

5.7 Operators

Operators are key elements in expressions that determine the processing to achieve the resulting value. Some examples of operators in EXIST that have already been met include ":", "&", "==" and ":=". Some others are explained here.

5.7.1 Union

The union operator yields a set which has as its members all those things that are members of the sets on either side of the union operator.

Example: (car .$union. red)

The result of this expression is the set of objects that are members either of the set "car" or the set "red". Note that the operator name is preceded by a "$" which indicates this is an EXIST name. The operator is delimited by period marks. This is the general form for delimiting operators, except in those cases where special characters are used for operators, such as "&".

5.7.2 Implication

One of the most important operators in EXIST is implication. For example, when we know that something is a car, then we know (can infer) that it is also a vehicle because car is a subset of vehicle. This ability to infer something means that we do not have to store the facts we can infer, because we can recreate them from a smaller (canonical) set of facts.

Example: (car: %x & (car .$subset. vehicle)) -> vehicle: %x

The "->" is the implication operator. What is on the right hand side is inferred from what is on the left hand side

5.7.3 Assertions

Logical expressions (propositions) can take a value of true or false. However, we usually want to say that something is true. When we do this we make an assertion.

Example: [car: UK DVLA/N199FOW]

The square brackets indicate that the proposition inside them is asserted to be true. So in this case we are saying that the object referred to by N199FOW in the namespace called UK DVLA is a member of the class car.

6. Processing EXIST

We have examined the motivation behind EXIST and some of its key elements. We now consider how we handle statements in the language however in a short paper we can only attempt to give a flavour of how EXIST is processed and what one can do with it.

Statements in EXIST generally take the form of *assertions* in which we declare a *proposition* to be *true* (or indeed, in some instances, *false*). As propositions are asserted they are added to the set of things that are known to be true (or false).

A *proposition* takes the form of a *relation*. If we assert [R: <a, b>] (for example, [is taller: <John, Fred>]) we are stating that "It is true that a stands in relation R to b" – we are classifying the tuple <a, b> as being a member of the class R. In essence we are defining the relation as being the set (or class) of ordered pairs that possess the property that the first element of the pair stands in the given relation to the second element.

At this point, at a superficial level, it might appear that we are not doing anything that could not be accomplished by using the relational model. A part of the power of EXIST lies in the fact that it allows us to introduce variables into our propositions – armed with variables we can assert both *facts* (propositions without variables) and *rules* (propositions containing variables).

A common form of rule involves *implication* – if P and Q are two propositions, the assertion:

```
[(P) -> (Q)]
```

states that if the proposition P (the *antecedent*) is true then the proposition Q (the *consequent*) must necessarily be true. Let us use as an example of such a rule the concept of specialisation.

If we state that A is a *subtype of* B we are stating that everything that is a member of the class A is necessarily a member of the class B. The statement "everything that is a member of the class A is necessarily a member of the class B" we could state as:

`[(A: %X) -> (B: %X)]`[1]

or 'for each X that is a member of A it is implied it is a member of B', where the notation %X represents a *universal variable* – it corresponds to "everything" in our statement. But this rule is true of all pairs of classes that stand in the relation *subtype of* with respect to each other. We can therefore generalise our rule one stage further and treat the supertype class and the subtype class as variables and can write:

```
[(subtype of: <%A, %B>) -> ((%A: %X) -> (%B: %X))]     (1)
```

or 'for each A that is a subtype of some B it is implied that for each X that is a member of A that X is also a member of B.' How do we process such a rule, and indeed how do we process facts? Let us start by asserting two facts:

```
[subtype of: <cat, animal>]    (2)
[subtype of: <dog, animal>]    (3)
```

If we know nothing else about 'subtype of', 'cat', 'dog' and 'animal' we simply record these assertions as being 'true'. (You can think of it as adding them to a table in a relation database where the name of the table is 'subtype of' and it has columns called something like 'subtype' and 'supertype').

If we now assert rule (1) we do two things. Firstly we tentatively add it to our set of things that we are stating to be true (we add it 'tentatively' because we might find that it contradicts some assertion that we

[1] We will return, in a paragraph or two, to show how we deal with the parts of the statement that read "a member of the class".

have already made). Then we attempt to *match* it against existing facts that we are aware of. In the case of an implication we attempt to match the proposition(s) in the antecedent against facts that are true. In this instance the proposition (subtype of: <%A, %B>) is true for two sets of facts by binding the variables %A and %B to the corresponding elements in the facts. It is true for:

```
a) %A = cat, %B = animal

b) %A = dog, %B = animal
```

For each of these two combinations it therefore follows that the consequent must also be true and we can therefore implicitly insert (that is *infer*):

```
[(cat: %X) -> (animal: %X)]    (4)
[(dog: %X) -> (animal: %X)]    (5)
```

We have thus generated two additional rules. If we now assert, for example:

```
[cat: Felix]                   (6)
```

this fact will trigger the antecedent in rule (4) and the consequent will also be asserted:

```
[animal: Felix]                (7)
```

An assert will not necessarily succeed. In particular, if a proposition has already been asserted to be false, as in:

```
[false: (animal: Felix)]       (8)
```

an attempt to assert that “Felix is an animal” (`[animal: Felix]`) will fail as it would bring about a *logical contradiction*.

If, for example, we already have asserted as facts (6) and (8):

```
[cat: Felix]
[false: (animal: Felix)]
```

prior to attempting to assert (1):

```
[(subtype of: <%A, %B>) -> ((%A: %X) -> (%B: %X))]
```

then the attempt to assert (1) would fail since it would implicitly trigger the assertion of (4) and then (7):

```
[(cat: %X) -> (animal: %X)]
[animal: Felix]
```

which would yield a logical contradiction since it would conflict with `[false: (animal: Felix)]`.

In an earlier example `[(A: %X) -> (B: %X)]`, where the first term, for example, could be read as “everything that is a member of the class A”, we indicated that we would come back to the handling of the that part of the term that talked about “class A“.

It turns out that we can handle it quite simply with another basic assertion:

```
[(%X: %%) -> (class: %X)]      (9)
```

This introduces another form of variable `%%` which is an *anonymous variable* that will match against anything. We can read the assertion as “if anything is classified in terms of X then X is a class”.

Quite clearly we can deal with expressions of much more complexity. For example, we can readily expand the mechanism such that the inference engine can draw inferences across large sets of facts held in conventional relational form. Here matching part of an antecedent that takes the form of a relation with one or more of the elements in the form of a variable is directly analogous to a SELECT statement in SQL. When some variable of the form %X appears in more than one such relation in an antecedent this is directly akin to a JOIN.

By now you should be getting a sufficient feel for the language to begin to see how these (and other) features can be harnessed to provide a powerful set of mapping facilities. Unfortunately space limitations dictate that we will have to cover the subject of mapping in a future paper.

To be able to do anything useful with EXIST we need to be able to interface to the environment in order to be able to exchange data (as indeed we talked about in the previous paragraph when connecting to an RDBMS). The mechanism to achieve this is to provide a small number of built-in functions (akin to the built-in predicates of Prolog). These look to all intents and purposes, like any other function (relation) that we might choose to define in EXIST, but which have a built-in implementation that provides the necessary interaction with the relevant host operating system services.

7. Conclusions

A brief paper such as his can only begin to skim the surface of its subject matter.

The primary motivation behind developing the EXIST language is to provide a mechanism for constructing data models containing statements that are more precise than can be provided by existing data modelling languages. This capability is important, for example, if we are to be able to achieve any real level of interoperability between diverse data models – diverse in terms of any of context, levels of abstraction, source modelling language, etc.

The key elements of the language are based on a small set of fundamental modelling constructs – objects, sets and classes, tuples – augmented by variables and operators to support set and logic operations. Statements in the language may be used to assert diverse sets of facts and rules and thus to construct mappings between models.

An inference engine can process statements in the EXIST language and can operate on sets of facts contained in data repositories, data exchange files, etc.

8. Bibliography

1. http://www.ncl.ac.uk/epistle/ *EPISTLE – European Process Industries STEP Technical Liaison Executive*.

2. ISO 10303-11:1994 Industrial automation systems and integration -- Product data representation and exchange -- Part 11: Description methods: The EXPRESS language reference manual

3. http://www.pdtsolutions.co.uk/iideas/ *IIDEAS – Integration of Industrial Data for Exchange, Access and Sharing*

4. Hodges, Wilfrid *Logic*, Harmondsworth, 1977.

5. http://www.nist.gov/sc4/ *SC4 On-Line Information Services (SOLIS)*.