Internet-Draft B. Long
draft-long-external-obj-lang-01.txt
April 29, 1997
Expires October 29, 1997

XODL: External Object Description Language

Status of this memo

ABSTRACT

This document describes a data structure (XODL: Object Description Language) and an associated method which, together, provide a means of representing situations or types of situations. It can thus be used to represent objects, events, or systems of objects and events or types of objects, events or systems. Objects represented can be computer data objects ("stack", "word processor", "user interface", etc.) or "real" objects such as computers, networks, users, and so on.

Table of Contents

1. INTRODUCTION

XODL provides a method of representing situations by representing the different ways information can flow or otherwise be structured. For example, one situation type is where a stack data structure exists. Such a situation is characterized in XODL by the way information is structured; for example, in a stack, the first information "in" is the last information to come "out." Of course that simple description is merely suggestive, but much more complex situations can be well described with XODL.

XODL can describe things done by programs, but the descriptions in XODL are not programs. That is, they are not a list of steps which must be done to produce results. Rather, an XODL description provides a way of formally communicating specifications of what resources are available and how to use them, or of communicating what resources are desired.

XODL interprets the world as if everything in it were pure information. So it may seem at first that only "computer objects" like stacks or user interfaces could be described. However, since physical objects can be simulated, that is, since they can be modeled in terms of information structures, XODL can also describe physical objects such as computers and modems, or even photons and molecules, bank accounts and governments.

2. THE XODL SPECIFICATION

This section describes the XODL language. First, a brief, informal discussion is given. Next, a formal specification of the syntax and semantics for XODL will be given. Though XODL is presented here as a language, it is just as importantly a data structure. That is, when it is readable by humans it is in the form of a language with a syntax using the ASCII character set, but when it is being used by a computer it is in the form of a data structure. Thus, instead of a syntax specification, a data description would do just as well. It is important to keep this in mind, as references to XODL as a data structure will be made as easily as references to it as a language.

After each syntax specification in the formal section, a discussion of the semantics will be given. Some of the semantics will be presented as comments in examples. Parts of this specification were taken from a thesis by Bruce Long for Colorado State University [1].

2.1. Informal Discussion of XODL

XODL code consists of a list of statements called a StatementList. Each statement is called a RelationStmt. A RelationStmt either asserts or denies that two or more pieces of information are identical to each other. A single RelationStmt may imply one or thousands of identities or non-identities. If two pieces of information are identical they can be substituted for each other in any other RelationStmt context. Pieces of information can be constants (7638 or "hello") or names of information (<PersonRec>). Sometimes groups of information pieces can act as a single piece. In such a case, the pieces or names of pieces are enclosed in braces: {<Name>, <Address>, <Age>}. A reference to information in the form of a constant, name or group is called an InfoRef. The important parts of XODL are

StatementLists, RelationStmts, InfoRefs, names, and constants.

Here is a sample StatementList:

```
StatementList Sample1;;
    (<A> == <B>);
    (<A> != <C>);
    (<A> == 8476);
    (<C> == ~EmpRec:Ed Hoolan, 132 Oak St., age 28);
    ([(A==B)] == [(C==D)]);
    (<D> == {<e>, <f>, <g>});
    # MainNet|<TcpIp_net>; //MainNet is a TCP/IP net.
EndList
```

2.1.1. Introduction to XODL Names

Names in XODL can be used to refer to a huge variety of information pieces and resources: Bytes, bits, records, files, disk drives, disks, computers, users, even such things as sums and integrals. Further, the need arises in XODL for names which refer to other names. The needs of XODL require a powerful naming scheme which is not satisfied by naming schemes such as those for URLs or OLE2 Monikers. For example, in XODL it is necessary to have names embedded in "parent" names, and to have a sort of "wild card" in segments of a name other than the last segment. So, for example, there might be a name which would mean the same as something like "C:\(PRG_PATH)*\help.txt", where (PRG_Path) is the name of a path, and the segment with the "*" means "everything here". The example pathname just given is NOT in the syntax of XODL; it is merely an example of the problems the naming scheme in XODL attempts to solve.

The complete syntax of XODL names will be given later. However, a brief primer will be given here. Names in XODL are enclosed in angle brackets, and they consist of segments separated by a slash ('/' or '\') or a dot. Whether a dot or slash is used is important. Thus, <Desktop/CDrive/games> is different from <Desktop/CDrive.games>. The dot separators are generally used to refer to items related to the "slash" names. For example we might have <Desktop/CDrive.FreeSpace> which would not be a file, but probably a number. Names can also be embedded in other names. For example: <<CurrentUser>/BankAccount.Balance>. There is also a method of referring to many items at once (like a complex

wild card), and a method of referring to name segments themselves. These will be presented in detail in the formal specification. Lastly, adding a new segment to a name does not mean a folder or directory is being referenced as in common OS's. It MAY mean that, but it does not have to. Thus, <A/B> may not be related in any way to <A> (though it probably is).

2.2. The Formal Specification

2.2.1. Notational Conventions

The following (modified Back-Naur) notation will be used to specify syntax:

(1) Terminal symbols are enclosed in double quotes.
(2) Non-terminal symbols are alphanumeric or '_'.
(3) Alternative items are separated by '|'.
(4) Items are grouped by enclosing them in parentheses.
(5) Items followed by '!' are optional.
(6) Items followed by '*' can occur zero or more times.
(7) Some items will be explained in English.
(8) Comments are between "//" and the end of the line.
(9) Whitespace is only a separator.
(10) Case is unimportant.
(11) Parameters to an item are in parentheses.
     (See the definition of NameOf.)

2.2.2. StatementLists

****

GENERAL NOTES: The start symbol is StatementList. When a StatementList is being read, whitespace is ignored. Also comments can be added to any line by using a double slash. Comments extend to the end of the line the comment is on.

```
StatementList:
   "StatementList"
       DatabaseID ! ";"
       UsesClause ! ";"
       ( RelationStmt ";") *
       ( "ShortCuts:" (RelationStmt ";")* )!
   "EndList"

DataBaseID:
   The DatabaseID is used to identify the statements to
```

follow. It is not yet formally defined; however, it will have at least a name, a version identifier, and a date. In this way, new versions and extended versions can be identified. The DatabaseID is used in conjunction with the UsesClause.

UsesClause:
The UsesClause, also not yet formally defined, identifies other StatementLists which will be referred to in the current one. Thus, a StatementList about a certain protocol might have the UsesClause "Uses TCPIP" where "TCPIP" is the DatabaseID of another StatementList.

****

SEMANTICS: The tokens "StatementList" and "EndList" identify the start and end of a StatementList.

The RelationStmts before the optional token "Shortcuts" are the main statements. Those after it are called "shortcuts." Shortcuts are statements that hold if the statements above them hold; the shortcuts could be inferred from the regular statements. They are analogous to theorems in mathematics. The shortcuts are used to decrease the amount of time it takes to find a solution to an information structure problem.

2.2.3. RelationStmts

****

```
RelationStmt:
     "(" InfoRef "==" InfoRef ( "==" InfoRef)* ")"
   | "(" MajorTermList "!=" MinorTermList ")"
   | "{" (RelationStmt ";")* "}"
   | "#" NamePart ("," InfoRef)* "|" RelationStmt
   | NameOf(RelationStmt)

MajorTermList, MinorTermList:
   InfoRef ("," InfoRef)*
```

****

SEMANTICS: RelationStmts can assert either that some InfoRefs are identical, or not identical to each other. More will be said about what that means later. The first form of RelationStmt asserts that two or more InfoRefs

are identical to each other. That is, they can be substituted for each other. The second form asserts that two or more InfoRefs are not identical to each other according to the following rule: Each InfoRef in the MajorTermList is asserted to be not identical to 1) every other MajorTerm, and 2) to every MinorTerm. MinorTerms may or may not be identical or not identical to each other. The third form of RelationStmt allows a group of RelationStmts to be asserted as if they were a single RelationStmt. Each RelationStmt in the curly braces ends with a semicolon. While the semicolon is not necessary in the syntax (the end of RelationStmts can be determined without it), I have found that it is a useful visual aid in seeing the end of a RelationStmt.

In addition to the RelationStmts explicitly asserted, it is assumed that InfoRefs in different StatementLists are not identical to each other, unless it is explicitly stated that they are.

The fourth type of RelationStmt will be explained later, after InfoRefs and Names have been explained. It allows for the easy application of universals. That is, it allows types to be instantiated.

Lastly, as with any type of information, a RelationStmt can be given a name, and that name can then be used in any context in which the actual RelationStmt can be.

It will be important to explain exactly what is meant by identity and non-identity. But this is better done after more of the formal aspects are taken care of. Following is an example of the RelationStmts just described. It is a valid StatementList.

Notice that the following example is in XODL, not in the syntax language. That means the quotes have a different meaning which will be explained shortly.

StatementList Example:

```
StatementList Example1; ;  // No UsesClause is needed.
(1, 2, 3 != 4);  // These pieces of information
                      // are not identical to each other.
(1 == "one" == "I");  // These are identical.
(2 == "two" == "II" == "**");
(4 == "IV");
("IV" != 3);  // "IV" is not the same information
```

```
                         // as 3.
   {
      (<Mars> != <Saturn>);
      (<EveningStar> == <MorningStar>);
   }; // These two RelationStmts act as one complex one.

   EndList
```

2.2.4. InfoRefs

```
   *************************

   InfoRef:
        ConstantInfo
      | "{" InfoRef ("," InfoRef)* "}"
      | NameOf(InfoRef)

   SimpleInfoRef:
        ConstantInfo
      | NameOf(SimpleInfoRef)

   *************************
```

ConstantInfo is defined formally below. ConstantInfo is what all InfoRefs eventually terminate in. Or at least it is what they ideally terminate in; information may not be available. For example, I can refer to the reader's shoe size, but I may not be able to access that information. ConstantInfo is actual information. For example: "Hello", or 1273.

If an InfoRef is not ConstantInfo, it might be the name of such information. E.g., <C:/wp/data.doc>. But notice that an InfoRef can be the name of any InfoRef type. This means that an InfoRef might be the name of a name of a name of some ConstantInfo.

Alternatively, there is the notation { a, b, c }. This is a very important feature of the notation. What it means is that all the InfoRefs in the curly braces are to be considered together to count as one single piece of information. Some examples will be given after the definition of ConstantInfo.

Note: The following definitions contain some English.

```
   *************************
```

```
ConstantInfo:
     "'" (Single Quote Delimited Information) "'"
   | """ (Double Quote Delimited Information) """
   | "~" Number ":" (Length Delimited Information)
   | "~" TypeID ":" (Type Delimited Information)
   | (Default Delimited Information)
   | (Token Delimited Information)

Number:
   A string of numeric digits terminated by a nondigit.

TypeID:  Token

Token:
   A case insensitive string of alphanumeric or "_"
   characters terminated by a non-alphanumeric-"_"
   character.
```

****

SEMANTICS: ConstantInfo is actual information. Many of the examples of InfoRefs given so far have been ConstantInfo.

Examples of each one in the order they were listed:

```
'Hello There!'
"1256"
~5:abcde
~AddressRec:1336 Chambers St., Boulder, CO 80303
[34,65, (2+3)]
Bruce
```

There are so many different ways of giving ConstantInfo because there are many different needs. Semantically, they are all equivalent. (In fact, once the algorithm I have written loads them from a file, it does not even use the information about which type was given.) But there are pros and cons to each one. Single quoted information is read until another single quote is found. So if a word in the quotes contains an apostrophe, a problem will occur. For example, 'Bob, don't do that' will be read as 'Bob, don'. Double quotes have a similar problem, but for information which includes double quotes. An alternative for information which may contain both types of quote is to use tacit length delimited information. A "~" marks that either tacit length delimited information or type delimited information follows. If a number comes next, then that number is interpreted as the number of

characters to read as ConstantInfo. Otherwise, a TypeID will tell what type of information follows. The type must have been defined via the language, with a TypeID telling the name of a RelationStmt that defines it.

Often, the program will know ahead of time what type of information is being given as ConstantInfo due to a default type that has been defined. As long as whoever defined the type took care to ensure that the end of the information stream can be formally identified by the system, this information can be given without any delimiting symbols at all. If such care has not been taken, the system may think that characters following the ConstantInfo are part of it.

Token delimited information is where a single token is held to be the information in question. This is handy for simple items such as names or numbers. Thus, "Hello" and Hello (without the quotes) will be semantically equivalent as an InfoRef. Note that since tokens are terminated by certain characters, "Hello There" and Hello There (no quotes) are not equivalent. The second one would be read as "Hello", and the "There" would be a syntax error.

A note about tacit length delimited information: anywhere else in the notation, numbers can be used only after they have been defined, that is, when a StatementList is written which defines numbers and is included in the UsesClause. In the current case, however, the numbers are defined tacitly. This means that expressions and functions cannot be used to specify length. Only a series of digits is allowed, and they will be read as a single base ten number.

SPECIAL INFOREF CONSTANT-INFORMATION

In actual use of the language, many different types of information will be defined and referred to. There is one special case of information which is recognized without being defined within the language. It is hard wired into the language. The information is named by names of the form: <IsKnown/%RelationStmt> where the %RelationStmt is some RelationStmt. The name refers to "Known" if the RelationStmt is implied by the StatementLists asserted; otherwise, it refers to "NotKnown". Because this will be used so often, and it is hard to read in some cases, the format "["

RelationStmt "]" where the RelationStmt is enclosed in brackets will be recognized as well.

EXAMPLES USING INFOREFS

```
   // This says: Something, <PersonRec>, is divided into two
   // non-overlapping parts (<name> and <address>):
   { (<PersonRec> == {<name>, <address>});
     (<name> != <address>); };

   // Something (A) is divided into two parts (B and C)
   // which overlap (a union).
   // The overlapping part is D, while the non overlapping
   // parts are BO and CO:
   { (<A> == {<BO>, <CO>, <D>}); // A is composed of these
                                    // three parts.
       (<BO>, <CO> != <D>);  // They are all three
                                    // separate parts.
       (<B> == {<BO>, <D> } );  // B is composed of DO
                                    // and D.
       (<C> == {<CO>, <D> } );  // C is composed of CO // and D.
    } ;

  // The amount of cash is $56.23:  (<Cash> == "$56.23");

  // The President is Ed Smith ( <President> == <"Ed Smith"> );

  // A three digit number (N) is "123" (said in a hard way)
  // This does not say that N is a number; that would be more
  // complex.
 (<N> == {<Digit1>, <Digit2>, <Digit3>});
 (<Digit1> == 1);  (<Digit2> == 2);
 (<Digit3> == 3);

 // The information that X is zero is not the information
 // that X isone:

 ( [X==zero] != [X==one]);
```

2.2.5. Names

In order for this notation to work, the names used in it must meet several requirements. Neither URL's, PathNames, nor ActiveX monikers meet the requirements. Therefore, the naming system used in this notation is somewhat different. However, the notation could be used to define the syntax and semantics of other types of names, and then they could be used in XODL. They could

be made to fit the syntax defined here by looking like this:

```
   <"http://www.bob.com/index.html">.
```

Following is the syntax for names:

```
************************

NameOf(TypeID):
 // The parameter "TypeID" is used to identify what type
 // will be expected.  But it is processed by the
 // semantic engine, not the syntax checker.  So it will
 // not appear to have any role in defining syntax.

   "<" NamePart ">"

NamePart:
     NameSegment (( "/" | ".") NameSegment)*
   | ("^" NameSegment
      (( "/" | ".") NameSegment)*
      ( "/" | ".") ":" ) )

NameSegment:
   ("%"!  SimpleInfoRef) | ("@" NameOf(path))

************************
```

It is assumed, if a name is used, that it is valid. That is, using a name implies existence. Notice that names are divided into segments similar to DOS or UNIX path names. Notice that the syntax where token delimited ConstantInfo is used as a segment of a name, the colon after "c:" is not allowed. Thus, if we are to use the notation to refer to a drive, we must use a slightly altered form. There are several choices: <"C:/help.txt">, <"C:"/help.txt>, or perhaps <DriveC/help.txt>. The way drives are described via the notation will determine which of the above will work.

Names are the most complex part of the information notation. Names are a single piece of information that is used to refer to another piece of information. However, that single piece might be divided into segments. In fact, a name can be divided into very complex segments. Perhaps the best example of how a name works are the PathNames from the DOS and UNIX operating systems. For example, a filename might be simple: "paper1.doc", or complex:

"C:\WP\misc\thesis\chapter3.doc". Each segment adds to the name. There are several differences between DOS PathNames and the names of the information notation. First of all, where PathNames in DOS refer to a hierarchy of directories and filenames, the names in the information notation refer to a network. For example, in the information notation it is possible that (<C\dir1\text.doc>==<C\dir2\data.txt>). This means more than that the two files contain the same data; it means that they are the same file! Deleting one would delete the other. (Saying that they contain a copy of the same data would be done differently.) Such identities are not possible under the DOS naming system. Another difference has to do with the relationship between segments. In DOS, the relationship between two segments of a name is something like containment. That is to say, for example, that in the name "WP\Paper.doc", "WP" is also a name, and it (WP) contains WP\Paper.doc. For example, if we copy WP, we will copy all of the files it contains. In the information notation, the name WP\Paper.doc would mean that WP\Paper.doc was associated with WP in some way, but not necessarily contained in it. Consider this example: Bruce/L_arm might refer to Bruce's left arm, and Bruce/head to Bruce's head. And when we say "Bruce went to the store" in the language, we will mean that Bruce's head and arms went along also. But consider Bruce/BankAccount. Here, Bruce/BankAccount is associated with Bruce, but when Bruce moves, it does not mean that his bank account goes with him. That is to say that Bruce's head and arms are identical to Bruce in some structured way. But (<Bruce/BankAccount> != <Bruce>).

Let us look closer at the structure of names. First, they are enclosed in the symbols "< >". This enclosure is to distinguish names which are embedded in other names. For example, consider the difference between <Bruce/RightArm> and <Bruce/<BrucesStrongestArm>>. The first name refers (presumably) to my right arm. But the second one refers to my right arm only if (<BrucesStrongestArm>==RightArm).

NAME-PARTS

Inside the "< >" symbols, lies a structure called a NamePart. There are two types of NamePart. The second is syntactically like the first, but with a "^" before it, and where the last NameSegment is a ":". The syntax for the first type of NamePart consists of one or more

NameSegments separated by either a slash ("/") or a period ("."). Though the formal syntax diagram suggests a forward slash, the program will respond to either a forward or backward slash. This provides for names which are similar to DOS path names. Some sample names are <sum/12/5>, and <sum.inverse/10>. Notice that either a slash or a dot can be used to separate name segments. Segments separated with a slash may refer to different information than a name with the same segments separated with a dot. For example, <A/B> is not the same name as <A.B>. This difference will be useful for keeping names organized. For example, we might define that two "slash" segments after "sum" (e.g., <sum/segment1/segment2>) are numeric segments. But it is useful to define a function (name) which is associated with sum to represent negation. If we called it <sum/inverse> we would be contradicting the statement that the segment after "sum" is a number. We can call it <sum.inverse> and avoid the problem. Semantically, names with dots are processed the same way names with slashes are. However, "dot" names are, in this notation, for special cases.

NAME-SEGMENTS

Each segment in a name is a NameSegment. A NameSegment can either be a SimpleInfoRef optionally prefaced by the "%" symbol, or the NameOf a Path which is indicated by the symbol "@". Let us look at what these symbols mean, and what a Path is. There are three cases.

First, a NameSegment might be a SimpleInfoRef without the percent sign in front of it. A SimpleInfoRef is either ConstantInfo (e.g., "DriveC") or the NameOf a SimpleInfoRef (e.g., <BootDrive>). In either case the information given (directly via ConstantInfo or indirectly via name) becomes a segment of the name.

A second kind of NameSegment is a SimpleInfoRef preceded by the "%" symbol. E.g., <Bruce / % appendages>. This is the syntax to specify that a segment is a variable "<%appendages>" in this case is a variable. The possible values that %appendages can have is determined by other RelationStmts. For example we might have: (Recall the special InfoRef "[...]" from the InfoRef section.)

```
(<%appendages> ==
   { [ <%appendages> == LeftArm ],
       [ <%appendages> == RightArm ],
       [ <%appendages> == LeftLeg ],
```

```
          [ <%appendages> == RightLeg ]
      } );
```

which says (do not worry excessively about this yet) that the information of whether <%appendages> is equal to "LeftArm" together with the information about whether it equals the other appendage labels is the entirety of <%appendages>. In this case, <Bruce/<%appendages>> might refer to all of my appendages.

Variables are not the only item that may need to be given a type. NameSegments themselves may need to be typed. Suppose I want to say that certain RelationStmts hold for a NameSegment whenever it is preceded by "<Bruce/Sisters/". I can refer to that segment by adding the symbol "^" to the beginning of the name and a ":" to represent the segment being referred to. For example, I could say that whatever follows "<Bruce/Sister/" is either "Rebecca" or "Valerie" like this:

```
( <^Bruce/Sisters/:> ==
   { [<^Bruce/Sister/:> == "Rebecca"]
      [<^Bruce/Sister/:> == "Valerie"] } );
```

I could then assert that (<Bruce/Sisters> == <Bruce/Sisters/<%Sister>>) so that <Bruce/Sisters> would refer to all of my sisters. I could then refer to all of them, or to each one individually: <Bruce/Sisters/Rebecca>.

The last possibility for a NameSegment is given by the example "@DosPath." This feature can be used to help write StatementLists that work in different situations (e.g., on a different computer) where the name space structure is not known. I will use a computer example for simplicity. Suppose that my DOS directory is in C:/OS/DOS. But most people have their DOS directory in C:/DOS. I can refer to their DOS directory by using the NameOf a Path, as in <@<DosPath>/command.com>. Each name space should have a definition such as:

```
   (<DosPath> == <EnvironmentVars/DosPath>)
```

defined in it.

EMBEDDED NAMES

Consider the SimpleInfoRefs in a name. So far, most of the SimpleInfoRefs we have seen in a name have been token delimited ConstantInfo. For example in "Bruce/Head" "Bruce" and "Head" are tokens. That is, they are a

series of alphanumeric characters or the character "_". But the SimpleInfoRef in a name segment need not be a token, or even ConstantInfo. Further, as was mentioned above, the segments of a name can have types themselves. (To say they have types, is to say that there are RelationStmts that refer to them). Thus, a particular name segment might be a number or a matrix, or vector. For example, <C/SpreadSheets/WorkSheet1/[F,42]> might refer to a particular cell in a spreadsheet. The syntax of the segment "[F,42]" will have to be defined with its own set of RelationStmts.
Reviewing, another possibility is that rather than having actual information, the name of information is used. Suppose we wish to refer to a cell in the above spreadsheet, but the cell we wish to refer to is the one named in another cell. We could do this (leaving off the full name): <...WorkSheet1/<WorkSheet1/[E,10]>>. This would refer to the cell pointed to in cell [E,10].

VARIABLES

Lastly, let us review what the symbol "%" is for. This symbol is perhaps the most powerful of all. It has a job similar to that of the quantifiers of the predicate calculus. It means that the current name segment is a variable. If there are no restrictions on the variable, then it refers to every possible value that the segment can take, rather like "*.*". Thus, we could talk about all the cells in a spreadsheet this way: <WorkSheet/<%X>>; and we should say nothing about the %X. Now suppose we want to refer to only a range of cells, or perhaps every other row, or every checker-board cell, or even cells [A,4], [F,6] and [G,19]. We can define, using RelationStmts which refer to %X, whatever restrictions or patterns we wish. The details of doing this will be touched upon later. We could also make a set of restrictions which would make the variable %X mean "some." Likewise, once numbers are defined, we could use the notation to say "at least 5 cells", "less than ten cells" or even "less than ten, but not exactly 3 cells." And ranges of any complexity can be defined.

Lastly, since %X is a variable, we can use it in more than one place. For example, by using it twice, we could say that the information in each cell in row 5 is identical to the information in the corresponding cell of row 8:

```
( <WorkSheet / [5,%X]> == <WorkSheet / [8, %X]> ).
```

2.2.6 Representing Types

Clearly, if a notation describing objects is to be of any real use, it must be able to work with types as well as actual information structures. For example, we would like to be able to define a system type by stating a list of RelationStmts once, and then applying it to different particulars. Further, we would like to be able to adjust certain aspects of our types that may differ from particular to particular. For example, if we define an array type, we would like to then be able to declare arrays of different sizes and types without changing the definition of arrays.

Recall that in the discussion of the semantics for RelationStmts, we skipped the description of the RelationStmts with the form: "#" NamePart ("," InfoRef)* "|" RelationStmt. Let us consider it now, as it provides us with a way to instantiate types.

It was mentioned that the type of an information structure is given by the structure of the RelationStmts that refer to it or its parts. Thus, if we wish for two structures to be of the same type, we merely assert an isomorphic set of RelationStmts of each one. That is, the RelationStmts asserted of one should be isomorphic to those asserted of the other. For example, if <A> is asserted to be composed of two nonoverlapping sub-parts, then to make <B> be of the same type, we should assert the same things of it as follows:

```
{(<A> == {<R>, <S>}); (<R> != <S>)};// describe A.
{(<B> == {<T>, <U>}); (<T> != <U>)};// describe B.
```

Here, <A> and <B> have isomorphic structures, and are, to that extent, of the same type. Notice that there are several problems with this. First of all, we will have to reproduce all the assertions relevant to a certain type for every item we wish to declare. For example, if we wish to assert that <N> is a number, we shall have to assert the relevant RelationStmts of it using entirely unique names (e.g., the subparts of <A> cannot have the same names as the subparts of <B>). A second problem is that the above solution does not allow for flexible recursive structures. Each level of the structure would have to be defined separately, and thus the number of levels would be fixed and finite.

A third problem, which is even more problematic, is that, when there is an isomorphism between items of equivalent structures, the mapping of the isomorphism is not represented. For example, with <A> and <B> above, does the <R> part of <A> correspond to <T> of <B>, or to <U>? In this case there is no way to tell since the order of the terms in curly braces is not significant. What we need is a method of generating unique names for each new particular's "relateends", while preserving the information about the isomorphism between them.

A handy way of generating unique names associated with a certain named particular system is to add a segment to the name of the particular in question. So, rather than using the names <R> and <S> for the parts of <A>, we could use <A/R>, and <A/S> respectively. Doing likewise for <B> we have the new assertions:

```
{ (<A> == { <A/R>, <A/S>});  (<A/R> != <A/S>); };
{ (<B> == { <B/R>, <B/S>});  (<B/R> != <B/S>); };
```

The "NewLevel" RelationStmts, (as I call them), offer a way to shorten this notation. Notice in the syntax specification, that after the "#" comes a Namepart, followed by a list of InfoRefs, and then a RelationStmt. The semantics are as follows. The RelationStmt part of the NewLevel RelationStmt is asserted in the normal way with the following exceptions. First, any name in the RelationStmt which has the token "parent" as its first segment will have the NamePart of the NewLevel RelationStmt appended where the "parent" is. Second, each of the InfoRefs will be asserted to be identical to a name formed by using the parent NamePart as the first part of the name with a dot segment added to it which identifies which InfoRef it is identical to. The new segment will be "param1" for the first InfoRef (parameter), "param2" for the second one, and so on. An example will clarify this. Consider the following NewLevel RelationStmt:

```
#M, 12, <Bob> | { (<parent>==<parent.param1>);
                  (<Friend>==<parent.param2>);};
```

This RelationStmt generates the following two assertions:

```
   (<M> == <M.param1> == 12);
   (<Friend> == <M.param2> == <Bob>);
```

Thus, we can shorten our original assertions of <A> and <B>:

```
#A | { (<parent>=={<parent/R>, <parent/S>});
       (<parent/R> != <parent/S>); };

#B | { (<parent>=={<parent/R>, <parent/S>});
       (<parent/R> != <parent/S>); };
```

The last problem to solve is the redundant entering of the RelationStmts involving <R> and <S>. Recall that the NameOf a RelationStmt can always be used in any RelationStmt context. Thus, suppose <TwoParts> is the name of the above RelationStmt. We could save producing the relevant RelationStmt multiple times by using its name. Here is the relevant code: Notice how <TwoParts> becomes defined.

```
{
    (<TwoParts> ==
       " { (<parent>=={<parent/R>, <parent/S>}),
           (<parent/R> != <parent/S>) } ");
    #A / <TwoParts>;  // A has two nonoverlapping parts:
                      // <A/R> and <A/S>.
    #B / <TwoParts>;  // B has two nonoverlapping parts:
                      // <B/R> and <B/S>.
}
```

Suppose we wish to define a ball whose size and color are parameters for the type. Skipping much of the detail such as defining numbers-as-sizes, colors and balls, and supposing that the names <size>, and <color> are referenced in the RelationStmt named by <BallType>, the outcome might look like this:

```
{
    #MyBall | <BallType>;
    (<MyBall/size> == 45);
    (<MyBall/color> == red);
}
```

This too can be further reduced by using the list of InfoRefs after the first NamePart. As was explained, these are tacitly assigned names where the first segment is the NamePart, and the second is the dot separated segment <param1>, <param2>, and so on for each InfoRef included. Thus, we can shorten the above RelationStmt to:

```
#MyBall, red, 45 | <BallType>;
```

as long as the necessary changes are made to <BallType> (i.e., add (<parent/size>==<parent.param1>), and so on). In this example, (<MyBall.param1> == red) and (<MyBall.param2> == 45).

3. Availables: Names of Available Information

Many times, it is important for the system utilizing XODL to have access to the information referred to by it. For example, if a piece of information is asserted to be an array with an index ranging from 0 to 10, the "0" and "10" will be needed in the process of marshaling the array. In other words, while the system does not, itself, need to reference the array, it does need to reference the information telling about the array, if it is to successfully marshal the array (or otherwise process it). In this case, the XODL ConstantInfos needed to be referenced.

There are also cases where the information being marshaled needs to be referenced. For example, in a graphics file, the width and height of the graphic need to be ascertained if the graphic is to be marshaled to a screen or printer. The width and height are often stored in the graphics file itself, and thus, the file would need to be accessed if its content is to be marshaled (or otherwise utilized) via XODL.

This type of referencing is done by the use of special names called "availables" which must be hard-wired into an XODL interpreter. Availables are similar to pointers to arrays of bytes. The following rules describe availables.

1) They begin with the segment "avail". E.g. <avail/....>. No other name should be allowed to start with "avail".

2) The allowable values for the second segment are determined by the implementation of XODL. They may refer to memory locations, or perhaps to the results of operating system calls, or something equally useful.

3) One of the values for the second segment is "const". The names beginning <avail/const> are the names of ConstantInfos in the StatementLists currently being used by the interpreter. The names

of ConstantInfos are used to give types to the ConstantInfos. The actual names of particular ConstantInfos can be determined as follows: after the "const" segment, comes the name of the StatementList in which it occurs. The next segment is a number which is determined by the order the ConstantInfos occur in the StatementList - the first ConstantInfo is "1", the second "2", and so on. Thus, the name of the first ConstantInfo in a StatementList named (say) "FTP_protocol" would be:

<avail/const/ftp_protocol/1>.

4) Availables all have two special segments: .data, and .length. the .data segment is a tag for the byte array containing the named information. And the .length segment names an integer which is the length of the data array. Thus, the above ConstantInfo name has the following names associated with it:

<avail/const/ftp_protocol/1.length>
<avail/const/ftp_protocol/1.data>

Suppose that (<.../1.length> == 2). In other words, the length of the .data array is two bytes. Then the following names are valid:

<avail/const/ftp_protocol/1.data/0>
<avail/const/ftp_protocol/1.data/1>

They refer to the 0th byte and the first byte of the data array.

Lastly, each byte is associated with names of each of its bits numbered from 0 to 7. Thus we have names such as:

<avail/const/ftp_protocol/1.data/0/2>
<avail/const/ftp_protocol/1.data/1/7>

which refer to the 2nd bit of byte 0, and the 7th bit of byte 1 respectively. The bits are either 1 or 0.

Reviewing, the availables are used as an interface to any real world information including ConstantInfos. They also may include implementation dependent items such as memory contents or the results of operating system calls. Each available has, at least, a byte array and a length

of the byte array. The structure of the byte array must be specified with other XODL statements.

3.1. Typing ConstantInfos

Consider a RelationStmt with ConstantInfo in it: (<A>==453). Suppose that it is given elsewhere that <A> is a number; it can then be concluded (by substitution of identicals) that the ConstantInfo 453 is a number. But how do the values of the ConstantInfo represent a number? How can the type of the ConstantInfos be specified? That is, how can the structure of the .byte array comprising a ConstantInfo be asserted? Again, how can ConstantInfos appear in RelationStmts? There are three different ways that ConstantInfo can be typed. All three methods have been alluded to earlier in this document. They are:

1) ConstantInfos can be referred to using the naming scheme of section 3. Not only can the entire ConstantInfo be referred to this way, but its byte array and the length of the byte array can be referred to. Such ConstantInfos can be referred to individually to specify the type of a particular ConstantInfo. Referring to ConstantInfos individually in this way is not usually desirable because the name of the ConstantInfo depends upon its location in its StatementList. Any changes made to the StatementList may change the name of its ConstantInfos. A better method is to use a variable to refer to all the ConstantInfos in a StatementList, and assert that [the information that a ConstantInfo is a number (for example)], is identical to [the information that it is related to the byte array in a certain way]. An example of this procedure will be given in the examples of section 4.

2) Recall from the syntax description of ConstantInfos that one of their forms is "~TypeID: Information". Every ConstantInfo of this form causes a statement of the form "# <Name-of-ConstantInfo> | <TypeID>" to be asserted. Thus, suppose a type <3DigitNumber> was defined (as it is in the examples) which specified a number in terms of a byte array. The ConstantInfo "~3DigitNumber:453" would tacitly assert that this case of "453" was a 3DigitNumber.

3) In Names, each segment can be referred to by the notation <^.../:> of section 2.2.5. If a segment is a ConstantInfo, then this segment-name notation can be used

to give a type to the ConstantInfo.

It should be noted that for ConstantInfos where the length of the byte array can be ascertained by the interpreter (e.g., where quotes around the information delimit it), the length will be ascertained automatically. However, with TypeDelimitedInfo the length must be asserted (perhaps calculated) in the associated <TypeID>. The length will then be used to determine how many bytes should be read in by the syntax checker.

Each of these techniques will be illustrated in the examples of the next section.

4. Examples

There are both explicit and implicit reasons for the examples below. Explicitly, each example illustrates how to represent a data type using XODL. Implicitly, some examples will utilize techniques that illustrate such features of XODL as polymorphism, inheritance, and so on.

These examples are intended only to show how XODL can be used to represent complex objects and data structures. They are not intended to describe a standard definition of such items as numbers or arrays. Nothing in the following examples should be interpreted as a description of a standard.

4.1. Enumerations

Suppose we wish to state that a piece of information <day> represents a day of the week. We could assert it with XODL like this:

```
(<day> ==
    { [<day>==Sunday],
      [<day>==Monday],
      [<day>==Tuesday],
      [<day>==Wednesday],
      [<day>==Thursday],
      [<day>==Friday],
      [<day>==Saturday] } );
```

In English this would read "the information <day> is identical to the group of information pieces which answer the following questions: { Is <day> Sunday?, Is <day>

Monday?, Is <day> Tuesday?, ... }" In other words, if you can answer the questions on the right, you know the information on the left, and vice versa.

There is one peculiarity here. The above RelationStmt does not assert that <day> cannot take on values other than the seven given. But if it does take on other values, those values will be informationally equivalent to one of the seven. For example, we might assert without contradiction that:

```
   ([<day>==Thursday] == [<day>==Thur]);
```

which says "the information that <day> is 'Thursday' is identical to the information that <day> is 'Thur' ".

Depending on the circumstances, we may also wish to assert that:

```
   ([<day>==Sunday],[<day>==Monday], [<day>==Tuesday],
    [<day>==Wednesday], [<day>==Thursday],
    [<day>==Friday] != [<day>==Saturday] );
```

which means that none of the above pieces of information are identical to each other. E.g., the information that it is Monday is not the same as the information that it is Tuesday.

4.2. Records and Type-Definitions

Suppose we need to assert that <EmpData> names an employee's name, age and salary. A simple (but not flexible) way would be:

```
   (<EmpData> == { <name>, <age>, <salary> } );
```

And suppose we have defined types <string>, <integer> and <real>. We could then declare the type of the fields:

```
   #name | <string>;
   #age | <integer>;
   #salary | <real>;
```

Notice that the above declaration does not tell how the <EmpData> is mapped to a character array. If such a map is desired, it must be asserted separately. Such examples will be given later in this section.

We have declared a single record, but suppose we need to declare a "type" which is a record with name, age, and salary fields. Section 2.2.6 describes how to represent types. Here is an example:

```
// Define a type <EmpRecord>.
(<EmpRecord> == "
    {
    (<parent> == { <parent/name>, <parent/age>,
                   <parent/salary> } );
    #parent/name | <string>;
    #parent/age | <integer>;
    #parent/salary | <real>;
    } " );

// Emp1 and Emp2 are EmpRecords.
#Emp1 | <EmpRecord>;
#Emp2 | <EmpRecord>;
```

The above XODL code generates the following names:

```
<Emp1/name>, <Emp1/age>, <Emp1/salary>,
<Emp2/name>, <Emp2/age>, and <Emp2/salary>
and <Emp1> and <Emp2>.
```

And it asserts that:

```
(<Emp1> == {<Emp1/name>, <Emp1/age>,
                  <Emp1/salary>} );
```

and similarly for <Emp2>.

Notice that a similar type definition method could have been applied to enumerations.

4.3. Unions, Multiple-Option Types

Often it is necessary for a type to contain one sub-type in one situation, but another sub-type in another situation. XODL can handle such situations in several ways. One method, traditionally called a "union" is to map two different names to the same bytes in a byte array. Consider an example:

Suppose that there is a byte array <record1> whose elements start from zero and are referenced by adding a segment to the name of the array which is the number of the byte to be referenced. For example, the name of byte 0 would be <record1/0>, and of byte 1: <record1/1>. (More will be said of this type of indexing in the next examples.)

And suppose we wish to have some cases where the first two bytes form a 16 bit word, and other cases where they are two bytes. Let us use the names <word1> and <byte1> and <byte2>. And suppose that the (formal) description

of words is that they have segments /hi and /lo to refer to their high and low bytes. We can then map the three names to our byte array like this:

```
(<word1/lo> == <record1/1>);
(<word1/hi> == <record1/0>);

(<byte1> == <record1/0>);
(<byte2> == <record1/1>);
```

We have thus established a traditional union. However, it is often useful for XODL to have a representation of when one interpretation is valid, and when not i.e., a multiple-option type.

In a multiple-option type, some piece of information (a "selector") is used to tell which of the options is the valid one. The selector may be any named piece of information. For the example, let us call the selector <selector>. Here is how to make the above union into a multiple-option type: Suppose that <selector> can either be a 1 or 0.

```
// <selector> is either 1 or 0.
( <selector> ==
    { [(<selector>==1)],
      [(<selector>==0)] } );

// The information that <selector> is 0 is
// identical to the information that ...
([(<selector>==0)] ==
    {
     [(<word1/lo> == <record1/1>)];
     [(<word1/hi> == <record1/0>)];
    });

// and similarly for <selector> == 1:
([(<selector>==1)] ==
    {
     [(<byte1> == <record1/0>)];
     [(<byte2> == <record1/1>)];
    });
```

Multiple-option typing can be used to express the type of ConstantInfos as was described in section 3.1. In other contexts, it can specify the type of information in a network stream or file. For example, using a little English to shorten the example, ([The information that a

file extension is 'gif'] == { Put here: the assertions describing a .gif file}). It is also useful in many other cases, as will be apparent in the following examples.

4.4. Indexing

In the last example, a byte array was discussed. Recall that if the name of the array was <RecByte>, then the names of the elements are <RecByte/0>, <RecByte/1>, and so on. (Arrays need not start with zero.) As it stands, the indexing segment (the 0 or 1) is not known by XODL to be a number. For example, it does not know that 0+1=1 or that 0<1. For all XODL knows, there is an element named <RecByte/jane>. If indexing is to be useful, there must be a way of asserting the type of the indexing segment or segments. This can be done by using the names for segments described in section 2.2.5. The name of the above indexing segment would be <^RecByte/:>. So if we had a description of numbers called <NumType>, we could assert " #^RecByte/: | <NumType>; " to let the XODL system know that RecByte is an array. Other statements could be used to specify the range of valid numbers for the array.

Of course numbers are not the only type that can be indexed upon. By using some other type we can create a map or associative array. Suppose we wish to refer to a color of a geometric figure that varies according to the shape of the figure. We shall call the "color function" <color/%shape>. We need to assert that the "%shape" is either triangle, circle, or square:

```
(<^color/:> ==
    { [(<^color/:> == triangle)],
      [(<^color/:> == square)],
      [(<^color/:> == circle)] } );
```

Next, we can define some values:

```
(<color/triangle>==red);
(<color/circle>==blue);
(<color/square>==blue);
```

The last three assertions could be made without the first one, but XODL would not know that triangle, circle, and square exhausted the possible values for <color/%shape>.

For really useful indexing, such as in arrays, we must have a description of numbers. In the next example, we develop a type definition for numbers, which can then be used as a parent type for bytes, words, reals, and so on.

4.5. Numbers, Encapsulation and Inheritance

Consider the traditional mathematical definition of numbers. The definition relies on a concept called a "group." A group, in mathematics, is defined as something with the following properties where G is a set of symbols, and + or * is an operation on those symbols: (These should seem familiar from algebra.)

1) G is associative, that is, for any x, y, and z from G, (x+y)+z = x+(y+z).

2) One of the symbols in G is such that x+I=x. It is called the Identity Element.

3) Every x in G has an inverse (-x or 1/x) such that x+(-x)=I.

G is called an "Abelian Group" if G is commutative, that is (x+y) = (y+x).

Next in the definition, the group concept is used to define "fields." A field is defined as something meeting the following four requirements where F is a set of symbols with operations sum (+) and product (*):

1) F under + is an abelian group with the identity element "0".

2) The set of symbols F, but without "0" under * is an abelian group with the identity element "1".

3) For all x, y, and z in F, x*(y+z)=x*y + x*z.

4) 0 != 1.

Lastly, a number is defined as an ordered field. "Ordered" means that for any two numbers, the symbols '<' and '>' have their usual meanings of greater than and less than.

Traditionally, a number is something like "THE number 2." That is, THE number 2 has identity. Of course, while we

can find instances of the number 2 in the world, we can not find "THE number two." 2 is called an "abstract entity." XODL cannot represent abstract entities; it can only represent information structures. Therefore, any description of numbers in XODL will have to represent them in terms of information structures. For the following example, let us say that individual numbers have identity. That is, we can refer to this 2 or that 2, but not to "the great number two." This switch will cause the discussion to focus on the operators (sum and product) rather than on the sets of symbols F and G.

In the following description of numbers, notice that groups are defined, then abelian groups are defined by "inheriting" the properties of groups. Next, sum and product are declared, then fields are described. The ordering axioms are given, followed by the declaration of an ordering function (side), then finally, numbers are described.

Notice how the features of a group such as its inverse function are encapsulated with it by appending a new name segment to the group's name. This type of encapsulation will work in many cases. For example, if a stack object is named <stack1>, then it may have the sub-names <stack1/pop> and <stack1/push>

```
StatementList Algebra_Draft; ;

(<GroupOp> == "
   {
   // The operation is associative.
   (<parent/%s1/<parent/%s2/%s3>> ==
           <parent/<parent/%s1/%s2>/%s3>);

   // There is an identity element.
   (<parent.ID> == <param1>);

   // The identity element is the correct type.
   (#parent.ID | <param2>);

   // a0 == a (if the ID is 0).
   (<parent/%s4/<parent.ID>> == <%s4>);

   // e.g.:  a + -a == 0
   (<parent/%s7/<parent.inverse/%s7>> == <parent.ID>);

   // The next group of RS's tell that the operation is
```

```
   // closed.

   // e.g., suppose param2== <number>
   (<parent.GroupType> == <param2>);

   // sum & product are numbers.
   #parent/%s8/%s9 | <param2>;

   // the inverse is a number.
   #parent.inverse | <param2>;

   // The param to Inverse is a number.
   #^parent.inverse/: | <param2>;

   // the first operand is a number.
   #^parent/: | <param2>;

   // the second operand is a number.
   #^parent/%sx/: | <param2>;
   }");

(<AbelianGroupOp> == "
   {

   #parent | <GroupOp>;  // Abelian groups are groups,

   // which are commutative.
   (<parent/%a1/%a2> == <parent/%a2/%a1>);
   }");

(<MultAbGrpOp> == "
   {
   ([<parent/%p3/%p4> != zero] ==
      [<parent/%p3/%p4>==
       <parent.NZProduct/%p3/%p4>]);
   #parent.NZProduct,<param1>,<param2>/<AbelianGroupOp>;
   }");

   // Declare sum and product operators.
   #sum, zero, <NumberType> | <AbelianGroupOp>;
   #product, one, <NumberType> | <MultAbGrpOp>;

(<Field>== "
   {
   (<parent> == <sum/%s1/%s2> == <product/%p1/%p2>);

   // a(b+c) = ab+ac
   (<product/%f1/<sum/%f2/%f3>> ==
```

```
               <sum/<product/%f1/%f2>/<product/%f1/%f3>>);
   ([<parent> == zero] != [<parent> == one]);
   }");

(<OrderRelOp> == "
   {
   (<parent/%o1/%o2>== { // trichotomy
                    [<parent / %o1/%o2> == '<'],
                    [<parent / %o1/%o2> == '>'],
                    [<parent / %o1/%o2> == '='],
                    });
   ([<parent/%o3/%o4>=='<'], [<parent/%o3/%o4>=='>'] !=
                    [<parent/%o3/%o4> == '=']);

   // a<b == b>a
   ([<parent/%o5/%o6> == '<'] == [<parent/%o6/%o5> == '>']);

   // nothing is less than itself.
   (<parent/%o7/%o7> != '<');

   // transitivity
   ({[<parent/%o8/%o9> == '<'], [<parent/%o9/%o10> == '<']}
       =={[<parent/%o8/%o10> == '<'],
          [<parent/%o9/%o10> == '<'],
          [<parent/%o8/%o9> == '<']});

   }");

 // Declare there to be an ordering operator "Side."
 // Side takes two numbers as parameters and refers to "<",
 // ">", or"=".
 #Side | <OrderRelOp>;

(<NumberType>== "
   {
   // The parent is a field.
   #parent | <Field>;

   // The next two RelationStmts synchronize the field
   // ordering operator "Side" with sum and product.
   ([<Side/%o11/%o12> == <'] == [<Side/<sum/%o11/%o13> /
           <sum/%o12/%o13>> == <']);
   ({[<Side/%o14/%o15> == <'], [<Side/zero/ %16> == <']}
       == {[<Side/%o14/%o15> == <'],
           [<Side/zero/%16> == <'],
           [<Side/<product/%14/%16>/<product/%15/%16>
                    == '<'] });
```

```
   }");

Shortcuts:

 // In this section a list of theorems can be given.
 // XODL interpreters should use the shortcuts to make
 // processing more efficient.

   // x*0=0
   (<product/%a/zero> == zero);

EndList
```

Let us work through this StatementList quickly. First, notice that <GroupOp> is declared to refer to a RelationStmt which describes mathematical groups. You can find the various aspects of group operators such as associativity in this definition. Next, a description of Abelian groups is given which references <GroupOp>, then adds one more RelationStmt to it (commutativity). The third description is for a multiplicative Abelian group operator. This operator either returns zero, or acts as an Abelian group for non-zero values.

The next two RelationStmts after the definition of <MultAbGroupOp> refer to the previous descriptions in order to declare the existence of sum and products. Notice that each line tells the name of the operator (sum or product), the identity element for the operator (zero or one), and the type of the operands and results. These names can be used to refer to sums and products. For example, the sum of X and Y could be referred to thusly: <sum/<X>/<Y>>. Shortly we shall see how to refer to actual numbers rather than names of numbers.

Next, sum and product are used to describe fields, and then an "ordering" operator (Side) is defined which takes two numbers (or other entity) and refers to either "<", ">", or "=" depending on whether the first number is less than, greater than, or equal to the second one. For example, (<Side/5/10> == "<"). Notice that while the group definition creates new names, the field definition borrows the names already created to be groups.

Lastly, the field description and the ordering operator are used to describe numbers. This description can be used to declare numbers: e.g., #EmployeeAge | <NumberType> declares <EmployeeAge> to be the name of a

number.

How it Works

The statement #Age | <NumberType> declares that the RelationStmt in <NumberType> is to be asserted in the normal way with the exception that "Age" is substituted in for <parent>. Looking at the information <NumberType>, we see that it contains (among other things) the statement #parent | <Field>. Substituting "Age" for <parent>, we get #Age | <Field>, and see that once again we must dereference a name and substitute "Age" for <parent>. In <Field>, we find the assertion (<parent>==<sum/%s1/%s2>) which is asserted as (<Age>==<sum/%s1/%s2>). Now, we must look at the description of <sum/...>. We find it in the line #sum, zero, <NumberType> | <AbelianGroupOp>. Thus, we must dereference <AbelianGroupOp>. Doing so, we find that sum is a <GroupOp>, and that (<sum/%a1/%a2> == <sum/%a2/%a1>). This means that wherever we find a sum operation, we can switch the operands without affecting the reference of the name. By continuing the process we have been engaged in, we can determine all the valid substitutions which can be made, and thus, which inferences are valid in the logic.

4.6. Functions

Obviously, it is important to be able to represent functions. In XODL, complex names take the place of functions. The arguments to the function are segments of the names.

Notice how we refer to sum to define subtraction, and product for square roots. We will not trace through this code, merely present it. These RelationStmts should be placed into Algebra-Draft. Let us use the abbreviation of "difference" "Diff" to mark subtraction, and Sqr and Sqrt for square and square root.

```
{
// A difference is a rearrangement of a sum operation.
(<%op1> == <sum/ %op3 / %op2>);
(<Diff/ %op1/ %op2> == <%op3>);

// A square of a number is the number times itself.
(<Sqr/ %s1> == <product/ %s1/%s1>);
```

```
// A square root is a rearrangement of a square oper.
(<%s2> == <Sqr/ %s3>)
(<Sqrt/%s2> == <%s3>)
}
```

Given the above RelationStmts, we can now refer to such information items as <Sqrt/9> or <Diff/5/3> which have the meanings Square root of 9, and 5-3. There is reason to believe that XODL can represent the semantics of any finitely specifiable function.

4.7. Syntax for Particular Numbers

The above descriptions allow us to talk about numbers. For example, we could ask what any number multiplied by zero was: <product/%x/ zero>. By substituting the identicals defined in the above definition, we could conclude that the answer was zero. (We would, after many substitutions, be able to substitute "zero" for <product/%x/ zero>.) However, notice a major deficiency: there is no easy way to represent any number other than zero and one. With the notation, we can define descendants of numbers which use the ASCII character set to represent other numbers. First let us describe a single decimal digit <DecDigit>. There are two steps in defining <DecDigit> (other than declaring that it is a number). First, enumerate the possible values a <DecDigit> may take, and second, establish the meanings of the values. You can see the two steps in the following RelationStmt, which we shall add to the definitions of the Algebra_Draft Statement List above.

```
(<DecDigit> == "
   {
   // A <DecDigit> is a number
   #parent | <NumberType>;

   // Specify Possible values:
   (<parent> == { [<parent> == 0 ],
                   [<parent> == 1 ],
                   [<parent> == 2 ],
                   [<parent> == 3 ],
                   [<parent> == 4 ],
                   [<parent> == 5 ],
                   [<parent> == 6 ],
                   [<parent> == 7 ],
                   [<parent> == 8 ],
                   [<parent> == 9 ]
```

```
                 } );
        // Semantics of the values:
        ( [<parent> == 0] == [<parent> == zero] );
        ( [<parent> == 1] == [<parent> == one] );
        ( [<parent> == 2] == [<parent> == <sum/1/1>] );
        ( [<parent> == 3] == [<parent> == <sum/2/1>] );
        ( [<parent> == 4] == [<parent> == <sum/3/1>] );
        ( [<parent> == 5] == [<parent> == <sum/4/1>] );
        ( [<parent> == 6] == [<parent> == <sum/5/1>] );
        ( [<parent> == 7] == [<parent> == <sum/6/1>] );
        ( [<parent> == 8] == [<parent> == <sum/7/1>] );
        ( [<parent> == 9] == [<parent> == <sum/8/1>] );
    }");
```

TYPING CONSTANT-INFOS

Now that a single digit number is defined, we would like to be able to use it in names and other InfoRefs as a ConstantInfo. For example, we might like to use (as has been done earlier without explanation) <Sqrt/9> to mean the square root of 9. But how does XODL know that the second segment of the name <Sqrt/9> is a <DecDigit>? Recall from section 3.1 that every ConstantInfo has a name, and is associated with a byte array (the data field) and a length field. There are two things that need to be done. First, criteria must be asserted whereby XODL can infer that a particular ConstantInfo is a <DecDigit> (or whatever). And second, assert that if a ConstantInfo is a <DecDigit> then it is identical to its data field's byte 0.

Suppose we are creating a statement list named "Stmts" in which we use <DecDigit>s. We can fulfill the first requirement (that the type of ConstantInfos be ascertainable by XODL when necessary) by asserting that {the information that a ConstantInfo is a <NumberType> and that its length field = 1} is identical to the information that it is a <DecDigit>:

```
([{#avail/const/Stmts/%ConstInfo1|<NumberType>;
  (<avail/const/Stmts/%ConstInfo1.length>==1>)] ==
 [#avail/const/Stmts/%ConstInfo1|<DecDigit>]);
```

It may seem circular that we use a "1" in asserting that a ConstantInfo is a <DecDigit>. But in this case, XODL needs only to check for equality, not do an arithmetic operation. Thus, it can tell that the length is 1 without having to look up what the 1 means.

Next, we must assert that any ConstantInfos in Stmts that are <DecDigits> are identical to their byte 0:

```
([#avail/const/Stmts/%ConstInfo2|<DecDigit>] ==
 {[#avail/const/Stmts/%ConstInfo2|<DecDigit>];
  (<avail/const/Stmts/%ConstInfo2> ==
       <avail/const/Stmts/%ConstInfo2.data/0>)} );
```

Notice that the condition that a ConstantInfo is a <DecDigit> is on both sides of the (main) identity symbol. This is because if it were not on the right side, then the information that the ConstantInfo was identical to its 0 byte would be identical to the information that it is a <DecDigit>. But there may be other cases where a ConstantInfo is identical to its 0 byte, but where it is not a <DecDigit>. In other words, we do not want to use a "biconditional" here, and repeating the "antecedent" on the right side of the identity statement removes the biconditionality.

3 DIGIT NUMBERS

Now the description of a <DecDigit> can be used to describe a three digit number. Notice how we can use single digit numbers now as a type of InfoRef. Also, notice that if we want to refer to 10, we must use <sum/9/1> since 10 is a two digit number, which has not been defined.

```
(<3DigitNum> == "
   {
   (<parent> == {<digit1>, <digit2>, <digit3>} );
   # digit1 | <DecDigit>;
   # digit2 | <DecDigit>;
   # digit3 | <DecDigit>;

   //<parent> == ((( D3 * 10) + D2) * 10) + D1
   (<parent> == <sum / <product/<sum/9/1>
                       / <sum/ <digit2>
                        / <product/ <digit3>/ <sum/9/1>>>
                      / <digit1>>);
   } " );
```

In order to map a <3DigitNum> to a byte array, we merely assert that <digit3> is byte 0, <digit2> is byte 1, and <digit1> is byte 2. Notice that in a <3DigitNum>, we must fill empty digits with '0'. E.g., a nine would be "009", not "9".

If we wanted to, we could continue the refinements we have been making to define more syntaxes such as numbers with an arbitrary number of digits, decimal numbers, and so on. We could also use the bit fields in ConstantInfo names to define integers of different sizes, floating-point numbers and so on. In fact, we could create a syntax for complex expressions which result in a number. These expressions might include functions, and even such numbers as pi. Let us quickly consider this so that we may use such a notation without using space here to define it.

EXTENDED NUMBER SYNTAX

While we could use a very simple syntax for InfoRefs which are to be interpreted as numbers (such as <3DigitNum>), complex expressions will be very cluttered looking and hard to comprehend. Therefore, let us assume for the rest of this document that a syntax for numeric InfoRefs called <expression> has been defined in some RelationStmt such that we can use the symbols +, -, *, and /. Of course if we use the slash, we will sometimes have to enclose the expression in quotes to avoid its being mistaken for a name segment separator. Let us also assume that the extended syntax can handle parentheses (which will allow us to use "/" for division without quotes if it is used inside parentheses), and a function notation including the function sqrt() for square roots. Thus, from here on, an example of a valid InfoRef in a numeric context is: (-2.56 + <x>) * sqrt(81) - 3. When the "-" sign is used in a unary position, it will signal that <sum.inverse> is being applied (i.e., for negative numbers).

The idea of defining new syntaxes and semantics via RelationStmts is partly to distance ourselves from the "<.../<...>>" type of syntax which is powerful, but ugly.

4.8. Polymorphism

Polymorphism is the ability of a language to use the same name for similar functions on different types of object. Consider that the name <sum/.../...> as it was defined above takes two <NumberType>s as arguments and produces a <NumberType> in return.

Suppose we wish to have a sum which added two vectors rather than two numbers. I will only discuss doing so

here, not actually do it. If the programmer were clever enough, she could actually define <sum/.../...> once, and have it apply to any system where a non-multiplicative group operation was involved. That is, it could automatically apply to numbers, vectors, matrices, and any other situation where the concept of a sum applies. The types of the argument segments and of the named item (the result) would have to be determined by statements asserting that if the arguments are of a certain type, then the result is of a certain type. Unfortunately, this means a lot more work will have to be done by the XODL interpreter. It may be better to name each different kind of sum a different name. For example, have <sum/.../...> for numeric sums, but have <VectorSum/[a,b,c]/[d,e,f]> for vector sums. The work on the interpreter would be significantly decreased.

4.9. Arrays and Complex References

With a description of numbers along with a syntax and semantics for their representation, we can now use the language to make references that were not available before. For example, suppose we wish to assert that there are 100 computers (or vectors, physical objects, integers, or files etc.). We can use a numeric variable which is limited to numbers from 1 to 100:

```
{
// There are some computers.
#Computers/%x | <CompType>;

// The "%x" is a 3 digit number (recall the definition).
#^Computers/: | <3DigitNum>;

// %x is not greater than 100 or less than 1.
(<Side/%x/100> != ">");
(<Side/%x/1> != "<");
}
```

With this definition, we can refer to such things as the fifth computer: <Computers/5>. Since such arrays are used often, it is handy to generalize the concept of arrays as follows:

```
(<ArrayType> == " {
   // There are some items of type param1
   #parent/%x | <param1>;

   // The "%x" is an integer.
```

```
      #^parent/: | <Integer>;

      // %x is not greater than param3 or less than param2.
      (<Side/%x/param3> != ">");
      (<Side/%x/param2> != "<");
   }");
```

With this definition of <ArrayType>, we can declare arrays easily:

```
// People is an array of 50 <PersonRec>s:
# People, <PersonRec>, 1, 50 | <ArrayType>;
```

Or suppose we assert that there are 100 vectors; we can say (for example) that the 10th through 30th vectors have a zero x component:

```
{
// There are 100 vectors:
#Vectors, <3VectorType>, 1, 100 | <ArrayType>;

// Some vectors' x components are zero.
(<Vectors/%v/x> == zero);

// these vectors are those referenced by numbers <= 30,
(<Side/%v / 30> != ">");

// and >= 10.
(<Side/%v / 10> != "<");
}
```

Consider several more examples:

```
//At least five of the above declared vectors
//(call them XVec) have an x component of (say) 3:
{
#Xvec/%v2 | <3VectorType>;// There are some vectors,
(^Xvec / : | <3DigitNum>;  // which are referenced by a
                           // three digit integer,
(<Side/%v2/5> != "<");  // whose maximum permissible
                           // value is at least 5.
(<Xvec/%v2/x> == 3);  // And these vectors have an x
                           // component of 3.

// None of them are the same vector.
([<%v4> != <%v5>] == [<Xvec/%v4> != <Xvec/%v5>]);

// They are identical to some vectors in <Vectors/...>.
```

```
(<Xvec/%v2> == <Vectors/<%v3>>);
}
```

Another Example:

```
// If a <Vectors> has /x == 3, then its y component ==
// twice its z component

{
([<Vectors/%v6/x> == 3] == {[<Vectors/%v6/y> ==
    (<Vectors/%6/z> * 2)], [<Vectors/%v6/x> == 3] });
}
```

In English, the above RelationStmt reads "The information that a <Vectors> is 3 is identical to the information that its y component is twice its z component and that it its x component is 3." The bit about the x component being 3 needs to be repeated on the right side to avoid a "biconditional" effect where a y's being 2*z implies that x==3.

4.10. Representing Complex Byte Arrays

In many if not most cases, the system being represented is an array of bytes with some complex structure. The array could be a file, a computer's memory, a disk surface, or a stream from a network. Consider how a <file> might be described and mapped to a byte array (let's call it a <ByteStream>. Suppose that we have defined integer to have /hi and /lo fields as was illustrated in section 4.3. Also, let us suppose we have defined <string> to describe an integer (/length) and a byte array (/data) mapped to a byte array in the usual way.

```
(<File> == "
   (<MaxUserName> == 32);  // max length of user name.
   (<MaxFileLen> == 65535);// max length of a file.
   (<MaxNameLen> == 255);  // max length of file name.

   // declare types on all names.
   #parent/FileOwner | <string>;
   #parent/FileName | <string>;
   #parent/FileData | <string>;
   #parent/FileInt | <integer>; // int rep. of FileType

   #parent | <ByteStream>; //the bytes of the file

   // Set max length on names.
```

```
      (<side/<parent/FileOwner/length>/<MaxUserName>>
                                                    != '>');
      (<side/<parent/FileName/length>/<MaxNameLen>>
                                                    != '>');
      (<side/<parent/FileData/length>/<MaxFileLen>>
                                                    != '>');

      // Types of files:
      (<parent/FileType> ==
          {[<parent/FileType> == text ];
           [<parent/FileType> == data ];
           [<parent/FileType> == exec ];
           [<parent/FileType> == gif ];
           <parent/OtherType> } );

   // <FileType> selects type option for FileData:
    // (In an actual implementation, this table would
    // probably be centralized, not in <file>.)
      ([<parent/FileType>==text] ==
          [<parent/FileInt>==0] ==
          [#parent/FileData|<TextFile>]);

      ([<parent/FileType>==data] ==
          [<parent/FileInt>==1] ==
          [#parent/FileData|<DataFile>]);

      ([<parent/FileType>==Exec] ==
          [<parent/FileInt>==2] ==
          [#parent/FileData|<ExecFile>]);

      ([<parent/FileType>==gif] ==
          [<parent/FileInt>==3] ==
          [#parent/FileData|<GifFile>]);

   // Now, map each item to the ByteStream passed in.
      // (There is an easier way to do this mapping,
      // but for the example, I choose the hard way.)

      (<parent/FileInt/hi> == <parent/data/0>);
      (<parent/FileInt/lo> == <parent/data/1>);

      (<parent/FileName/length/hi> == <parent/data/2>);
      (<parent/FileName/length/lo> == <parent/data/3>);
      (<parent/FileName/data/%FnD>==<parent/data/%FnD+4>);

      // record the byte after the filename in NameEnd.
      (<parent/items/NameEnd>==<parent/FileName/length>+4);
```

```
      (<parent/FileOwner/length/hi>==
          <parent/data/<parent/items/NameEnd>>);
      (<parent/FileOwner/length/lo>==
          <parent/data/(<parent/items/NameEnd>+1)>);
      (<parent/FileOwner/data/%FoD> ==
          <parent/data/(%FoD+<parent/items/NameEnd>+2));

      // record the byte after the file owner.
      (<parent/items/OwnerEnd>==
                    (<parent/items/NameEnd> +
                     <parent/FileOwner/length>+2));

      (<parent/FileData/length/hi> ==
          <parent/data/<parent/items/OwnerEnd>>);
      (<parent/FileData/length/lo> ==
          <parent/data/(<parent/items/OwnerEnd>+1)>);
      (<parent/FileData/data/%FdD> ==
          <parent/data/(%FdD+<parent/items/OwnerEnd>+2)>);

   ");
```

Suppose that a byte stream called "MyFile" is mapped to an available byte stream. (Recall, available items are those which are directly accessible to the XODL interpreter. Then, using the above description of a file we could assert:

```
      # MyFile | <File>;
```

Doing so would provide names of all the parts in MyFile, along with a way for XODL to access those named pieces of information via the available MyFile is mapped to.

5. Interpreting XODL

There are many different ways that XODL interpreters may be implemented, but the basic process must be the same: substitute identicals and check non-identicals to move information around. There are two related problems to solve: The first is, how do we use XODL to represent problems we wish to have solved. Let us look at this problem first.

5.1 How do XODL interpreters solve problems?

It may seem that there are many different ways to use XODL to solve problems. While there are many different ways to implement XODL interpreters, the different

problems that can be solved with XODL can be generalized and solved with a single algorithm. Let us consider the algorithm in term of its inputs (arguments). Let us call the algorithm task.engage.

The inputs necessary for an XODL interpreter can be divided into two parts: problem lists, and tacit lists. Each of these two parts contains three lists, all of which are XODL StatementLists: Assertions, Capabilities, and Tasks. Thus, the inputs to task.engage consist of six lists. Assertions are those statements which describe the world to the interpreter. They mostly declare the existence of objects. For example, a StatementList might use the names of various computers, network connection, users, programs, and so on.

Capabilities are statements which are not necessarily true, but which could be made true by the XODL interpreter if need be. For example, a capability might be to make certain operating system calls. What those calls actually do is specified in the assertions. An example might be of the form ([input to op-sys call] == 432);. Of course that is greatly simplified.

Lastly, besides assertions and capabilities, there are tasks. A task list is a StatementList which, rather than describing the way the world IS, or the way the interpreter COULD make it, describes the way we would LIKE the world to be. For example, I might like to have a copy of a certain file or record on my hard drive or in a certain document. I.e., the information on the drive == certain other information. Or I might like to have a system set up whereby I can edit a certain file.

5.2 A Simple Example

Suppose the inputs to task.engage are as follows:

```
***************
   StatementList // Asserts.smt
   (a==b==c);
   (B==D);
   (X==Y);
   EndList

   StatementList //Test Capability List
   (y==t);
   (x==w);
```

```
(d==x);
(a==q);
(b==s);
EndList

StatementList // A Task
(A==Y);
EndList
```

****

The interpreter's job is to make sure that the task (A==Y) becomes the case. First, it (the interpreter) might look in the assertions to see if (A==Y) is already the case. It may need to do some substitution of identicals to do this. For example, it is asserted that (Y==X); if it is also the case that (X==A) then (A==Y) is already true and the interpreter need not act at all. Alas, (A==Y) is not the case according to the above assertions. The capabilities are items that the interpreter can MAKE true. In this case, making the third capability, (d==x), true, will complete the task since (D==B), and (b==a). In a real case, names might include URLs, references to the internals of documents, or to people. And the solution may consist of hundreds of steps of capabilities to do.

Task.engage has two parts: first, find a suitable solution, and second, make that solution so.

5.3 Internal problems

Task.engage is not merely the way a user or programmer interacts with the XODL interpreter. It is a vital part of how XODL is to BE interpreted. Consider an example. Suppose a task consists of retrieving a certain piece of information which is in an array. E.g., <PersonRec/53>. Now suppose that the index (the 53) was not directly known, but is stored in a document on the web. The reference to the information might be something like:

```
<PersonRec/<"http://.../WorkSheet"/[C,32]>>
```

The interpreter will have to retrieve the WorkSheet, or at least the contents of cell [C,32], and this will be a separate task. Thus, in almost any task, there are sub-tasks which the interpreter will have to generate for recursive calls to task.engage.

The problem is that the simple assertions and capabilities passed in to the top level task.engage may not cover such things as looking up items on the web, or parsing spreadsheets. This is the reason for the "tacit" assertions, capabilities and tasks. Tacit assertions and capabilities are those items which the interpreter may use in solving any problem. For example, in the case where a problem references some information on the Internet, StatementLists describing TCP/IP, HTTP and so on may be needed to reference that information.

Tacit tasks are a security measure. The tacit tasks will often be negative, that is of the form (A!=B). For example, tacit tasks might be "do not erase the hard drive", "do not try to guess passwords", "do not try to thwart system security." Using tacit tasks to increase security is only a precaution. It should not be the main method of directing the actions of the interpreter. See the section on security considerations for more information.

5.4 Availables revisited

Recall that availables are named items to which the interpreter has access. The question is, what is the nature of this access? The answer is, whenever a piece of information resides in available memory, an implicit StatementList is being asserted. Suppose that <byte1> named an available byte somewhere in system memory. And suppose that memory cell contained the number 255. Then the statement(<byte1>==255) would automatically be asserted, as well as (<byte1/bits/0>==1) and so on for all eight bits. Thus, if the interpreter needs to access a piece of information, it can engage a task to move that information into an available memory location, and then read it from there.

5.5 An Example

Suppose we wish to have the XODL interpreter design an AND gate. There are many different ways that AND gates can be created: out of transistors and resistors, via software connected to an I/O port, out of NAND or NOR gates, and so on. We can select which of the possible solutions XODL will use by limiting the capabilities it has to work with. Let us look at how XODL might construct an AND gate out of NOR gates.

The first step is to describe AND and NOR gates. Let us assume that our gates are ideal in the sense that there is no time lag between when the signal arrives at the gate and when the new signal leaves. We can index the gates over time. That is, we can act as though we have an array of gates where each one is at a different time. The reference to the inputs of the gates have the form:

<And/%t/Input1> and <Nor/%t/Input1>, and so on for Input2, and output.

Where %t is the time. Let us consider the three StatementLists for this problem: Assertions, Capabilities, and Tasks.

```
///////////////////////////////

   StatementList And_Nor_Assertions;;
   // Type description for a two state system:
   (<BinDigit> ==
      "{(<parent> ==
          { [<parent>==0], [<parent>==1] });
       ([(<parent>==0)] != [(<parent>==1)]);}");

   // Describing AND:
   (<And_type> == "{
      # parent/%t1/Input1|<BinDigit>; // input1 is 1 or 0.
      # parent/%t2/Input2|<BinDigit>; // input2 is 1 or 0.
      # parent/%t3/output|<BinDigit>; // result is 1 or 0.

      // The result is 1 if & only if both inputs are 1.
      ([<parent/%t4/output> == 1] ==
                 {[<parent/%t4/Input1>==1],
                  [<parent/%t4/Input2>==1]});
   }");

   // Describing NOR:
   (<NOR_type> == "{
      # parent/%t1/Input1|<BinDigit>; // input1 is 1 or 0.
      # parent/%t2/Input2|<BinDigit>; // input2 is 1 or 0.
      # parent/%t3/output|<BinDigit>; // result is 1 or 0.

      // The result is 1 if & only if both inputs are 0.
      ([<parent/%t4/output> == 1] ==
                 {[<parent/%t4/Input1>==0],
                  [<parent/%t4/Input2>==0]});
   }");
```

```
   // There are two inputs In1 and In2 which are either
   // 1 or 0 depending on the time.
   # In1/%t1 | <BinDigit>;
   # In2/%t1 | <BinDigit>;
   # Out1/%t1| <BinDigit>; // the output is binary.

   // There are three Nor Gates:
   // (Recall the Array example.)
   #Nor, <NOR_Type>, 1, 3 | <ArrayType>;

   EndList

/////////////////////////////////

   StatementList And_Nor_Capabilities;;

   // In1 can attach to any gate input:
   (<In1/%t1>==<Nor/%n1/%t1/Input1>);
   (<In1/%t1>==<Nor/%n2/%t1/Input2>);

   // In2 can attach to any gate input:
   (<In2/%t1>==<Nor/%n3/%t1/Input1>);
   (<In2/%t1>==<Nor/%n4/%t1/Input2>);

   // Gate outputs can attach to gate inputs:
   (<Nor/%n5/%t1/output>==<Nor/%n6/%t1/Input1>);
   (<Nor/%n7/%t1/output>==<Nor/%n8/%t1/Input2>);

   // Out1 can attach to any gate output:
   (<Out1/%t1>==<Nor/%n9/%t1/output>);

   EndList

/////////////////////////////////

   StatementList And_Nor_Task;;

   // Create a new AND gate,
   #NewAnd | <And_Type>;

   // where in1, in2, and out1 are the I/O.
   (<In1/%t1>==<NewAnd/%t1/Input1>);
   (<In2/%t1>==<NewAnd/%t1/Input1>);
   (<Out1/%t1>==<NewAnd/%t1/output>);

   EndList

/////////////////////////////////
```

Given the above inputs, an XODL interpreter should produce a list of statements similar to the following one:

```
{
(<Nor/1/%t1/Input1> == <In1/%t1>);
(<Nor/1/%t1/Input2> == <In1/%t1>);
(<Nor/2/%t1/Input1> == <In2/%t1>);
(<Nor/2/%t1/Input2> == <In2/%t1>);
(<Nor/3/%t1/Input1> == <Nor/1/%t1/output>);
(<Nor/3/%t1/Input2> == <Nor/2/%t1/output>);
(<Nor/3/%t1/output> == <Out1/%t1>);
}
```

It can be shown that the above RelationStmt will produce an AND gate if it is instantiated.

6. References

[1] Bruce Long, "The Concept of Causality in Ethical Theories", not published. This paper can be requested from xbruce@dimensional.com.

7. Security Considerations

XODL interpreters are given a specification of a state of affairs, and they try to find a sequence of actions which will bring that state of affairs to be. Different interpreters, or interpreters with slightly different StatementList driving them will come up with different sequences of actions. This means that special care must be taken to ensure that none of the actions taken are harmful, illegal, or immoral. For example, if a local machine did not have enough resources to complete a task, one solution would be to hack into another computer and use its resources via a hacked password.

While such scenarios must be watched for, they need not cause a panic. There are several ways to control the behavior of XODL interpreters.

1) Do not give it access to certain resources.
2) Do not give it the capability of accessing them.
3) Do not give it the knowledge (StatementLists) of them.
4) Use tacit tasks to forbid certain actions.
5) Program the interpreter to follow a system of values.

The last item may sound hard to do, but I have developed

a theory which will allow a complex value system to be "digitized." This theory can easily be hardwired into an XODL interpreter such that any action taken would conform to the stored value system. Such a system would probably be quite secure, as the XODL interpreter could be on the look out for items which would cause problems, and prevent them; whether they were rogue StatementLists or other programs. This theory is partly documented in [1]. More information will be forthcoming, and will be posted at http://www.dimensional.com/~xbruce/.

8. Author's Address

Bruce Long
1335 Chambers Drive
Boulder, CO 80303

Phone: 303/494-3985
E-mail: xbruce@dimensional.com