The NIST EXPRESS Toolkit – Lessons Learned

Don Libes <libes@cme.nist.gov>
Steve Clark <clark@cme.nist.gov>
National Institute of Standards and Technology
Gaithersburg, MD

Abstract

In 1990, NIST released a software toolkit for building EXPRESS-related tools. The software was based on early drafts of EXPRESS and included a parser, a resolver, and a framework for producing schema-independent tools.

During the past year, we have redesigned the toolkit with respect to user experience and the EXPRESS Draft International Standard. Our work will be of interest to implementors of EXPRESS tools and toolkits. This paper describes our work and the current state of the NIST EXPRESS toolkit.

As well as describing our efforts in improving the toolkit performance, we will comment on:

- the "hard" parts of implementing the EXPRESS language,
- things we did wrong,
- several implementation-dependent extensions to EXPRESS,
- sophisticated error and warning detection.

We will also revisit our original ideas underlying the toolkit concept. We will discuss whether, in light of our experiences, the concepts still make sense, can be amended, or should be entirely scrapped.

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

Reprint of *Proceedings of the 1992 EXPRESS Users' Group (EUG '92) Conference*, Dallas, TX, October 17-18, 1992.

Introduction

In 1990, NIST released a software toolkit [1] for building EXPRESS-related tools. The software was based on an early draft of EXPRESS [2] and included an EXPRESS parser, a resolver, and a framework for producing schema-independent tools. Several schema-independent tools were also included such as a graphical editor [3] and an EXPRESS-to-SQL translator [4].

All of this software was placed into the public domain and has seen wide use, functioning both as products and as examples of how to build EXPRESS products (and in some cases, how *not* to build them).

This paper describes the present state of the NIST EXPRESS toolkit. We have revised the toolkit with respect to user experience and the EXPRESS Draft International Standard (DIS) [5]. Our work will be of interest to implementors of EXPRESS tools and toolkits. The toolkit now runs in less than 1% of the time than the previous release. Space consumption has been reduced by 25%. Unlike the earlier release, the entire EXPRESS language has now been implemented.

Feedback from both inside and outside of NIST as new EXPRESS language revisions provided the impetus for revisiting the original NIST tools. In particular, the NIST toolkit:

- was outdated by recent EXPRESS revisions,
- skipped a number of the more esoteric EXPRESS constructs (some deemed "esoteric" only by the fact that they were skipped, so that users simply did their best to avoid them),
- implemented several concepts incorrectly, and
- implemented several concepts (originally specified ambiguously) differently than newer restatements.

An obvious goal, then, was to update, complete, and in some cases fix the implementation. An additional interest was to:

- increase the efficiency of the software. Users complained that applications were too slow. For example, translation alone of a typical 500Kb schema took about 10 minutes (on a Sun SPARCstation 2)[1]. We suspected it was possible to improve on this, but if not, we wanted to know why.

But for these points, we felt that the NIST software functioned well and had met its goals. Updating, and in some cases, fixing the software did not seem to present a difficult task; however, if the efficiency of the software could not be improved, it called into question the original concept on which the software was based.

As an example, other researchers used the NIST software as a translator, producing source code that could be recompiled to build schema-dependent programs [6]. This gave up the advantages of the schema-independent paradigm, but was the only reasonable solution for researchers who wanted programs that ran "reasonably" fast.

1. Trade names and company products are mentioned in the text in order to adequately specify experimental procedures and equipment used. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

Terms and concepts

For simplicity, the current EXPRESS specification will be referred to as *the specification*. We will occasionally refer to explicit drafts by their “N” number.

Frequent reference will be made to the *new* implementation of the EXPRESS translator (based on the DIS, also known as N151). This implementation has not yet been publicly released. The *previous* implementation is the most recent public release to date (based on N14). The terms *translator* and *toolkit* are also used, in which case, the implementation is not specified (although it may be implied by context).

The terms *resolving* and *resolution* refer to the connecting and connection between an EXPRESS schema token and its meaning relative to other tokens. For example, during parsing, an Entity reference is just a string representing the name of an entity. After resolution, the Entity reference provides access to all information about the entity, such as attributes and their types. In the implementation, this access is provided by pointers to the true objects.

The abbreviation *OO* is short for Object-Oriented. We assume that the reader understands the nature of OO programming and its advantages and disadvantages.

Changes Motivated by the DIS

There were few outright changes between N14 and the DIS. Some explanations, however, were reworded in ways that necessitated significant changes in our code. We will describe one of the more significant examples – scoping.

Scoping

The concept of a scope is implemented as a dictionary (via open hashing). Objects can be entered into a scope (specifically, the dictionary representing the scope) by name, and later retrieved the same way. Such objects can include scopes of their own. Taken together, these two techniques can represent the multiple hierarchies of scopes.

Our scopes are augmented with additional information such as a handle to the lexically enclosing scope, and in the case of entities, schema interface information (see Use and Reference on page 6).

Scoping seems basic to EXPRESS schemas, yet each publicly distributed draft of EXPRESS made significant changes to the concept. Enumerations stand out as particularly unlike the original scoping intent, and much of the wording in that section of the specification describes the exceptions to the “usual” scoping rules demanded by Enumerations.

The reader is presumed to understand the fine details of Enumeration types. In order to implement this, we chose to skip the implementation advice offered by the specification (to paraphrase, “have scoping work differently when handling an Enumeration”). Instead, the toolkit enters Enumeration items into both the Enumeration type scope and the next outer scope. Collisions are handled differently depending upon whether zero, one, or two Enumerations are involved, and if the collision occurs in an enumeration scope or not. An interesting case is when collisions occur in a non-Enumeration-type scope. If two Enumeration items collide, the previous Enumeration item is deleted and an “ambiguous Enumeration” placeholder is entered into the scope. If this placeholder is ever found during resolution processing (i.e., during later expression analysis), an error message is issued.

If the collision involves a single Enumeration (or the aforementioned ambiguous Enumeration), a diagnostic is issued as per the General Rules of Visibility in clause 10.2.1, exception e: “An identifier is not visible in the scope outside the local scope unless the identifier is an enumeration item identifier, in which case it is visible in the next outer scope”. This could happen, for example, if an attribute is declared with the same name as an entity item, thereby shadowing it. The code, of course, must account for this occurring in either order.

Our solution sacrifices extra storage space (for duplicate entries in two scopes), but greatly reduces processing time since scoping exceptions are only handled during dictionary collisions, rather than having a complex scope lookup algorithm.

Changes Motivated by Efficiency

Several changes were motivated by efficiency. A description of some of these follows.

Internal Implementation was Object-Oriented

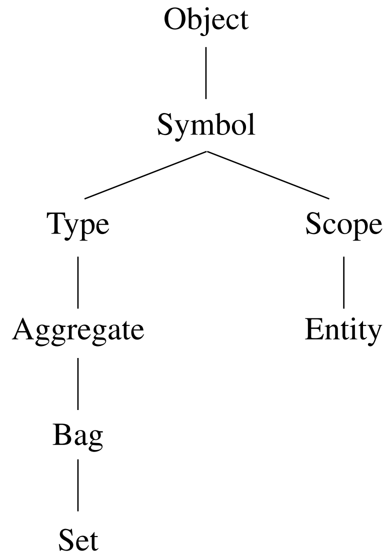
Our previous implementation used an OO engine. This simplified aspects of the coding in several ways. For example, objects were self-describing and it was not necessary for the programmer to check explicit type information as to what, say, an element in a scope might be or how to extract its name.

Our OO system provided single-inheritance. Using this, EXPRESS concepts were modelled directly with internal objects in a very straightforward mapping. As an example, a typical class, such as Entity, was represented as a class that included the following derivation:

```
Object
  |
Symbol
  |
Scope
  |
Entity
```

In other words, Entity was a subclass of Scope, which in turn was a subclass of Symbol. Object was a superclass of Symbol. All objects in the system were derived from Object either explicitly or implicitly.

A Set type had the lengthiest derivation (shown below sharing classes with the Entity derivation).



All of these classes are intuitive except Symbol, which was used to define certain basic attributes such as a name. These basic attributes were then inherited by all subtypes of Symbol. In contrast, an Expression (yet another class derived from Object) would not have a name.

This framework provided a pleasant structure within which the toolkit was written. Unfortunately, the OO implementation - constructed specifically for the toolkit - was itself a major source of inefficiency. Class typing was performed during object manipulation. This provided great flexibility - for example, the parser generated many Symbols (as above, named objects but with unknown meaning). These Symbols would later be specialized when their semantics were discovered during resolution. All of this was neatly accomplished by the OO system, although at significant expense.

Our OO implementation was written in C. The OO engine did all of its work at run-time, unlike a traditional C++ implementation which can do a significant amount of class-typing at compile-type. While a C++ system would undoubtedly have performed much better than our hand-built engine, some of our needs (in particular, run-time typing) could not have been met any more efficiently by C++.

The dynamic single-inheritance alone imposed a geometric price in run-time. Even worse, some constructs required inheritance from unrelated parts of the class hierarchy. For example, an Enumeration was both a Type and a Scope. This required yet another run-time type evaluation. Unfortunately, the flexibility of providing multiple inheritance exacts a price from all other objects in the system even if they do not take advantage of it, if only because they must at least check for the possibility on each member access.

Rewriting the system using a compiled OO system (such as C++) could have removed some of the geometric cost for single-inheritance, but would still have left the multiple-inheritance cost. We decided to experiment with the possibility of scrapping the OO nature entirely. We considered the trade-offs including:

- We no longer needed the ability to change underlying implementations easily. Major changes to EXPRESS had stopped. At the same time, we had become experienced enough with the requirements of an EXPRESS implementation that we almost always knew the optimal algorithms and data structures to use for each part of the system.[2]

- Without an OO system, the code and data structures must explicitly support the flexibility. In the new implementation, C unions are heavily used to support the data structures. The resulting code to handle type-dependent decisions is more complex but runs much faster.
- Only logic switches of significant size (or number of them) benefit from hiding the complexity. We see nothing wrong in using explicit 'case' statements to replace some of the logic made implicit by the OO system. Sufficiently large branches were replaced with type-specific arrays, in effect providing tiny OO systems but only where they were clearly needed, such as for the large number of expression operators and built-in functions.

Undeniably, the resulting code is more complex. The implementor must remember to do more things. Logically, however, it is not all that bad. By judicious choices, the hierarchy of classes was logically compressed from six levels down to two levels. Inevitably, two levels is easier to write code for than six.

While the type-specific code is more complex, the overall code size actually shrank – the OO engine itself plus all the per-object access functions were a significant portion of the system.

Use and Reference

When Use and Reference were first introduced, implementation advice (since removed from the EXPRESS specification) suggested schemas or partial object hierarchies be copied. This was exactly the approach taken by our earlier implementation. This was inefficient for several reasons, the least of which was that it took space and time to copy objects. Our single-pass approach to resolution (described later on) made the situation more complex. For instance, schemas had to be suspended during resolution when unresolved objects from other schemas were encountered. Partial schema resolution was tricky yet necessary in order to support schemas that referenced parts of one another. For example, new objects could be added to a schema while the schema was suspended from resolution, potentially interfering with a scope (dictionary) traversal due to a change (addition or deletion) of an object in the scope. Recovering from an error during resolution of objects from a remote schema could potentially mean trying to ignore (from then on) such remote objects, a non-obvious situation.

Our new implementation dedicates a single pass to resolving all inter-schema references before they are actually needed. Objects are not copied. Instead, a pointer is retained to the final (defining) schema. During object resolution, the names are looked up in a dictionary of such pointers. Separate dictionaries are kept for Use and Reference pointers, allowing the code to account for the different behaviors required.

2. To be fair, we gained this experience having rewritten this code several times! Steve Clark wrote the first version, a non-OO, single-pass-resolution version for the "Tokyo" draft. After a short-lived version in C++, he rewrote it in C but with a hand-built OO engine for N496. Don Libes and Dave Briggs (PDES, Inc.) then added support for N14 (referred to in this paper as the *previous* implementation). Later, Don removed the OO support and then rewrote it again changing it to a multiple-pass design for N151 (the *new* implementation).

Suspicious Constructions

By default, our implementation reports errors and warnings based on what it finds as it transforms schema files into their in-memory representations. If requested by a user, the translator can look for errors that would not otherwise be reported.

There are an infinite number of errors or more interestingly "suspicious constructions" which could be checked. For example, shadowed types are not illegal but are probably a mistake nonetheless. (Or they might not be.) Classical compiler optimization such as code-hoisting can discover likely errors in semantics, yet this does not seem appropriate to the realm of the translator. Lacking experience in what people find valuable in the way of schema analysis, we encourage people to tell us what kinds of things are feasibly worth checking.

Parsing

Perhaps the least amount of change has occurred to the parser and scanner. Besides the continual modifications it has undergone to account for the evolution of EXPRESS, the parser's most significant change has been the improvement of syntax errors.

The parser has always been written in Yacc [7], and one of Yacc's shortcomings has been its crude error messages. We have improved that in large part by incorporating a mechanism [8] to report what tokens were encountered and what were expected. The parser also provides good explanations of comment or scoping problems.

The Yacc parser is composed of machine-generated C, and as such is not particularly efficient (although we use GNU's Bison [9], a more efficient version of Yacc). Nonetheless, we have not tried to rewrite an EXPRESS parser by hand. While such an effort would produce more efficient code, we expect that it would be a lengthy task, and further changes would be extremely painful.

The scanner is similarly constructed by Lex [10] (or Flex, a faster Lex). Currently, the parser and scanner together take 75% of the time used by parsing and resolution of an EXPRESS schema. We speculate that a hand-built pair could reduce the total run-time by as much as 50%.

Changes Motivated by Esthetics

We resisted the desire to make changes purely for esthetic purposes (we have enough to do that we can direct our esthetics on necessary endeavors rather than unnecessary ones). While the changes mentioned in this section were partly motivated by esthetics, there was also some impact on efficiency as well in each case. Of even greater benefit was that comprehensibility of the system had improved dramatically.

Resolution – Single-Pass vs. Multiple-Pass Implementation

The original implementation used two passes. The first pass was almost entirely dedicated to parsing (using Yacc) and will not be mentioned further. The second pass was devoted to resolution. The resolution pass is worth studying simply because it was just a single pass – at least, conceptually.

In reality, the data structure produced by the parser was traversed depth-first. As unknown objects were encountered, they were resolved. Consider the attribute declaration:

ATTRX: TYPEX;

Here, ATTRX is being defined. Clearly, TYPEX must be resolved first. (Otherwise, backpatching or something similar must be used.) If the system has not yet encountered a definition for TYPEX, it must suspend the definition of ATTRX and find and resolve TYPEX.

This could be complicated if TYPEX is Used from another schema. Schemas that made references to each other could cause considerable processing before returning. It was not uncommon to find the stack hundreds of calls deep. (This was particularly annoying during debugging!) More importantly, each object could end up being revisited again (and again) as it was encountered in its normal sequence of resolution. Indeed, this is the expected result as objects are invariably referenced multiple times. Thus, despite the seeming one-pass appearance, objects were typically visited many times. A fair amount of time was devoted towards things such as preventing objects from being resolved twice or recording a resolution failure.

When we recognized that the traversal time was not expensive, we broke up the resolution pass into a number of passes. The intent (and result) is that no objects are dereferenced before they are resolved. For example, all the types are resolved before any attributes are resolved, so in the above example, TYPEX would be resolved by the time ATTRX was ready for resolution.

The specific passes are as follows:

Pass 1: Parse

Tokens are scanned and a parse tree is built representing a collection of schemas. A dictionary is created for each scope, and names are entered into the dictionaries during parsing.

If any referenced schema is unknown, a search is made to find an appropriate file and the Parse pass is repeated.

Pass 2: Resolve Use and References

Use and Reference lists are traversed and connections are created to objects in the defining schemas.

Pass 3: Resolve Subtypes and Supertypes

Entity subtypes and supertypes are resolved.

Pass 4: Resolve Explicit Types

All explicit type definitions, attribute types, return types, etc., are resolved.

Pass 5: Resolve Expressions, Statements, and Implicit Types

Expressions and statements are resolved. Types for Alias and other implicitly-typed variables are resolved here since they depend upon expression return types, and the types are not necessary for earlier results.

Pass 6: Application-Specific Backend

While not restricted to following all the other passes, application-specific code is usually performed at this point.

Ordering Diagnostics

Using a multiple-pass approach solved a different problem as well. In the previous implementation, diagnostics (for the user) were sorted before being presented to the user. The intent was that the user would want to go through a schema fixing errors top-to-bottom, as they seemingly appear.

Once we adopted the multiple-pass approach, we noted that the most serious errors were generated first. For example, an error such as “could not find SCHEMA S1” would always be generated before “could not find ENTITY E1 USEd from SCHEMA S1”. Indeed, hundreds of later errors could be generated by having an incorrect schema specification. Ordering these by line number could mean that the schema diagnostic itself might be seen only after all the others.

Obviously, the solution was not to reorder the error messages at all. A pleasing benefit is that the user sees error messages as they are generated. Earlier, the messages had to be buffered for sorting, with the result that users did not see the first message until the program had completed detecting all the errors. A lesser albeit significant benefit is that further time and space is saved simply by not having to do sorting in the first place. (The sort capability remains as an option for users who want it.)

Some other minor changes relating to diagnostics are as follows:

- Formats were modified to be acceptable to “GNU emacs compile-mode”. Using this ability of the GNU emacs editor [9], the user can move the cursor from one diagnostic to the next in one window, while in a second window, the schema is automatically repositioned to the line that the diagnostic describes.
- Each diagnostic includes (at least one) file name and line number (and a scope identification if appropriate) to identify the context.
- Syntax diagnostics describe not only the unexpected token, but a choice of expected tokens. Semantic diagnostics have been improved as well (although not in any consistent way).

Information Hiding

The previous implementation made a large attempt at information hiding. Even though it was entirely written in C, no direct access to structure elements was permitted. All access was granted via function calls (or macros) so that the underlying representation could be changed without impacting the users.

This worked well, but to a point. For example, toolkit users (i.e., application programmers) often had to be concerned with the form of the data returned by the access functions. For example, a function to return the entities in a schema had to return them using yet another data structure such as a list, dictionary, etc. Usually, only one access function was provided, and if the data type of the return value wasn’t obvious from the name, (e.g., SCHEMAget_entity_*list* vs. SCHEMAget_entities), then users had to be aware of the specific function (e.g., *LIST*get_next vs. *DICT*get_next) with which to read the return value.

While most functions were easy to recode in the new implementation, some were impossible. In particular, the removal of the OO system meant that there was no way to find out the type (i.e., class) of an object. This essentially crippled the usefulness of some of the other functions. For example, SCHEMAget_objects could return a list but there was no way to tell what any object in the list was. When the OO system was removed, the user interface *had* to change.

Of course, we still believe strongly in information hiding. But in our case, the underlying system changed enough so that there was no way to hide the differences from the user. We tried to keep as many of the original interfaces as possible, but had to redesign quite a few of them. Toolkit applications will need to be converted by hand. We are not entirely happy with this, but we feel that the trade-off in speed is worth it. We have already converted several applications.

Extensions

The philosophy of the EXPRESS specification is to ignore the OS environment if at all possible. For example, file names, directories, etc. are all outside the scope of EXPRESS. We agree with this intent. In our environment, we found it useful to specify only a few environment details.

Mapping Schemas to Files

For any kind of processing, it is necessary to understand how files map to schemas. By default, a schema is assumed to be stored in a single file. We provide an explicit mechanism and an implicit mechanism for referencing objects or schemas not stored in the current file.

Explicit References via Include

It is possible to logically insert other files during analysis by use of an Include statement. This is similar in spirit to the obsolete Include statement from earlier specifications. It is best to think of Includes as a preprocessing phase of the implementation that has nothing to do with the language proper.

Include statements can appear outside a schema or at the top-level of a schema. Included files are not restricted to including schemas, but may include, for example, a set of entities, a rule, etc. For example:

```
INCLUDE 'schema-file.exp';
```

While the Include statement is occasionally useful, schemas using it do not conform to the specification. Because of this drawback, we therefore discourage its use. We leave it in primarily as a comparison between it and the method described next.

Implicit References via Use and Reference

Referencing a schema that is not defined in the file (or included from another file) causes the implementation to search for a file with the same name as the schema and with a ".exp" extension in the directories named by the environment variable EXPRESS_PATH. For example, in the C-shell, one could say:

```
setenv EXPRESS_PATH "/pdes/data/part42 /pdes/data/part202 ~/part"
```

This would define three directories in which to search for schema files. Following UNIX conventions, the first two are absolute references within the file system while the "~" names a directory relative to a particular user's hierarchy of files.

Imagine a schema being processed contains the following statement:

```
USE FROM TOOL_SCHEMA;
```

If TOOL_SCHEMA is already defined, either in a file which was read earlier or the current file, the reference can immediately be completed and no external file need be read. Otherwise, the directories named by EXPRESS_PATH are searched in order for a file by the name "tool_schema.exp". If such a file is found, it is parsed. If TOOL_SCHEMA is found, the previous USE processing is resumed. If the schema is not found, a diagnostic is issued.

In order to facilitate this, we recommend that all schema files have symbolic links to the names of any schemas within them that are likely to be externally referenced from the other schema files. Stable schemas may have symbolic links placed in a directory of stable schema files, while unstable schemas should be referenced from a specific schema directory.

For example, imagine that the directory for stable schemas is /pdes/data/schemas/standard while "part 42" is a definition still undergoing evolution. In this case, the appropriate command might be:

```
setenv EXPRESS_PATH "/pdes/schemas/part42 /pdes/schemas/standard"
```

If EXPRESS_PATH is not set, the default path of "." (the current directory) is used.

Revisiting of the Schema-Independent Tool Concept

Earlier (see Introduction on page 2), we described the idea of a schema-independent library for translating and manipulating EXPRESS schemas. This idea has its advantages and disadvantages. The primary disadvantage is that once a schema is standardized, applications waste time rereading, reparsing, and retranslating schemas each time they are run. This was particularly annoying with the previous implementation but is substantially ameliorated by the speed of the new implementation. Indeed, in actual applications of any significance, start-up overhead of the application matches and in many cases overwhelms any overhead of schemas translation. Even schema-dependent applications must provide for some flexibility[3] often in the form of a configuration file that is read dynamically upon start-up.

One such application is an EXPRESS to C++ translator [11][12]. The obvious use for this is to produce schema-dependent applications. The result of this is that users have a choice of whether to produce schema-dependent or schema-independent applications.

Some advantages of the schema-independent approach are:

- Schemas can be modified as necessary without rebuilding the application. Despite the presence of standards, there is always a reason why one day you will need to modify the schema.
- Only a single copy of the application is necessary even if used with different schemas. Already, we commonly build 1Mb applications (using our C++'ed EXPRESS). Even a few of these can rapidly use up disk space. The problem is exacerbated by supporting multiple machine architectures, which requires separate programs for each.

3. Otherwise, why would you ever have to run it more than once?

- While the appropriateness of the paradigm is arguable for standardized schemas, many schemas are in development, and rebuilding applications after each schema change is expensive. With the spread of EXPRESS as a modelling language outside the STEP project, schema-independent applications will continue to be an efficient approach to dealing with rapidly changing schemas.
- While OO may be good for modelling, it is not necessarily good for execution speed. While the EXPRESS-to-C++ approach avoids redundant retranslation, a programmer may well recode parts (or all) of the C++, possibly in C or even in assembler, much the way that any space- or speed-sensitive software is created.

Conclusion

During the past few years, we have learned some important lessons having to do with programming and modelling. Along the way, we rewrote the NIST EXPRESS Toolkit from top-to-bottom. While users will have to make changes to any applications using the toolkit, we believe that they will be happier with the result.

The toolkit adheres to N151, the current EXPRESS specification. The diagnostics have been improved and are believed to be much more explanatory. Disk space and running time have been significantly decreased. These improvements rejustify the premise on which the toolkit was original designed.

References

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

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

[3] Clark, Steve N., “QDES User’s Guide”, NISTIR 4361, National Institute of Standards and Technology, Gaithersburg, MD, June 1990.

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

[5] 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.

[6] 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.

[7] Johnson, S.C., “Yacc: Yet Another Compiler compiler”, *UNIX Programmer’s Manual*, Seventh Edition, Bell Laboratories, Murray Hill, NJ, 1978.

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

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

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

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

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