The NIST EXPRESS Toolkit – Updating Existing Applications

Don Libes

Abstract

The NIST EXPRESS toolkit is a software library for building EXPRESS-related tools. The toolkit was previously released in 1991, based on ISO TC184/SC4 N14 (familiarly called "EXPRESS N14"). The current release is based on Draft International Standard (DIS) 10303-11 (N151) and while the philosophical underpinnings are similar, much of the interface has changed significantly. This paper describes changes that must be made to existing applications so that they can work with the new toolkit.

This paper should be read by anyone maintaining software based upon the NIST EXPRESS toolkit. This paper will also provide insight to people interested in the internals of EXPRESS implementations and some of the ways they have changed over time due to experience and the different EXPRESS specifications.

Keywords: compiler, EXPRESS; implementation; National PDES Testbed; PDES; STEP

Context

The PDES (Product Data Exchange using STEP) activity is the United States' effort in support of the Standard for the Exchange of Product Model Data (STEP), an emerging international standard for the interchange of product data between various vendors' CAD/CAM systems and other manufacturing-related software [1]. A National PDES Testbed has been established at the National Institute of Standards and Technology to provide testing and validation facilities for the emerging standard. The Testbed is funded by the Computer-aided Acquisition and Logistic Support (CALS) program of the Office of the Secretary of Defense.

As part of the testing effort, NIST is charged with providing a software toolkit for manipulating STEP data. The NIST EXPRESS Toolkit is a part of this. The toolkit is an evolving, research-oriented set of software tools. This document is one of a set of reports (an overview of each appears in [2]) which describe various aspects of the Toolkit.

Introduction

The bad news is: Much has changed. You will not be able to recompile applications without changing them.

The good news is: The system is faster. Much faster. And the library conforms to the EXPRESS DIS, and implements everything needed to do full resolution of all features of EXPRESS.

This document describes how to change your code to work with the new toolkit. Some of the changes are not one-for-one; in this case, do not expect a direct replacement. You may be forced to recode parts of an application.

While the intent of this document is to describe how to convert programs, we occasionally describe other features, especially if they provide additional important benefit in some way.

Draft International Standard

As this report is being written, the EXPRESS specification is now a Draft International Standard [7]. It has changed significantly from earlier drafts. Correspondingly, the toolkit has changed. There is no support for anything not mentioned in the current EXPRESS DIS. In particular, schema files must conform to the DIS. No attempt is made to process schemas based on any other EXPRESS specification.

Object-Oriented Engine

The previous version of the toolkit provided an object-oriented view of the internals. This view is almost entirely gone. In particular, most of the **`OBJ`** functions no longer exist. A few such as **`OBJfree`** exist as empty macros only to ease the pain of converting to the new version.

Due to changes in the implementation, most of the functionality originally provided by the OO engine is relatively straightforward to obtain without it. This section describes how to accomplish this.

External Types

Some types are now stored in the dictionary rather than in the object. In particular, when retrieving an object from a dictionary by name, the type of the object is set in the global **`DICT_type`** as a side-effect. The following code demonstrates how entities and functions in a scope could be located and processed:

```
DictionaryEntry de;
Generic x;

DICTdo_init(scope->symbol_table,&de);
while (0 != (x = DICTdo(&de))) {
      switch(DICT_type) {
      case OBJ_ENTITY: /* do something specific to entities */
            ENTITYdo_something((Entity)x);
            break;
      case OBJ_FUNCTION: /* do something specific to functions */
            FUNCTIONdo_something((Function)x);
            break;
      }
}
```

`DICTdo_type_init`, a variant of **`DICTdo_init`**, provides a shorthand when looking for one specific type in a dictionary. For example, the following code prints all the names of the schemas:

```
DictionaryEntry de;
Schema s;

DICTdo_type_init(scope->symbol_table,&de,OBJ_SCHEMA);
while (0 != (s = (Schema)DICTdo(&de)))
      printf("%s\n",SCHEMAget_name(s));
```

Internal Types

Once an object type is known, further typing information (i.e., subclasses) can be determined by referencing the `type` element of the current structure. In most cases, the type value will act as a tag for an associated union, typically called "`u`". This union contains the type-dependent data for each structure.

For example, given a statement, it could be processed as follows:

```
switch (statement->type) {
case STMT_ALIAS:
      ALIASdo_something(statement->u.alias);
      break;
case STMT_LOOP:
      LOOPdo_something(statement->u.loop);
      break;
case . . .
}
```

Temporary Conversion Aids

As mentioned earlier, some of the old OBJ functions still exist simply to ease the pain of converting. Here are some notes on each:

`OBJequal`	Does a shallow comparison. This is usually desirable now that objects no longer point to structures containing object information, but rather point to the structures of the true data.
`OBJfree`	Does nothing. Since most toolkit functions no longer create copies, this is usually desirable. In the special case that memory should really be freed, you must use the object-specific free function.
`OBJref`	Does nothing. Reference counting is no longer appropriate on a system-wide basis.
`OBJget_data`	Returns original object unchanged. This function originally stripped the `Object` wrapper off of an object. Since `struct`s are no longer wrapped in `Object`s, there is no need to do anything.

All other functions should be replaced with their object-specific counterparts. For example, instead of

```
OBJcreate(Class_Linked_List,&errc);
```

use

```
LISTcreate()
```

Symbol

In the previous release, some objects were derived from **`Symbol`**. The most obvious use of **`Symbol`** was to provide objects with a name. This could be retrieved with **`SYMBOLget_name`**. **`Symbol`**s have died along with the OO engine. Thus, there is no way to apply **`SYMBOLget_name`** to arbitrary objects.

To retrieve the name of an object, call the **`XXXget_name`** where **`XXX`** is the type name. For example, the name of a **`Schema`** is:

```
SCHEMAget_name(schema)
```

Scopes

A variety of changes have been made to the scope mechanisms.

The scope resolution functions are completely different in order to match the change from single-pass resolution to multiple-pass resolution. The description of these is beyond the scope of this paper. Interested readers are referred to the Programmer's Reference [5].

SCOPEget_XXX

The following functions all retrieved a subset of information from a scope:

```
SCOPEget_types
SCOPEget_variables
SCOPEget_algorithms
SCOPEget_constants
```

In virtually all cases, these functions should be rewritten as an explicit scope traversal. Most existing usage of such functions are immediately followed by a list traversal anyway. Combining these removes the overhead of creating, traversing, and destroying a list.

For example, a typical use of SCOPEget_types might have been written this way:

```
Linked_List list = SCOPEget_types(scope);
LISTdo(list,s,Symbol)
     printf("%s\n", TYPEget_name(s));
LISTod
OBJfree(list);
```

This should be rewritten as:

```
DictionaryEntry de;
Type t;

DICTdo_type_init(scope->symbol_table,&de,OBJ_TYPE);
while (0 != (t = (Type)DICTdo(&de)))
      printf("%s\n",TYPEget_name(t));
```

The scope lookup functions have also been radically revised. Unfortunately, there is no simple one-to-one substitution that can be suggested for these. Here are the new functions:

`VARfind`	Find an attribute (or variable or constant) reference in a scope.
`SCOPEfind`	Find an object that is not inherited by the super/subtype hierarchy.
`SCOPEfind_for_rename`	Find the true object referenced from another schema.
`EXPRESSfind_schema`	Find the named schema. This may cause new schema files to be read and parsed.

For efficiency, some of the scope functions also take an argument defining what object types should be skipped while searching. This argument is a bit representation of the object types. It is not the same thing as the types used by the dictionary (see External Types on page 2) which are enum-like in nature[1].

For example, this provides a way of determining the type, given the (legal) attribute declaration:

```
A1: A1;
```

It is not sufficient to merely start searching at a superscope since types can be defined within the current scopes. The important thing is to ignore attributes. An example piece of code might use **`OBJ_TYPE_BITS|OBJ_ENTITY_BITS`** to find either a type or an entity. To find anything, use **`TYPE_ANYTHING_BITS`**. A complete discussion of this can be found in the reference manual [5].

Implicit loop controls and **`ALIAS`** are handled by associating with them a scope of one element.

The remaining discussion about scopes is provided for completeness but is probably not of interest to most readers.

The scope structure used to look like this:

```
struct Scope {
      Linked_List     parents;
      Dictionary      symbol_table;
      Dictionary      references;
      Linked_List     use;
      int             last_search;
```

1. The OBJ_XXX types are macro definitions rather than enumeration values. They are used in enum-like ways, however defining these as a single enumeration type would prevent the addition of other types by application programmers.

```
        Boolean         resolved;
    };
```

Now it looks like this:

```
    struct Scope {
            Symbol symbol;
            char type;
            int search_id;
            Dictionary symbol_table;
            struct Scope *superscope;
            union {
                    struct Procedure *proc;
                    struct Function *func;
                    struct Rule *rule;
                    struct Entity *entity;
                    struct Schema *schema;
                    struct Express *express;
                    struct Increment *incr;
            } u;
    };
```

The **`type`** and **`u`** elements are described elsewhere (see Internal Types on page 3). The symbol originally inherited by the OO system is now declared explicitly. **`search_id`** is just **`last_-search`** renamed. **`superscope`** contains the information that was otherwise found in parents. **`resolved`** is now contained within **`symbol`**. Use and Reference are described elsewhere (see Use/Reference Changes on page 6).

Note that two nominal scopes do not appear in **`u`**: **`query`** and (enumeration) **`type`**. Both of these need to be members of (or put another way "inherit behavior from") other structures. In an object-oriented system, this would be called multiple inheritance. In our non-OO system, we simply pick one structure and simulate the rest with some extra code.

Use/Reference Changes

In the previous release, Used and Referenced objects were actually copied from one schema to another. Nonetheless, there were Use/Reference elements in the **`Scope`** structure which could be used to determine whether objects were non-local.

The new release keeps a clean separation between local and non-local objects. Separate functions are available to find objects in the local scope versus objects in the remote scope. For instance **`SCOPEget_entities_supertype_order`** now no longer returns Used entities. Instead, use **`SCHEMAget_entities_use`** to get Used entities and **`SCHEMAget_entities_ref`** to get Referenced entities.

No functions exist for certain obvious queries (such as "what schema is the owner of this object"). If more than a few such queries will be made by an application, it is cheaper to traverse the Use/Reference data structures once rather than having the toolkit do it multiple times, once for each request. See the reference manual for more information on this.

Expressions

Except for expression types (see Types on page 7), most of the functions having to do with expressions from the previous release should work in the new release. However, the underlying implementation is completely different. The following explanation is only provided in case the provided functions are insufficient. The reader is referred to the reference manual for more complete information.

In the previous release, **`Expression`**s relied heavily on the OO inheritance. In the current system, **`Expression`**s are quite explicit. The new definition of an **`Expression`** is:

```
struct Expression {
        Symbol symbol;
        Type type;
        Type return_type;
        struct Op_Subexpression e;
        union expr_union u;
};
```

`type` is the syntactic type of the expression (expression-with-operands, function-call, etc.) while **`return_type`** is the semantic type of the value returned (integer, real, set of real, etc.). One of **`e`** (for operands) or **`u`** (for anything else) is used to hold type[2]-specific data depending on the value of **`type`** and **`return_type`**.

Algorithms

Some of the algorithm access functions have been changed to be specific to the type of algorithm. For example, **`ALGget_parameters`** should be changed to **`FUNCget_parameters`**, **`RULEget_-parameters`**, or **`PROCget_parameters`**.

Types

Some of the functions having to do with **`Type`**s from the previous release will work in the new release. However, the underlying implementation is completely different and some functions may need some conversion. The following sections describe the impact in more detail. Unless otherwise mentioned, functions from the previous release should still work.

Object Functions

Much of the typing was implicitly handled by the OO engine. This typing is now explicit. In particular in the previous implementation **`OBJget_class`** was used to get the "class" of a type. There is now no global notion of class, although the concept still exists for types. For this reason, the old **`Class`** references have been fixed to continue to work but only on **`Type`**s. Do not use functions such as **`OBJget_class`** on anything but **`Type`**s.

For consistency with all other functions, a type's "class" is now called a type's "type".

2. syntactic

Enumerations

Each enumeration type and item is represented by an **`Expression`** which contains a scope (see Scopes on page 4). This is done in order to support different enumerations with common item names. Thus, the function **`ENUM_TYPEget_items`** now returns a dictionary instead of a list. Each element is an expression of type **`enumeration`** instead of a symbol.

Enumeration items are not only entered into the scope of the enumeration type definition, but they are also entered into the immediately outer scope since that is where their primary visibility is required. In this higher scope, it is possible to find *ambiguous enumerations*. These are overloaded enumeration items. Finding one normally indicates that the enumeration item name needs to be qualified with the enumeration type name.

Libmisc and Other Directory Rearrangements

The **`libmisc`** library has been bundled in with the EXPRESS library, so this reference can be removed from Makefiles. While **`libmisc`** was originally seen as a separate toolkit, it was heavily customized only for use with the EXPRESS toolkit in this version. Because of this, there seemed no point in keeping the include files separate. Thus, the EXPRESS and **`libmisc include`** files now live in the EXPRESS source directory.

A description of specific changes follows:

OBJ Functions

The object engine is gone [see Object-Oriented Engine on page 2]. Some new object functions exist, but these provide new functionality and are probably not of help in converting old code.

DICT Functions

The dictionary is slightly different. In particular, the dictionary not only explicitly knows the name of an object, but it knows the type and the line number and file in which the object was originally found. This is necessary to make up for the missing OO engine as well as to produce error messages that understand multifile EXPRESS models.

STRING Functions

The **`String`** abstraction support routines have been removed. The abstraction allowed different underlying representations for strings, but was incomplete to the point that users had to assume that the usual C representation was used. It was pointless to complete the abstraction since the Standard C library is now very rich in string support. Use the Standard C library string functions.

The typedef **`String`** and the function **`STRINGequal`** continue to exist but are deprecated. All other references to the string abstraction no longer exist.

Main

Default

The previous version of the toolkit provided a default main procedure that had a strong view of how applications should look and feel. For instance, applications were expected to produce an output file. For this reason, applications were expected to name their entry point as:

```
void print_file(Schema,FILE *);
```

The new toolkit provides a main, but it is significantly more configurable. A number of options may be controlled by setting variables inside the user-defined function **`EXPRESSinit_init`** which is called immediately at program start-up. To simulate the earlier interface, the variable **`EXPRESSbackend`** is set to the application entry. This function must return **`void`** and take a single argument of **`Express`**. For example:

```
#include “express.h”
#include “resolve.h”

void print_file(Express);

void
EXPRESSinit_init() {
        EXPRESSbackend = print_file;
}
```

`EXPRESSinit_init` must be defined by the user as a true function which matches the example above.

The new **`main`** is significantly more flexible and can likely replace **`main`** routines that had to be handwritten in prior releases. You can find other flexibility provided by the new **`main`** in the reference manual.

Dynamic Loading

The **`express_dynamic`** package is gone. No one (to my knowledge) used this. The functionality was not worth the increase in code space and the result was highly nonportable.

The **`express_static`** package is all that remains. This was modified so that it is invisible to the application. All references to either of these options can be deleted.

Passes

All the references to **`pass1`** and **`pass2`** are gone. The following have direct substitutions:

Old	New
`void EXPRESSpass_2(Express)`	`void EXPRESSresolve(Express)`
	`Express EXPRESScreate();`
`Express EXPRESSpass_1(FILE *)`	`void EXPRESSparse(Express,FILE *, char *filename)`

Either of the 2nd or 3rd arguments to **`EXPRESSparse`** may be null, but at least one must be non-null. The filename is preferred since this can be used to generate more descriptive messages.

`EXPRESSparse` must always be preceded by **`EXPRESScreate`**, although not necessarily immediately. In particular, **`EXPRESSparse`** may be called multiple times on the result returned by **`EXPRESScreate`**. (This is useful when parsing new schemas that are being added to schemas that have already been parsed.)

Warning Options

The interface to the warning system has been completely redesigned. Code such as:

```
warnings[ecnt].name = STRINGcopy(“subtypes”);
warnings[ecnt++].error = ERROR_unknown_subtype;
```

should be rewritten as:

```
ERRORcreate_option(“subtypes”,ERROR_unknown_subtype);
```

Once an option is created, no parsing is necessary to determine which option to set or unset. Rather, the function **`ERRORset_option`** should be called with the option string as:

```
ERRORset_option(optarg,1);
```

The 2nd argument is a boolean describing whether the option should be set or unset. It is especially convenient to use one flag character to set an option and another to unset it. Then one can say:

```
case ‘s’: /* set */
case ‘u’: /* unset */
     ERRORset_option(optarg,optchar == ‘s’);
```

See the reference manual for more information.

Error Functions

Sorting

By default, diagnostics are printed immediately rather than buffering them up and sorting them by line number. The underlying function to toggle this is defined as follows:

```
ERRORbuffer_messages(boolean);
```

While the buffering code has been speeded up (it used to call two extra processes, now it doesn’t call any), there is little point to sorting by line numbers. The order in which diagnostics are presented to the user are the order in which problems should be resolved. I.e., a missing schema will be now detected immediately and will cause many spurious errors rather than vice versa.

ERRORreport_with_line

`ERRORreport_with_line` still exists, however users are encouraged to use **`ERRORreport_with_symbol`**. Symbols now contains filenames describing the file in which the symbol was encountered. This makes for more helpful diagnostics. If **`ERRORreport_with_line`** is used, the filename is heuristically derived.

All diagnostics are formatted slightly differently than before. This change was made specifically to support the Emacs compile hook [11] which manipulates the source and diagnostics in two separate windows. Each time the **`next-error`** function is invoked (typically by the two key sequence: control-X backquote), both windows are scrolled so that the next diagnostic and the corresponding source line are displayed.

How to Obtain the Toolkit

The toolkit and its documentation may be obtained in a variety of ways. The simplest way is through anonymous ftp via the Internet. In this case, the source is /pub/step/npttools/exptk.tar. Complete documentation on obtaining the toolkit and its documentation is /pub/step/ntpdocs/exptk-obtaining-installing.ps.Z [17].

Alternately, it possible to receive the toolkit by email. To do this, send the following mail to ntpserver@cme.nist.gov:

```
send step/npttools/exptk.tar.Z
send step/nptdocs/exptk-obtaining-installing.ps.Z
```

If you do not understand these instructions or for any other reason cannot successfully use ftp or email, contact:

FASD – National PDES Testbed
National Institute of Technology and Standards
Bldg 220, Rm A-127
Gaithersburg, MD 20899

npt-info@cme.nist.gov
1-301-975-3508

Questions, Problems, and Support

While we are willing to listen to problems, requests for extension, etc., we cannot guarantee any kind of response. Since the system is distributed in source form, you are encouraged to experiment with the system, especially if you have problems with it. While it is often quicker for you to have us diagnose your problems, it is quicker for us to have you diagnose your own problems. This software is a research prototype, intended to spur development of commercial products.

Nonetheless, if you do have questions and/or problems, you may send e-mail to hotline@cme.nist.gov. Please include schemas, version numbers, platform descriptions, and any other information that could be relevant.

References

[1] Mason, H., ed., "Industrial Automation Systems – Product Data Representation and Exchange – Part 1: Overview and Fundamental Principles", Version 9, ISO TC184/SC4/WG PMAG Document N50, December 1991.

[2] Libes, Don, "The NIST EXPRESS Toolkit – Introduction and Overview", National Institute of Standards and Technology, Gaithersburg, MD, to appear.

[3] Clark, Steve N., "An Introduction to The NIST PDES Toolkit", NISTIR 4336, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

[4] Schenck, D., ed., "Exchange of Product Model Data - Part11: The EXPRESS Language", ISO TC184/SC4 Document N496, July 1990.

[5] Libes, Don, "The NIST EXPRESS Toolkit – Programmer's Reference", National Institute of Standards and Technology, Gaithersburg, MD, to appear.

[6] Clark, Steve N., "QDES User's Guide", NISTIR 4361, National Institute of Standards and Technology, Gaithersburg, MD, June 1990.

[7] Morris, K.C., "Translating EXPRESS to SQL: A User's Guide", NISTIR 4341, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

[8] Spiby, P., ed., "ISO 10303 Industrial Automation Systems – Product Data Representation and Exchange – Part 11: Description Methods: The EXPRESS Language Reference Manual", ISO DIS 10303-11:1992(E), July 15, 1992.

[9] Morris, K.C., Sauder, David, and Ressler, Sandy, "Validation Testing System: Reusable Software Component Design", NISTIR 1992-X, National Institute of Standards and Technology, Gaithersburg, MD, September 1992.

[10] Johnson, S.C., "Yacc: Yet Another Compiler compiler", *UNIX Programmer's Manual*, Seventh Edition, Bell Laboratories, Murray Hill, NJ, 1978.

[11] Schreiner, Axel T. and Friedman, Jr., H. George, *Introduction to Compiler Construction with UNIX*, New York, NY, Prentice Hall, 1985.

[12] Stallman, Richard M., et al, *GNU's Bulletin*, Free Software Foundation, Inc., Cambridge, MA, June 1992.

[13] Lesk, M.E. and Schmidt, E., Lex: A Lexical Analyzer Generator, *UNIX Programmer's Manual*, Seventh Edition, Bell Laboratories, Murray Hill, NJ, 1978.

[14] McLay, Michael J. and Morris, K.C., "The NIST STEP Class Library", *C++ at Work-'90 Conference Proceedings*, (reprinted as NISTIR 4411,) September 1990.

[15] Morris, K.C., "Architecture for the Validation Testing System Software", NISTIR 4742, National Institute of Standards and Technology, Gaithersburg, MD, January 1992

[16] Libes, Don, and Clark, Steve N., "The NIST EXPRESS Toolkit – Lessons Learned", *Proceedings of the 1992 EXPRESS Users' Group (EUG '92) Conference*, Dallas, Texas, October 17-18, 1992.

[17] Libes, Don, "The NIST EXPRESS Toolkit – Obtaining and Installing", National Institute of Standards and Technology, Gaithersburg, MD, to appear.