U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 4371

National PDES Testbed Report Series

Fed-X: The NIST Express Translator

Revised November, 1990

NATIONAL PDES TESTBED

NIST

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 4371

National PDES Testbed Report Series

NATIONAL PDES TESTBED

Fed-X: The NIST Express Translator

Stephen Nowland Clark

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary of Commerce

National Institute of Standards and Technology
John W. Lyons, Director

November 19, 1990

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied

Table Of Contents

Fed-X: The NIST Express Translator

Stephen Nowland Clark

1 Introduction

The NIST (Federal) Express Translator (Fed-X), and the associated Express Working Form, are Public Domain software tools for manipulating information models[1] written in the Express language [Schenck90]. The Express Working Form is part of the NIST PDES Toolkit [Clark90a]. It is intended to be used to provide the input to various conceptual-schema-driven applications in a STEP implementation. For example, tools such as QDES, a prototype STEP model editor developed at NIST [Clark90d], and the STEP Working Form with its associated STEP physical file parser, STEPparse [Clark90b], have been written independently of any particular information model. Fed-X-based translators are used to provide the information model definitions to drive these applications. This approach results in smaller applications (which need not have entire information models embedded within them), as well as insulating these applications against changes in the conceptual schema and, to a certain extent, in Express itself. Indeed, an application such as STEPparse can be used with different conceptual schemas, or different versions of the same schema, without modification. QDES has been used to edit STEP product models in the context of several different Express information models.

A primary goal in the development of Fed-X was to provide a clean back-end interface, in order to allow various output modules to be easily plugged into a basic front-end parser. To accomplish this, the Fed-X parser populates a set of data structures (the Express Working Form, or WF) containing all of the information in an Express specification. Fed-X can then dynamically load one or more output modules. Each module walks through the data structures, extracting relevant portions of the available data and producing an appropriately formatted output file. Two Fed-X output modules are provided with the NIST PDES Toolkit[2]. One of these produces Smalltalk-80™ class definitions [Clark90e] for use with QDES. The other forms the back end of Fed-X-SQL, a translator which produces relational database table definitions in SQL from an Express information model [Morris90] [Metz89].

---

1. The terms *information model*, *data model*, and *conceptual schema* are used interchangeably throughout this document.

2. In the past, the GMAP Batch Input Language (BIL) has also been generated by Fed-X [Perlotto89], as have various experimental sets of C language data structures and functions. However, these output modules are no longer supported, and cannot be used with the current version of the Working Form without modification. The latter were early attempts at a working form for STEP product models, now replaced by the schema-driven software described in [Clark90b].

1.1 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 [Smith88]. 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 CALS (Computer-aided Acquisition and Logistic Support) 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 PDES data. This NIST PDES Toolkit is an evolving, research-oriented set of software tools. This document is one of a set of reports which describe various aspects of the Toolkit. An overview of the Toolkit is provided in [Clark90a], along with references to the other documents in the set.

2 Implementation Environment

Fed-X was developed on Sun Microsystems Sun-3™ and Sun-4™ series workstations running the Unix™ operating system. The Working Form is implemented in ANSI Standard C [ANSI89]. The Fed-X parser itself is implemented in Yacc and Lex, the Unix languages for specifying parsers and lexical analyzers. In the NIST development environment, the grammar is actually processed by Bison, the Free Software Foundation's[1] implementation of Yacc. The lexical analyzer is produced by Flex[2], a fast, Public Domain implementation of Lex. The C compiler used is GCC, also a product of the Free Software Foundation, although the Working Form code does not specifically depend on any particular compiler.

3 Running Fed-X

Fed-X takes several optional command-line arguments:

```
fedex [-d <number>]
      [-e <express>]
      {-w|-i all|none|<warning>}
```

The `-d` option controls the debugging level; the argument can range from 0 (the default) to 10. The Express source file is specified with `-e`; if no `-e` option is given, Fed-X reads from standard input. The last two options control which warning messages Fed-X will produce. `-w` is used to turn on warning classes and `-i` (ignore) to turn them

1. The Free Software Foundation (FSF) of Cambridge, Massachusetts is responsible for the GNU Project, whose ultimate goal is to provide a free implementation of the Unix operating system and environment. These tools are not in the Public Domain: FSF retains ownership and copyright privileges, but grants free distribution rights under certain terms. At this writing, further information is available by electronic mail on the Internet from gnu@prep.ai.mit.edu.
2. Vern Paxson's Fast Lex is usually distributed with GNU software. It is, however, in the Public Domain, and is not an FSF product. Thus, it does not come under the FSF licensing restrictions.

off. A parameter of `all` behaves in a predictable fashion, instructing Fed-X to enable/disable all of the warning classes initially; similarly, `none` instructs Fed-X to begin with no warning classes enabled/disabled. Allowable values for `<warning>`, with their interpretation and default values, are:

- `subtypes` - Warnings about subtypes: Fed-X only traverses the class hierarchy by way of superclass information, so problems in subclass lists can "safely" be ignored. Default: on.
- `code` - Warnings about problems in algorithms and `where` clauses. Fed-X does not yet handle all of Express' scoping rules properly, nor does it attempt to compute the return types of expressions, so some of these warnings may be extraneous. Default: off.
- `assume` - Warnings about schema names listed in assume directives. Default: on.
- `comment` - Nested comment warning. Default: off.
- `shadows` - Warnings about overloaded names. The scoping rules of Express can disambiguate these shadowed definitions, but cannot be invoked outside of Express, e.g. in STEP files. Default: on.

Fed-X can be built in two different ways, resulting in different interaction patterns. For many applications, a single output module is bound into Fed-X at build time. In this statically linked case, after the first two passes are completed, the user is normally prompted for a single file name. This is the name of the file to which Fed-X's output will be written. In the other (dynamically linked) version, no specific output module is loaded at build time. In this case, when the first two passes are complete, the program asks for an output module. If the file named is an appropriate object file, it is loaded and an output file name requested. This is the name of the file to which the output will be written. Another output module is then requested, and this sequence continues until an empty line is entered as the name of the output module, which signals Fed-X to exit. This dynamic loading facility is available only under BSD4.2 Unix and its derivates.

4 Design Overview

Fed-X is a three-pass parser. The first two passes are the standard parsing and symbol-table resolution passes of a traditional compiler. The third is a flexible output generation pass. The Working Form which is produced by the first two passes consists of tightly-linked data structures which directly reflect the structure and contents of the Express source. The third pass, which can be tailored to various specific applications, traverses these data structures and produces output in a specified format.

4.1 Fed-X Control Flow

4.1.1 First Pass: Parsing

The first pass of Fed-X builds up a set of loosely-connected data structures which completely represent the information in the Express input. This pass makes no attempt at resolving most name references; thus, the resulting data structures are linked only indirectly by names: in order to resolve a function call, the name of the function must be looked up in the symbol table for the appropriate scope. The entire structure of the file is represented at this point, however. If any syntax errors are encountered, the parser attempts to print meaningful error messages and to continue parsing.

4.1.2 Second Pass: Reference Resolution

In the second pass, an attempt is made to resolve all names. An error message is generated for any reference to an undefined name and for any use of a name in an inappropriate context (e.g., an algorithm name as the type of a variable). Some checks are made on the consistency of the model during this pass. For example, one check ensures that every supertype of a given entity also lists the entity as a subtype, and vice versa. Also during this pass, warnings may be issued about names which are multiply defined in different scopes. Express has a hierarchical scoping mechanism to disambiguate these names, so that such overloading is allowed. In practice, however, Express models are mapped onto STEP physical files, which have no notion of a hierarchically scoped information model. When this "flattening out" of the model takes place, overloaded names may conflict; hence the need for these warnings about shadowed definitions.

4.1.3 Third Pass: Output Generation

After the first two passes have built and linked the in-core Working Form, the third pass kicks in to write the output. This pass can load several output modules in succession, so that several file representations of the Express input can be produced from a single parse. Alternatively, a specific module can be built into the translator, and this dynamic loading phase bypassed.

4.2 Working Form Data Structures

The Express Working Form is designed in object-oriented fashion, with one data abstraction corresponding to each concept in Express. Thus, there are abstractions which represent types, entities, variables (which include entity attributes and formal parameters, as well as local variables), expressions, statements, algorithms, and schemas. An additional concept which recurs in Express, and which is represented by a corresponding data abstraction, is that of a scope, which is, in effect, a symbol table. Algorithms, schemas, and entities all introduce their own local scopes.

In the following sections, we examine each abstraction in turn. Although each abstraction parallels the corresponding construct in Express quite closely, so that the descriptions below often seem to be echoing [Schenck90], bear in mind that the objects described are actually the abstract data types of the Express Working Form.

4.2.1 Constant

The Constant abstraction represents symbolic constants. In the current implementation of the Working Form, constants appear only as elements of an enumerated type. A constant is named, and is marked with a type. The type of an enumeration constant simply points back at the enumeration of which it is an element. Each constant has a value, which can be of any C type (although it should be compatible with the type of the constant); in the case of enumeration constants, this value is always an integer.

4.2.2 Type

The Type abstraction is used to represent Express types. Every type has a name, which is empty in many cases. When it is not, the type represents a type declaration, as in the `TYPE <id> = <type> END_TYPE` construct of Express. When the name is empty, the type represented appears within some other context - perhaps as the type of a function parameter or the base type of an aggregate. A type may have a list of constraints (WHERE rule) associated with it; these constraints restrict the legal values of the type.

Several classes of types are represented, including simple types (numeric, logical, string), enumerations, various aggregates, entity types, and select types. Several type classes are implicitly or explicitly subclasses of other type classes. Thus, boolean is a subtype of logical, and the various classes of aggregation types are subclasses of the general aggregate type. The attributes of a type depend on its class. Thus, integer, floating point, and string types may have a precision specification: an expression which constrains the number of significant digits or characters allowed in a value of the type. An enumeration type includes a list of the enumeration constants which are the allowable values for the type.

Every aggregate type (which may be an array, bag, list, set, or general aggregate) includes a base type, which indicates the type of objects which can be inserted into an instance of the aggregate type. In addition, an aggregate type may have lower and upper bounds. In the case of an array, these expressions indicate the first and last allowable index into the array. For other aggregate types, these expressions constrain the total number of objects which can (must) appear in an instantiation. If the bounds are not specified, they are assumed to be 0 and infinity, respectively. Two flags are also associated with each aggregate type, corresponding to the UNIQUE and OPTIONAL keywords in an Express aggregate definition. The 'unique' flag, if set, indicates that all elements of an aggregate must be unique among themselves. As this requirement already applies to a set, the flag is not valid for a set type. The 'optional' flag, which applies only to an array type, indicates that all positions in the array need not be filled in a valid instantiation of the type - the array may contain null entries.

An entity type is simply one or more entities packaged as a type. No further information is added beyond the entity definitions themselves. Entity types exist to allow entity instantiations to be represented (c.f. STEP Working Form [Clark90b]), and to provide a clean mechanism for recognizing entity names in type contexts.

A select type consists of a list of selectable types. An instantiation of any of these selections is a valid instantiation of the select type. In this sense, the select is similar to the C language `union` construct and the Pascal variant `record`. In Express, the list of selections may only include references to named types.

There are two type classes, generic and number, which are distinguished by the fact that the corresponding Express types (GENERIC and NUMBER, respectively) cannot be instantiated. These can only be used as types of formal parameters to algorithms, where an actual parameter will provide an instantiation of a more specific type at run time.

A special type class is used to represent type references. These are (possibly qualified) references which appear in type contexts, but which are not yet resolved to a particular type. In normal operation under the control of Fed-X, they are replaced during the second pass by appropriate type constructs. A type reference uses an expression (see section 4.2.5) to record the qualified type name it represents. The components of this expression are identifiers, and they are combined into binary expressions with the dot operator.

There are several type constants available. These constants can be used to avoid creating multiple copies of some common types, including generic, integer, unbounded generic set, logical, etc.

4.2.3 Entity

The Entity abstraction represents Express entity declarations. Every entity consists of a name, and (possibly empty) lists of attributes, subtypes, and supertypes. In addition, an entity includes a boolean expression which describes the relationships among its various subtypes. The attributes are represented as variables which are defined in the local scope of the entity. The sub- and supertypes are themselves entities.

In order to give a hierarchical structure to an Express model, entities are arranged in a class hierarchy, as in the Object-Oriented world. This hierarchy is defined by the subclass and superclass lists of its component entities. As specified by Express, the class hierarchy provides for conjunctive as well as disjunctive subclassing: `foo SUPERTYPE OF (bar AND blat)` means that any instance of `foo` is also an instance both of `bar` and of `blat`, while `foo SUPERTYPE OF ONEOF(bar, blat, blit)` represents standard inheritance, in which an instance of `foo` is also either an instance of `bar` or an instance of `blat` or an instance of `blit`.

An entity may also include a list of uniqueness sets (from the Express UNIQUE rule) and a list of constraints (from the Express WHERE clause). Each uniqueness set is a list of attributes whose values, when taken together, must uniquely identify a particular instance of the entity. The constraints, if any, are expressions which compute logical results. Each must evaluate to `true` in a valid product model. These constraints can apply to individual instantiations of the entity as well as to the collection of all instances of the entity.

Since one possible way of looking at an entity class is as the collection of its instances, provision is made in this abstraction for maintaining this collection. Thus, it is possible to add instances to an entity, or to retrieve a list of all of the instances of an entity. This mechanism is used by the STEP Working Form.

4.2.4 Variable

The Variable abstraction is used to represent entity attributes and formal parameters to algorithms as well as local variables in a scope. A variable consists of a name, a type, a reference (or storage) class, an offset, and some flags. A variable may optionally have an initializer, which is an expression used to specify an initial value for the variable.

The reference class of a variable, meaningful only for entity attributes, may be 'internal,' 'external,' or 'dynamic,' corresponding to the three reference classes defined in Express. An 'internal' attribute can only be instantiated with an embedded entity in a STEP physical file. An 'external' attribute must be instantiated with a reference to another entity instance in the physical file. A 'dynamic' attribute can be instantiated in either way.

A variable's offset indicates its position in a storage block. Thus, the offset of a local variable is its offset into the data space of the scope in which it is defined, while the offset of an entity attribute is its position relative to the first attribute of the entity. It is important to realize that, in the latter case, this offset is not sufficient to locate the attribute in an instantiation of the entity, since this total offset cannot be determined from the entity definition alone. To see this, consider entities `A` and `B`, each with a single attribute (call these `aa` and `bb`, respectively) The offset to `bb` in an instantiation of `B` is 0. But now suppose there is a third entity class, `C`, which inherits from both `A` and `B`, in that order. Then the offset to `bb` in an instance of `C` must be 1, even though `bb` is inherited from `B`, where its offset was 0. Thus, a variable's offset may not be a useful piece of information by itself.

The 'optional' flag is used with entity attributes, and indicates that the attribute need not have a value in a valid instantiation of the entity. A variable representing an entity attribute can also be marked 'derived,' indicating that the attribute value is always derived from the values of other attributes, and can never be specified by a user. The 'variable' flag, meaningful for formal parameters, indicates that the parameter is to be passed by reference, i.e., it can be modified by the receiver.

4.2.5 Expression

Expression is one of the more complex abstractions, simply because of the wide variety of expressions found in Express. There are five basic classes of Expressions, some of which are further divided into conceptual subclasses: literals (including integer, logical, real, set, and string literals), identifiers, operations (including unary operations and binary operations), function calls, and queries. Every expression includes a type, which is the type of the value it computes. Although this type is intended to be computed automatically, it currently is neither computed nor used by the Working Form code, except in the case of a literal. In this case, the type is an implied part of the definition of the literal's class.

Literal classes exist for most of the concrete simple types (as opposed to the abstract simple types, NUMBER and GENERIC). Boolean literals do not exist in Express; they are interpreted as logical literals instead. There may also be set literals (notably, the empty set). There are several literal expression constants representing, for example, zero, infinity, and the empty set.

An identifier expression represents a reference to a variable. It consists simply of the variable referenced. (Simple) identifier expressions can be composed using (binary) field reference expressions to form the complex qualified identifiers which Express provides.

An operation expression includes one (unary operation) or two (binary operation) operands, which are themselves expressions, and an operator, such as addition, negation, array indexing, or attribute extraction. All of the operations of Express are supported.

A function call is composed of an algorithm (which may not be a procedure) and a list of actual parameters to the algorithm. The actual parameters to the function call are themselves expressions. Entity subtype expressions (see section 4.2.3) make use of a closely related expression class, the oneof expression, which consists of a list of entity references.

A query expression represents the set-theoretic "set of all *x* in *X* such that ..." construct. It consist of a domain set (*X*), a temporary identifier which represents each element of the domain successively (*x*), and a list of conditions to apply to each *x*. The result computed is a set containing all of the values of *x* which satisfy the constraints.

4.2.6 Statement

The Statement abstraction is used to represent the wide variety of statements which occur in Express. There are many classes of statements, including assignments, case statements, conditionals, loops, procedure calls, returns, and with statements. A series of statements may be combined into a single compound statement.

An assignment statement consists of a left-hand-side expression, which must be assignable (this limits the expression to a possibly qualified identifier, although the restriction currently is not enforced by the Working Form), and a right-hand-side expression, computing the value to be assigned.

A CASE statement is, as in Pascal, a multi-branch conditional. It contains an expression (the case selector) and a list of branches. Each branch is a case item, represented by the Case Item abstraction. A case item consists of a list of one or more values against which the selector will be compared and a statement to be executed if the selector matches on of these values.

The looping construct in Express is quite general, combining the functionality of the `repeat .. until`, `while .. do`, and `for` loops of modern programming languages. An Express loop consists of a controlled statement (the body of the loop) and a list of loop controls. There are three classes of loop control: increment (corresponding to a `FOR` loop), until, and while. The first consists of a controlling identifier expression, initial and terminal expressions, and an optional increment expression, which defaults to 1 if not present. The controlling identifier takes on successive values from the initial to the terminal expressions, and is incremented by the increment expression on each iteration. An until control consists of a single expression (which must compute a boolean result); it causes the loop to terminate when this expression evaluates to `true`. Similarly, a while control causes the loop to terminate as soon as its single expression evaluates to `false`.

A procedure call is very much like a function call, with the exception that the algorithm is expected to be a procedure, rather than a function or rule. The procedure call statement includes a list of expressions, representing the actual parameters to the call.

A RETURN statement is the mechanism by which a function reports a value to its caller. It contains a single expression, which computes the value to be returned.

A simple statement is one which consists of a single keyword. There are two such statements in Express: ESCAPE and SKIP. No statement class is provided for simple statements; rather, they are represented by statement constants, unique instances the Statement abstraction itself.

Finally, Express includes the WITH statement, which resembles Pascal's construct of the same name. It includes a controlled statement and a controlling expression which provides (optional) partial qualification to any expression in this statement. If a name in the controlled statement cannot be resolved, an attempt is made to resolve the name as if it were prepended with the controlling expression. The Working Form currently does not attempt to acknowledge WITH statements when resolving identifiers.

4.2.7 Algorithm

Express functions, procedures, and rules are each represented by a subclass of the Algorithm abstraction. A procedure is simply a sequence of statements. A function is a sequence of statements which computes a result and returns it to the caller. A rule is a special kind of function whose result is always a boolean (logical). A rule also has slightly different scoping rules than other algorithms, to allow it to manipulate entity classes as well as instances.

Any algorithm consists of a name, a list of formal parameters (which are represented by variables), and a list of statements forming the body of the algorithm. In addition, a function has a return type. A rule implicitly returns a logical value. This value is computed by a list of constraints (WHERE clause), which is evaluated after the statements which form the rule body.

4.2.8 Scope

All scoping and symbol table functionality are managed by the Scope abstraction. A local scope is established by each algorithm, schema, and entity. For this reason, each of these abstractions is considered to be a subclass of scope, thereby inheriting all of its functionality. Pascal-like hierarchical scoping and inheritance are implemented by having each scope point to its immediate containing scope(s), if any. For example, an algorithm's local scope points to the scope in which the algorithm is defined; an entity's scope may have several parents: the scope in which the entity is defined, and all of the supertype entity scopes. In its role as a symbol table, a scope includes definitions of various names as entities, types, variables, algorithms, constants, and schemas.

A scope can be queried for its definition of a particular symbol. If the scope does not itself define the symbol, its superscopes are in turn queried, and so forth. If no definition can be found, the query fails.

In order to support the ASSUME directive of Express, a scope includes a list of imported schemas. When a query to the scope (including all ancestor scopes) fails, the local scope of each schema on the import list is queried for an appropriate definition.

Finally, a scope includes a list of definitions which are private. These definitions cannot be seen outside the scope, even if the scope is ASSUME'd elsewhere.

4.2.9 Schema

The Schema abstraction represents the Express construct of the same name, which is, in effect, a named scope. Most operations of interest are performed on the scope.

The object produced by the first two passes of Fed-X is a schema, which ultimately contains all of the definitions found in the source file. This corresponds to the fact that and Express source file, contains, at the highest level, a single `SCHEMA ... END_SCHEMA` construct.

5 Missing Features

Although Fed-X accepts all of the syntactic constructs of Express, the Working Form does not yet represent all of them; nor does it observe all of those which it represents. The MAP directive and the cardinality operator are both accepted and silently discarded. The PRIVATE directive is represented in the Working Form, but is ignored at the crucial moment: during symbol lookup. With statements are parsed and represented, but have no effect when identifiers are being resolved.

Although the full type system of Express is represented in the Working Form, type derivations are not performed. It is theoretically possible to assign a type to any expression on the basis of the operator and operands (or by looking up a function in the symbol table), but this functionality is not yet implemented. Thus, erroneous messages about type mismatches are sometimes produced simply because type information about certain expressions is not available.

Due to problems with the Express language definition, qualified identifiers may not always be interpreted properly. Problems are particularly common when dealing with enumeration identifiers. Similarly, Express allows a subtype entity to redefine an attribute which it inherits from a supertype. The effect of this redefinition on scoping remains an open issue, and so Fed-X currently does not allow it.

Fed-X responds robustly to semantic errors. Syntax error recovery is somewhat more haphazard.

Comments are discarded during lexical analysis and so have no chance of being recorded by the parser.

6 Conclusion

Although the Express Working Form in its current state is sufficient for current applications, it is only a matter of time before some of the missing features are required. In addition, Express is still evolving, and the software must continue to change with the language.

Fed-X has proven to be an effective tool for the creation of schema-independent applications based on STEP. Translators using each of the output modules distributed with the Express Working Form are in common use at NIST, as are three applications driven by Fed-X: QDES, STEPparse, and the NIST Oracle® database for PDES.

For further information on Fed-X, the Express Working Form, or other components of the Toolkit, or to obtain a copy of the software, use the attached order form.

A References

[ANSI89] American National Standards Institute, Programming Language C, Document ANSI X3.159-1989

[Clark90a] Clark, S. N., An Introduction to The NIST PDES Toolkit, NISTIR 4336, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[Clark90b] Clark, S.N., The NIST Working Form for STEP, NISTIR 4351, National Institute of Standards and Technology, Gaithersburg, MD, June 1990

[Clark90c] Clark, S.N., NIST Express Working Form Programmer's Reference, NISTIR 4407, National Institute of Standards and Technology, Gaithersburg, MD, September 1990

[Clark90d] Clark, S.N., QDES User's Guide, NISTIR 4361, National Institute of Standards and Technology, Gaithersburg, MD, June 1990

[Clark90e] Clark, S.N., QDES Administrative Guide, NISTIR 4334, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[Metz89] Metz, W.P., and K.C. Morris, Translation of an Express Schema into SQL, PDES Inc. internal document, November 1989

[Morris90] Morris, K.C., Translating Express to SQL: A User's Guide, NISTIR 4341, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[Perlotto89] Perlotto, K. L., The Use of GMAP Software as a PDES Environment in the National PDES Testbed Project, NISTIR 89-4117, National Institute of Standards and Technology, Gaithersburg, MD, June 1989

[Schenck90] Schenck, D., ed., Exchange of Product Model Data - Part 11: The Express Language, ISO TC184/SC4 Document N64, July 1990

[Smith88] Smith, B., and G. Rinaudot, eds., Product Data Exchange Specification First Working Draft, NISTIR 88-4004, National Institute of Standards and Technology, Gaithersburg, MD, December 1988

ORDER and INFORMATION FORM

MAIL TO:

NATIONAL PDES TESTBED

National Institute of Standards and Technology
Gaithersburg MD., 20899
Metrology Building, Rm-A127
Attn: Secretary National PDES Testbed
(301) 975-3508

Please send the following documents and/or software:

- [ ] Clark, S.N., An Introduction to The NIST PDES Toolkit
- [ ] Clark, S.N., The NIST PDES Toolkit: Technical Fundamentals
- [ ] Clark, S.N., Fed-X: The NIST Express Translator
- [ ] Clark, S.N., The NIST Working Form for STEP
- [ ] Clark, S.N., NIST Express Working Form Programmer's Reference
- [ ] Clark, S.N., NIST STEP Working Form Programmer's Reference,
- [ ] Clark, S.N., QDES User's Guide
- [ ] Clark, S.N., QDES Administrative Guide
- [ ] Morris, K.C., Translating Express to SQL: A User's Guide
- [ ] Nickerson, D., The NIST SQL Database Loader: STEP Working Form to SQL
- [ ] Strouse, K., McLay, M., The PDES Testbed User Guide

OTHER (PLEASE SPECIFY)

____

____

____

____

These documents and corresponding software will be available from NTIS in the future. When available, the NTIS ordering information will be forthcoming.

NIST

IST-114A
REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER: NISTIR 4371
2. PERFORMING ORGANIZATION REPORT NUMBER:
3. PUBLICATION DATE: AUGUST 1990

TITLE AND SUBTITLE

"FED-X: The NIST Express Translator"

AUTHOR(S)

Stephen Nowland Clark

PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

). SUPPLEMENTARY NOTES

I. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

The Product Data Exchange Specification (PDES) is an emerging standard for the exchange of product information among various manufacturing applications. PDES includes an information model written in the Express language; other PDES-related information models are also written in Express. The National PDES Testbed at NIST has developed software to maniputlate and translate Express models. This software consists of an in-memory working form and an associated Express language parser, FED-X. The design and capabilities of FED-X and the Express Working Form are discussed.

2. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

date modeling; Express; product data exchange; PDES; PDES implementation tools; schema-independent software; schema translation; STEP

3. AVAILABILITY

- [X] UNLIMITED
- [ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
- [ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
- [X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES: 13

15. PRICE: A02

ELECTRONIC FORM