*Closing the Loop: Considerations for
Integrating Multiple Applications that
Implement the Same Express Specification*

RECEIVED
AUG 17 1998
OSTI

MASTER

HH
DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos
NATIONAL LABORATORY

*Los Alamos National Laboratory is operated by the University of California
for the United States Department of Energy under contract W-7405-ENG-36.*

An Affirmative Action/Equal Opportunity Employer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither The Regents of the University of California, the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by The Regents of the University of California, the United States Government, or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of The Regents of the University of California, the United States Government, or any agency thereof. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

DISCLAIMER

**Portions of this document may be illegible
electronic image products. Images are
produced from the best available original
document.**

*Closing the Loop: Considerations for
Integrating Multiple Applications that
Implement the Same Express Specification*

Ron Daniel, Jr.

Closing the Loop: Considerations for Integrating Multiple Applications that Implement the Same Express Specification

by
Ron Daniel Jr.

Abstract

This paper is a post-mortem report on one phase of a complex project, the Advanced Reservoir Management (ARM) project [1], more commonly known as the "Oil and Gas" project. This particular phase, developing an Express-to-IDL compiler, was a high-risk/high-return attempt to implement a solution for the Oil and Gas project that would greatly ease the development of the virtual organization tools, and could be easily re-used in a weapons-related project. The potential leverage between these two projects was the major motivation for the task. Section 1 briefly describes the two projects, and the consequent motivation for developing the compiler. Section 2 describes the development process for the compiler and some of the minor difficulties along the way. Section 3 elaborates on the major difficulty uncovered once we had the compiler. Section 4 outlines considerations for projects that might want to use the technology.

Executive Summary

Express is a data definition language that is part of ISO 10303, better known as STEP (STandard for the Exchange of Product data). STEP is of interest to the Lab's stewardship mission because CAD packages, such as Pro-Engineer (Pro-E), are able to read and write data according to the STEP specification as well as in proprietary formats. Express has been used in the development of other standards, in particular the Epicentre data model from POSC (Petrotechnical Open Software Corporation). The Lab has been investigating the Epicentre model as the basis for repositories in the "virtual enterprise" effort being funded by the Advanced Reservoir Management (ARM) project.

In an attempt to leverage work on the "Gas and Oil" project (as ARM is also known) to benefit the weapons mission of the Lab, I was asked to investigate the possibility of developing an Express to CORBA IDL translator. (CORBA is the Common Object Request Broker Architecture, a standard for the construction of distributed object systems. IDL is the Interface Definition Language, which is used to define the interfaces between the distributed objects.) The initial purpose of the translator would be to provide IDL interfaces to the data elements in the POSC model, and thus to object-oriented (OO) databases, etc. The leverage comes from the fact that the GWIS (Global Weapons Information System) project was already using CORBA as the basis for their architecture. If the translator could be successfully applied to STEP models produced from CAD packages, GWIS might be able to take a large leap forward for minimal additional cost by having IDL access to the internals of the CAD models of the weapon systems. This was conceived as a high-risk/high-reward experimental activity.

Late in FY97 I began to look into developing the translator. I located a tool that was part of a commercial software package the Lab already had a license for (ST-Developer 1.5) and was able to translate the Epicentre Express model into a set of IDL files within only a matter of days. Then the fun began. The Epicentre data model is very complex and resulted in the creation of approximately 1600 IDL files. Unfortunately, the generated IDL was buggy and an employee of the vendor stated that fixing that particular part of the larger package was a very low priority. I wrote a set of PERL scripts that cleaned up the IDL and was able to use the Visigenic product to compile the clean IDL into client-side stubs and server-side skeletons. This resulted in approximately 15,000 Java source files and approximately

15,000 class files (compiled Java bytecodes). Those familiar with CORBA development will know that this still leaves the matter of actually implementing the server-side functionality, which in this case means modifying the generated server-side skeleton code to talk with the software that actually does the storage of the Epicentre data. This was where we encountered the major difficulty in the project, which brings us to the point of this paper.

In addition to the Epicentre Express model, POSC provides "projections" of that model onto several standard relational databases, such as Oracle, Sybase, and ANSI-standard Structured Query Language (SQL). Unfortunately, additional information is introduced in the projection process. One example of this is that names of data entities may be abbreviated to fit into length restrictions of databases. A more serious problem is that many-to-many relationships between data entities result in the construction of intersection tables in the database, and these intersection tables are essentially invisible from the viewpoint of the Express specification of the data model. In order for the server-side code to be able to map between an IDL call like `getWellLogTrace()`, and the SQL queries actually needed to get that information out of the database, the code has to have complete knowledge of what is going on inside the database. If there were only a small number of interfaces, a person could figure this out, but for a model the size of Epicentre only automated means are feasible. We refer to this task of linking between two different implementations of the same logical model as "closing the loop."

The major lesson of this paper is that Express models are, by design, not a complete specification of an implementation. If one wants to automatically generate code to deal with implementations of an Express model, one must also have access to a machine-readable specification of all the additional choices that were made in implementing the model. This has implications for the weapons work. If the approach of using IDL interfaces to the CAD models is still of interest (and I believe it is still worth looking into), the first order of business would be to see what additional choices were made by the developers of Pro-E and estimate the difficulty of formalizing those changes in such a way that the loop can be closed. A second alternative to be pursued is to investigate the SDAI (Standard Data Access Interface) portion of the STEP standard. That alternative was not investigated during this project. If a repository already provides an SDAI interface, it may simplify the communications with the repository. If, on the other hand, we have to develop the SDAI interface for a particular repository, exactly the same problem of needing complete knowledge of the actual internals of the repository would apply.

1.0 Introduction

This paper is a post-mortem report on one phase of a complex project, the Advanced Reservoir Management (ARM) project [1], more commonly known as the "Oil and Gas" project. This particular phase, developing an Express-to-IDL compiler, was a high-risk/high-return attempt to implement a solution for the Oil and Gas project that would greatly ease the development of the virtual organization tools, and could be easily re-used in a weapons-related project. The potential leverage between these two projects was the major motivation for the task. Section 1 briefly describes the two projects, and the consequent motivation for developing the compiler. Section 2 describes the development process for the compiler and some of the minor difficulties along the way. Section 3 elaborates on the major difficulty uncovered once we had the compiler. Section 4 outlines considerations for projects that might want to use the technology.

1.1 GWIS

The GWIS (Global Weapons Information System) [2] effort has been using CORBA-style distributed objects as the basis of their approach in developing a system for tracking weapons design drawings, test results, etc. CORBA (Common Object Request Broker

Architecture) [3] allows one to define interfaces using an Interface Definition Language (IDL) that looks very similar to C++ header files. Once the interfaces are established, the actual implementation of an object can be hidden. It may be written in FORTRAN, C++, Java, or any other language with bindings to IDL. Interfaces can be defined to "wrap" existing systems such as simulation codes, databases, or data files.

GWIS would like to have CORBA interfaces defined for the CAD (Computer-Aided Design) models of weapons, in order to be able to extract design information and get it into a form needed by other objects in the system, such as simulation codes. However, since different packages have different proprietary formats and different internals, this could become quite an extensive effort. Fortunately, many CAD packages have begun supporting STEP (ISO 10303 - STandard for the Exchange of Product data) as a neutral format. If the CORBA interfaces could also conform to the STEP standard, then the GWIS effort would have a valuable degree of isolation from changes in proprietary formats.

STEP is a large, complex standard. One part of it is a data description language known as Express. Express allows one to define data entities that inherit properties from other entities and to impose constraints on relationships between entities (e.g., If I have a part description, there needs to be a matching part number in the table of known parts). Listing 1 provides a portion of an Express model to give a flavor of its capabilities. Describing Express is outside the scope of this paper so the interested reader is referred to ISO 10303 for the definitive description.

```
ENTITY Well_log
  SUBTYPE OF (
    business_object
  );
  kind : OPTIONAL ref_sonde_log_type ;
  process_state : OPTIONAL ref_well_log_process_state ;
  u_wellbore : OPTIONAL wellbore ;
INVERSE
  u_logging_pass_interval : SET [0:?] OF logging_pass_interval FOR well_log ;
  u_equipment_item : SET [0:?] OF equipment_item FOR well_log ;
  u_well_log_trace : SET [0:?] OF well_log_trace FOR well_log ;
  u_pty_dataset_creation : SET [0:?] OF pty_dataset_creation FOR well_log ;
UNIQUE
  ni: kind,
      process_state,
      wellbore;
END_ENTITY;
```

Listing 1: Sample portion of an express model.

Express has been used to define other data models. In particular, Express was used by POSC (Petrotechnical Open Software Corporation) to define the Epicentre data model [4]. Epicentre is a complex model relating many of the data sets used in the petroleum industries, such as well logs, drill rig information, lease information etc. (Listing 1 is actually drawn from the Epicentre model.)

1.2 Advanced Reservoir Management

A separate project at LANL, the Advanced Reservoir Management (ARM) project [4,5], has been tasked to develop a software system for enabling a "virtual corporation." The idea is that an independent oil or gas producer can contract with different analytical service organizations to take existing well log information, come up with a reservoir model, and then a production plan based upon that model. These separate organizations would utilize a common data representation when exchanging information. Rather than rely on a

centralized data store, the producer would obtain "tickets" upon payment for services rendered and could pass those tickets along to the next analytical service. That service would then obtain the necessary information from a database at the previous analytical service, perform their work, put it into their own repository, and give the producer tickets to access it once payment was received. Since POSC's Epicentre is an emerging standard in the petroleum industry, it is a natural choice for the common data representation needed to make such a scheme work.

Like all Express models, Epicentre defines a logical data model that could be implemented a number of different ways. In some sense this is analogous to the interfaces in a CORBA system. If we can agree on the interfaces, then it doesn't matter too much how the machinery behind those interfaces is implemented.

While Epicentre is a logical data model, it does not define an actual API (Application Programming Interface). In other words, while the data model defines a `WellLog` data element, it does not define calls such as `getWellLog()` or `putWellLog()`. Without a standard API, client programs do not have an interoperable set of calls to make when interacting with a repository. One of the necessary orders of business for the Oil and Gas project was to define such an interface. This would allow an application to make a standard call (i.e. `getWellLog()`) and get the same information from any of the repositories in the virtual corporation.

1.3 Motivation for an Express to IDL Compiler

One way to develop such an API would be to somehow convert the information in the Express specification of the Epicentre model into a set of CORBA IDL definitions. We would then have a set of calls that could be expressed in a number of programming languages, allowing clients to be written in Java while the server side was written in C++ or FORTRAN.

While it would be possible to go through the Express definition of a model by hand, manually generating the corresponding IDL, it would be a very tedious job. Also, since the Epicentre data model defines about 1,500 data elements and undergoes periodic revision, it would be difficult to complete the interface before it became obsolete. However, the content of an Express file bears many similarities to a set of IDL files. If an Express to IDL compiler could be developed, we could easily generate the APIs for the virtual organization project and keep them updated as Epicentre evolved. Furthermore, we might be able to take that compiler and apply it to the needs of GWIS for access to the internals of CAD models. This potential leverage was a major motivation for this experiment.

2.0 Development Procedure

In the summer of 1977, Bob Webster of XCM contacted me about developing such a compiler. I agreed to look into it beginning in August. Early in August of 1997, I began working on the project. Since I'm lazy by nature, my first step was to see if there were commercial tools that could perform this task, or at least assist in its development. Relatively quickly I found that a company called STEPtools had a product called `ST_Developer` that included an Express to IDL compiler. Furthermore, the Lab already had a license for version 1.5 of that product. Once that tool was located, Beth Lee and I were very quickly able to generate a set of IDL files from the Express definition of the Epicentre data model. This milestone was achieved on August 6. Rather than marking the completion of the job, this was only the beginning of the fun.

The output generated by the `exp2idl` compiler was approximately 1600 IDL files, many of which contained errors of one form or another. The most common error was that the compiler would generate illegal IDL statements of the form:


```
attribute sequence<idl_Well_log> well_log;
```

which must be expressed as two statements of the form:

```
typedef sequence< idl_Well_log > sq_idl_Well_log;  
[...]  
attribute sq_idl_Well_log well_log;
```

in order to be legal IDL. There were other forms of errors as well, all of which would be relatively simple to repair through textual manipulations.

I created a PERL script that would automatically clean up the output of the exp2idl compiler. A listing of that script is attached as Appendix A. With a small amount of hand-cleaning after using the scripts, the IDL was legal and could be compiled by the Visigenic Visibroker for Java compiler. The resulting Java files were then compiled to Java .class files.

The major difficulty in this subtask was dealing with the size of the data model and the large number of files that were generated. As mentioned earlier, the Epicentre data model resulted in approximately 1,600 IDL files, which expanded to almost 15,000 Java source files and a similar number of Java class files. Compilation on a basic Sun workstation using the Javac compiler from the Java Development Kit (JDK) 1.1.2 took the better part of a week. Eventually the files were all compiled on a multiple-processor Windows NT box using Symantec's Java compiler, which is much faster than the free compiler from Sun. On this system the compiles completed over a weekend. (As a side note, an attempt to run the compiles on a Windows 95 machine failed, apparently because Windows 95 can't deal with tens of thousands of files in a directory).

Attempts were made to reduce the complexity of the model. Other PERL scripts were written that would trace out the dependencies of the elements, such as WellLog, that we really cared about. However, the Epicentre data model is not modular, and we discovered that the well log related elements eventually depend upon 1200 of the 1600 classes. We also attempted to package the class files into a JAR (Java Archive) file. However, the jar tool ran out of memory during this process. Also, it should be noted that simple things, such as doing an "ls" of a directory, become hard when the directory contains tens of thousands of files. Almost all operations on the files had to be handled using PERL scripts.

3.0 Closing the Loop

While the size of the data model made many tasks awkward, various small PERL scripts automated many of those tasks. The major difficulty, and the real point of this paper, was something much more fundamental.

CORBA programming proceeds by first defining the interfaces using IDL. The interface is then compiled to client-side stubs and server-side skeletons that are expressed in a particular programming language. The server-side functionality must then be implemented by filling out the skeleton of the server-side code. In our case, the server-side skeletons needed to be fleshed out by adding the SQL calls to communicate with a data repository to actually fetch and store data when calls such as `getWellLog()` or `putWellLog()` were made.

As part of the Epicentre package, POSC provides schemas for common relational databases, such as Oracle and Sybase, which can be used to store the data. Rather than generate our own schemas for an OO database, we decided to use the predefined schemas.

In contrast to the logical model defined in Express (which says what to do), the schemas mentioned specify an implementation (i.e., they say how to do something). Consequently, they embody particular choices that were made about how to implement the logical model, and those choices are invisible from the viewpoint of the logical model. Some of the choices might be relatively small. For example, the names for data elements might be abbreviated or case-converted to meet the limitations of a particular database system. Other choices go far beyond mere lexical matters. As an example, Express allows many-to-many relationships between data elements. When implementing those in a relational database, intersection tables must be created. The mapping from many-to-many constraints in Express and the implementation of those constraints as intersection tables and foreign key constraints in a relational database management system (RDBMS), is complex.

If the Epicentre data model were small, a programmer could tease out that information and write the server-side implementation by hand. However, the size of the Epicentre data model makes such an approach impractical. Given the large number of classes to implement, the only feasible way to implement them is through the use of code-generating tools. If the mapping from the Express data model to the actual tables in a database were simply a matter of lexical conventions, such as the way names are abbreviated, a small PERL script could do the job. However, that is not the case. To be able to write the server-side code, we need to have complete knowledge of the structure and naming conventions in the database. To be able to automatically generate that code, we need a machine-readable description of the implementation choices that were made, and we need a sophisticated program that can read the description of those decisions and then "know" how to generate the SQL calls to implement the functionality of the calls like `getWellLog()` and `putWellLog()`. This is not easy code to write, nor is it easy to specify all the design choices in a machine-readable fashion.

Figure 1 illustrates this problem. The Epicentre Express file is shown at the top. Going along the right-hand side of the figure, we can follow the process of converting that description into the IDL files, and then into the Java files. Looking to the left we see that the various database schemas were generated using a POSC-supplied projection tool. The gray box represents code that needs to be generated for the Java files to be able to talk with the database. This represents a break in the loop that should, ideally, exist. The gray numbered circles identify other potential loops, each with a break in it.

Loop 2 comes about by proceeding around the loop in the other direction. (This is the implementation direction that is currently being pursued on the Oil and Gas project.) Commercial tools have started to appear that will take an existing database and automatically generate Java classes and Java DataBase Connectivity (JDBC) [5] calls for storing and retrieving information from the database. If we still want to use IDL, there are also tools that can take Java classes and derive IDL interface definitions from them. However, this IDL will bear very little relationship to the IDL that came from the Epicentre model. It will reflect all the implementation choices that were made for the database, and consequently it will differ depending on whether the target database was Oracle, Sybase, or something else. This gap in the loop is marked with the shaded circle #2.

A third loop is possible. Instead of trying to talk with an existing database, we could simply use the IDL definitions and resulting Java code to generate the schema for our own object-oriented database. This seems quite feasible but does not meet the basic design requirement of being able to apply the same logical model to any number of physical data stores. This approach would require all the partners in the virtual organization to run the same database and same database schema. If we can make everyone use a particular database, there is no need to try and provide an abstract model for that database.

The major lesson of this paper is that Express models are, by design, not a complete specification of an implementation. If one wants to automatically generate code to deal with implementations of an Express model, one needs two things. First, one must have access to a machine-readable specification of all the additional choices that were made in implementing the model. Second, one must be able to write the program that will take that description of design choices and be able to use it when generating the code that closes the loop.

!

!

References

1. J. M. Kindel, "Advanced Reservoir Management Project," Los Alamos National Laboratory, <http://ees-www.lanl.gov/EES5/arm/index.html>
2. R. B. Webster, "GWIS: Global Weapons Information System"; Los Alamos National Laboratory; 1995; <http://www.acl.lanl.gov/GWIS/>
3. Jon Siegel; CORBA Fundamentals and Programming; John Wiley and Sons; New York; 1996. (Also see <http://www.omg.org> for more general information on the CORBA specifications and the Object Management Group, the consortium that produces them.)
4. Epicentre version 2.2", Petrotechnical Open Software Corporation, 1997. (see http://www.posc.org/Epicentre.2_2/SpecViewer.html)
5. "JDBC 2.0 API"; Sun Microsystems, Inc.; 1998
(<ftp://ftp.javasoft.com/pub/jdbc/jdbc20-spec.pdf>)

Appendix A: PERL Script for Cleaning Up Output of Exp2idl

The following PERL script was used to perform the bulk of the task of repairing the output of the exp2idl compiler. Subsequent versions of that compiler may have addressed these problems, so don't assume it is necessary to actually use this. Please note that there are a couple of limitations of this script that preclude it from being a complete, general, cleanup script for the output of the exp2idl compiler. First, there is at least one error in translating the Epicentre model that is explicitly tested for in this script. Obviously that pattern would not be matched if the Express model came from a different source. Second, after running the script, there were about 6 errors still remaining in the IDL, those problems were fixed by hand since I was anxious to compile things. (Recall that this script was developed very early in the effort, and given the lessons learned during this study, it did not seem crucial to go back and add the last error cases to this script.)

```
#!/perl -w
#
# cleanup_idl.pl - Massage the IDL files from exp2idl
#
# usage: perl cleanup_idl.pl dir1 dir2
#
# Takes all the IDL files in dir1 and cleans them up, putting the
# new version in dir2. The cleanups applied are:
# 1) Fix any occurrences of the "attribute sequence" bug
# 2) Remove any SOM-specific code
# 3) Look for known bugs, such as the idl_long class, and fix them.
# DIR2 is created if it does not exist. If it does exist, files in
# it will be overwritten.

$fc = @ARGV;
if ($fc != 2) {
    print STDERR "usage: perl cleanup_idl.pl dir1 dir2\n";
    die;
}
($dir1, $dir2) = @ARGV;

# Make sure dir1 exists, get list of all the generated idl files.
opendir DIR1, $dir1 or die "Can't use directory $dir1: $!\n";
@files = grep /\.*\.idl/, readdir DIR1;
closedir DIR1;

# Make sure dir2 exists.
unless (-e $dir2) {
    mkdir $dir2, 777 or die "Couldn't make directory $dir2: $!\n";
}

# Start processing all the IDL files

foreach $filename (@files) {
    $infile = "$dir1/$filename";
    $outfile = "$dir2/$filename";
    # Get the input and output files opened.
    open INFILE, $infile or die "Can't open $infile: $!\n";
    open OUTFILE, ">$outfile" or die "Can't open $outfile: $!\n";

    # Do the work
    &mung;

    # Close the files.
    close INFILE;
    close OUTFILE;
}
```

```

exit 0;

# Here's the call that actually mungs the individual files and fixes them.
sub mung {

    # Declare a hash table to hold the buggy lines and the fixed ones.
    local (%theHash);

    # Patterns for the "attribute sequence" bug.
    $unbound_seq = 'sequence<\s*(\w*)\s*>';
    $bound_seq   = 'sequence<\s*(\w*)\s*,\s*(\d*)\s*>';
    $bug_bound_seq = 'sequence<\s*(\w*)\s+(\d*)\s*>';
    $seq_seq      = 'sequence<.*sequence<';

    # Scan file to build list of sequences that will need
    # to be typedef'ed.
    while (<INFILE>) {
        next unless (/attribute\s*sequence/);

        if (/ $seq_seq/) {
            ($k, $t, $u) = &ss;
            $theHash{$k} = $t;
            next;
        }
        if (/ $unbound_seq/) {
            $t = fixup($1);
            $theHash{$t} = "sq_$t";
            next;
        }
        if (/ $bound_seq/) {
            $t = fixup($1);
            $theHash{"$t,$2"} = "sq$2_$t";
            next;
        }
        if (/ $bug_bound_seq/) {
            print STDERR "$infile has buggy bound sequence:\n";
            print STDERR "$_";
            $t = fixup($1);
            $theHash{"$t,$2"} = "sq$2_$t";
            next;
        }
    }

    # Rewind the inputfile
    seek INFILE, 0, 0;

    # Here is where we do the real work. Read lines from the file,
    # possibly rewrite them, and print to the output file. The sorts
    # of rewrites we do:
    # 1) Insert typedefs before the interface definition
    # 2) change "attribute sequence" lines to use typedefs.
    # 3) Delete SOM implementation sections.
    # 4)

    $inOmit = 0;
    while (<INFILE>) {
        $line = $_;
        if (/#ifdef __SOMIDL__/) {
            $inOmit = 1;
            next;
        }
    }
}

```



```

    if ($inOmit > 0) {
        if (/#endif/) {
            $inOmit--;
        }
        if (/#ifdef/) {
            $inOmit++;
        }
    }
    next;
}

# Emit any new typedefs between the external interface declarations
# and the main interface definition
if (/interface .* :/) {
    foreach $k (keys %theHash) {
        print OUTFILE "typedef sequence< $k > $theHash{$k};\n";
    }
    print OUTFILE $line;
    next;
}

# Rewrite any buggy "attribute sequence..." lines to use the typedefs
if (/attribute\s+$unbound_seq\s+(\w*);/) {
    $t = fixup($1);
    print OUTFILE "\tattribute $theHash{$t} $2;\n";
    next;
}

if (/attribute\s+$bound_seq\s+(\w*);/) {
    $t = fixup($1);
    $key = "$t,$2";
    print OUTFILE "\tattribute $theHash{$key} $3;\n";
    next;
}

if (/attribute\s+$bug_bound_seq\s+(\w*);/) {
    $t = fixup($1);
    $key = "$t,$2";
    print OUTFILE "\tattribute $theHash{$key} $3;\n";
    next;
}

if (/seq_seq/) {
    # print "Fix $line";
    # print OUTFILE $line;
    ($key, $tname, $uname) = &ss;
    print OUTFILE "attribute $tname $uname\n";
    next;
}

# Look for known cases where an IDL primitive type should be
# used.
if (/idl_string\s*<(\d*)>\s*(\w*);/) {
    print OUTFILE "\tattribute string<$1> $2;";
    next;
}

print OUTFILE $line;
}

# Deal with nested sequence lines, parsing them into
# the hash key, the type name, and the variable name.
#
# usage: ($key, $tname, $uname) = &ss;
sub ss {
    my($line) = $_;
    my($key, $uname, $tname);

```



```

unless ($line =~ /attribute\s*sequence/) {
    die "bad input to seqseq()\n"
}

$line =~ s/idl_sequence/sequence/g;

$line =~ /attribute\s*(sequence<\s+(.*)\s+>\s+(\w*);(.*)$/;
$key = $2;
$name = $3;
$name = construct($1);
return ($key, $name, $name);
}

# Build up a reasonable type name for the typedef statement
# we will emit for a sequence<sequence<...>>.
sub construct {

    my ($chunk) = @_;

    if ($chunk =~ /sequence<\s*(.*)\s*\s+(\d*)\s*>\s*$/ ) {
        $tail = construct($1);
        return "sq$2$tail";
    }
    if ($chunk =~ /sequence<\s*(.*)\s*>\s*$/ ) {
        $tail = construct($1);
        return "sq$tail";
    }
    if ($chunk =~ /\s*(\w*)\s*/ ) {
        return "_$1";
    }
    print STDERR "Unknown structure of line\n$line\n in file $infile\n";
    return "_$chunk";
}

# Look for known bad types, such as idl_long, and replace them
# with the known correct type, such as long.
#
# NOTE: This routine looks for an error that is known to occur
# with the translation of the Epicenter model to IDL (the
# v_Fluid_phase_interface string).
sub fixup {
    my($t) = @_;

    if ($t =~ /idl_long/) {
        return "long";
    }
    if ($t =~ /v_fluid_phase_interface/) {
        return "v_Fluid_phase_interface";
    }

    return $t;
}

```