A Standard Textual Interchange Format for the Object Exchange Model (OEM)

Roy Goldman, Sudarshan Chawathe, Arturo Crespo, Jason McHugh

Database Group
Stanford University

1. Introduction

The Object Exchange Model (OEM) serves as the basic data model in numerous projects of the Stanford University Database Group, including Tsimmis, Lore, and C^3. This document first defines and explains the model, and then it describes a syntax for textually encoding OEM. By adopting this syntax as a standard across all of our OEM projects, we hope to encourage interoperability and also to provide a consistent view of OEM to interested parties outside Stanford.

2. OEM Model

An OEM object contains an object identifier, a descriptive textual label, a type and a value. In most cases, the object identifier is used only within a system implementation and remains hidden from an end user. A value may be atomic or complex. Atomic OEM values can be integers, reals, strings, images, video clips, sound clips, queries, programs, or any other data value that should be considered indivisible by the database. Types are always associated with atomic values, but the different types are not specific to OEM. A complex OEM value, on the other hand, is a collection of 0 or more OEM objects. The complex OEM object can be thought of as the parent of any number of OEM children objects. Note that a single OEM object may have multiple parent objects. With this simple recursive definition, we can build arbitrarily complex OEM networks to model relationships among data.

Slight variations of OEM have evolved across the different Stanford projects. In Lore, labels are actually on parent-child "links" rather than objects. Thus, if an OEM object does indeed have multiple parents, different parent objects may use different labels to identify that object. For example, an atomic value encoding a person's name might be included in one complex object using the label "Author" and in another complex object using the label "Editor." In C^3, additional attributes are required for each object to annotate the changes to the object that have occurred over time.

3. Goals for an Interchange Format

We have established a set of goals for our standard OEM interchange format. We want textual encodings of OEM to be easy to read, easy to edit, and easy to generate or parse by a program. Further, we require the format to be compatible with all current variations of OEM and also to allow the possibility of extensions to the model in the future. (As a related project, we are also working on a standard protocol for exchanging OEM objects over a network.)

4. Syntax Specification

In this section we describe all of the elements of the OEM textual representation syntax via examples and explanations.

Note that all whitespace in textual OEM syntax is ignored, except when part of a quoted string. Comments are allowed within OEM text files using the standard C++ comment symbols and their semantics (`/* */` and `//`).

4.1 Basics

First, we present a very simple example of the syntax to convey the general idea of OEM and the syntax. Please note that the line numbers on the left side of the example are *not* part of the syntax; they are only there to aid the explanation.

```
1     <Birthday {
2           <Month "January">
3           <Day 7>
4           <Year 1972>
5     }>
```

This example encodes a single complex OEM object containing as its value three atomic OEM objects. Each OEM object is encoded with a `<` symbol, followed by a label, a value, and a terminating `>`. The complex object definition beginning on line 1 has `Birthday` as its label. A complex value uses braces (`{` on line 1 and `}` on line 5) to enclose the collection of OEM objects that form that object's value—in this case, the three atomic objects on lines 2-4. On line 2, the atomic object has the label `Month` and a string value of `"January"`. Lines 3 and 4 similarly describe two more atomic objects with integer values. In this case, type specifiers are omitted because they can be inferred directly from the data; section 4.3 describes rules for type specification. This example also shows that object identifiers are not explicitly included in object encodings since they are assigned by the database at load time.

4.2 Syntactic Details

String constants under OEM syntax conform to the definition for string constants in the C language, such as using `\n` to encode a newline. Similarly, integers and reals used to define atomic values follow C syntax for integer and real constants.

Long string constants, such as encodings of large binary objects, will need to span multiple lines to remain readable. To support this, multiple string constants may be concatenated by using the # character in between constants. For example:

```
"ABC" # "DEF" # "GHI" ⇔ "ABCDEFGHI"
```

In general, labels must begin with either an underscore or a letter, followed by any number of letters, digits, or underscores; we refer to this syntactic form as an *identifier*. To support the possibility of more creative labels (in particular those with spaces), OEM labels may optionally be quoted string constants.

4.3 Type Specification

One may optionally specify a type immediately before an atomic value. For example:

```
<Price real 8.95>
```

The types `int`, `real`, and `str` are optional since they can be inferred by examining the value directly. Unlimited additional types may be added, though each must accept values encoded using the syntax of an integer, real, or string constant. For example, a `gif` type might require binary image data to be encoded as hex digits inside a quoted string. A `binary` type, on the other hand, might support the syntax of both integers and strings to encode zero or one. We advocate that all OEM project developers periodically publish new type names and their corresponding encoding methods to encourage continued OEM compatibility. Types must follow the syntax of identifiers, described in 4.2.

4.4 Symbolic Object Identifiers

The final major element of our syntax is the notion of a Symbolic Object Identifier (SymOid). As described earlier, a single OEM object might have multiple parent complex objects. To support this graphical structure in a textual representation, we use SymOid *definitions* to identify objects to be included as children of multiple complex objects, and we use SymOid *references* to indicate additional references to an object defined elsewhere. Here is an example of a SymOid definition:

```
<NINE: Price 9.00>
```

As we can see, a SymOid name followed by a `:` may optionally precede the label of an OEM object. SymOid's, like types, must be identifiers (see section 4.2). Optionally, a second `:` may directly follow the first to specify that the SymOid is *persistent*. In Lore, a persistent SymOid is actually saved as part of the OEM database and may be used as an entry point into the database.

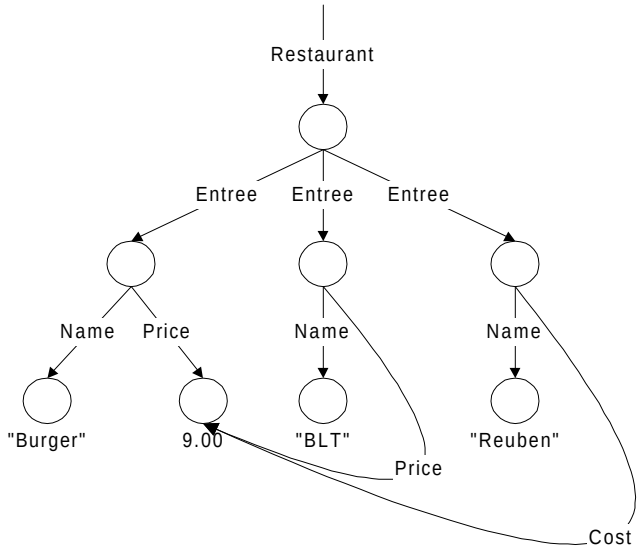
To refer back to an object, use the SymOid preceded by an ampersand:

`<&NINE>` or `<Cost &NINE>`

As described earlier, some interpretations of OEM decouple labels from values. In such cases, a new label may optionally be added for the new reference, as in `<Cost &NINE>`. If no new label is specified, as in `<&NINE>`, the label associated with the object when the SymOid was defined (in this case `Price`) is used for the new reference. Hence, both of the above SymOid references are legal, though in some systems a new specified label might be ignored. The motivation for these differences is best seen with an example:

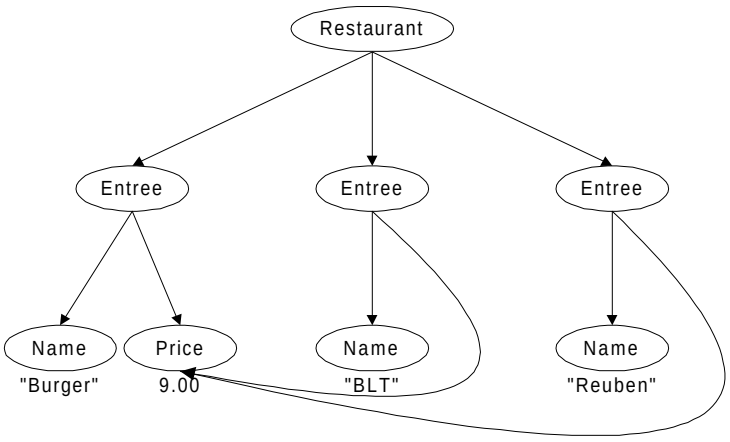
```
<Restaurant {
      <Entree {<Name "Burger"> <NINE: Price 9.00>}>
      <Entree {<Name "BLT">    <&NINE>}>
      <Entree {<Name "Reuben"> <Cost &NINE>}>
      }>
```

The following is the OEM graph constructed from this example in Lore, which places labels on links.



The atomic object with value `9.00`, identified by the SymOid `NINE`, is shared by both the BLT and the Reuben Entrees. Each added reference needs a new label, since none exists in the object itself. Since no label was specified for the BLT's reference to `NINE`, the `Price` label from the original SymOid definition is copied for the new reference. In the case of the Reuben, the new reference to `NINE` is given the label `Cost`.

In contrast, here is the corresponding OEM graph in Tsimmis, which includes labels in objects:



Now that the `Price` label is a part of the object defined by the SymOid `NINE`, we see that any reference to `NINE` must include that label too. Hence, the label `Price` identifies the `9.00` value for both BLT and Reuben Entrees. The `Cost` label in the OEM specification is ignored.

Finally, note that SymOid definitions are not required to lexically precede SymOid references.

4.5 Full Example

Here we provide a more intricate OEM example demonstrating all of the features of the syntax. An explanation follows the example. Recall again that the line numbers are not part of the syntax.

```
1     <DB:: Eats {                  // A Sample Database
2           <Restaurant {
3                 <Name "Darbar">
4                 <Entree {
5                       <Name str "Masala Dosa">   <_895: Price 8.95>
6                 }>
7                 <Entree {
8                       <Name "Mushroom Bhajee">
9                       <Opinion "This entree is excellent, " #
10                                    "though it is a bit spicy">
11                      <&_895>
12                }>
13                <"Credit Card" "Visa">
14          }>
15    }>
```

On line 1, we give the complex object with label `Eats` a persistent SymOid `DB`. In Lore, at least one persistent SymOid is required to serve as an entry point into the database. We also see an example of a comment.

On line 5, we see that we may include two OEM objects on one line. Note that the `str` type specifier is optional. Additionally, we see the definition of the SymOid `_895`. Here we use a non-persistent SymOid since our goal is simply to mark a node for inclusion by another complex object.

On lines 9 and 10, we see an example of string constant concatenation. The # character causes both string constants to be merged together, ignoring the actual carriage return separating the two lines.

On line 11, we see a SymOid reference, linking back to the atomic object defined in line 5.

Finally, on line 13 we see an example of using a quoted string to identify a label.

4.6 Optional Parameters

To support future enhancements, especially for C^3, values in OEM object definitions may be followed by any number of additional parameters. The semantics of these optional values remains unspecified at this point, since none are integral to OEM per se. Each optional parameter may be an integer, real, string constant, or identifier (defined in 4.2).

4.7 Grammar

A full grammar is provided as a reference. In this grammar, [Type] means that Type is optional. Also OEM* means 0 or more OEM elements.

OEM:
 Atomic | Complex | Ref

Atomic:
 < [SymOid:[:]] Label [Type] Value [OptParams] >

Complex:
 < [SymOid:[:]] Label { OEM* } [OptParams] >

Ref:
 < [Label] &SymOid >

SymOid:
 Ident

Label:
 Ident | String

Type:
 Ident

Value:
 Int | Real | String

OptParams:
 (Value | Ident)*

Ident: a letter or underscore followed by any number of letters, digits, or underscore characters. Identifiers are case sensitive.

String: The standard C definition of a string constant. String constants may be concatenated by including a '#' between strings. "ABC" # "DEF" # "GHI" ⇔ "ABCDEFGHI".

Int/Real: Standard C definitions for such constants.

// or /* ... */ may be used as in C++, following identical semantics.