THE NIST STEP CLASS LIBRARY (STEP INTO THE FUTURE)

Michael J. McLay
Katherine C. Morris

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Center for Manufacturing Engineering
Gaithersburg, MD 20899
THE NIST STEP CLASS LIBRARY (STEP INTO THE FUTURE)

Michael J. McLay
Katherine C. Morris

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Center for Manufacturing Engineering
Gaithersburg, MD 20899

August 1990
The NIST STEP Class Library
(STEP Into The Future)
Michael J. McLay (mclay@cme.nist.gov)
Katherine C. Morris (kc@cme.nist.gov)
Factory Automation System Division
National Institute of Standards and Technology

"Any meaningful exchange of utterances depends upon the prior existence of an agreed set of semantic and syntactic rules. The recipients of the utterances shall use only those rules to interpret the received utterances if it is to mean the same as that which was meant by the utterer."

--Helsinki Principle

Key Words
STEP, PDES, CALS, C++, data exchange standards, National PDES Testbed, CAD, CAM, CAPP, IGES, STEP Class Library, Express Language

Abstract
This paper describes a C++ class library that implements the *STandard for the Exchange of Product Model Data* (STEP). The STEP Class Library (SCL) is under development at the National Institute of Standards and Technology as part of the National PDES Testbed. It provides a core set of classes for tools used to validate the STEP conceptual data models and for STEP based application prototypes. The library is also intended to facilitate the development of STEP compliant applications. The current version of the library provides a file exchange mechanism based on a protocol defined by STEP. Consequently, users of the class library will not have to create an input/output mechanism for STEP.

The paper provides an introduction to STEP and the conceptual schema language, Express, in which STEP models are defined. The supporting classes that provide the STEP compliant input/output mechanism and the mapping between the Express specification and the C++ classes are described. The paper concludes with descriptions of some applications that use the class library and a discussion of future directions for the library.

1. The library and its source code, are under development by the National Institute of Standards and Technology (NIST), a U.S. government agency. As with all software developed by the government, the library is not subject to copyright restrictions.

Reprinted from the C++ at Work '90 conference proceedings with permission granted by The C++ Report. (No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.)
1.0 Introduction

This paper describes the Standard for the Exchange of Product Model Data\(^1\) (STEP) [NCGA90] [Smith89] and the STEP Class Library (SCL). STEP is a conceptual specification that forms a basis for communicating product information (e.g., shape, materials, electrical functions, part numbers, etc.). SCL provides C++ classes that can be used to implement applications such as electrical and mechanical CAD/CAM tools, Computer Aided Process Planning (CAPP) tools, manufacturing systems, product configuration control systems, and engineering analysis tools.

The STEP Class Library (SCL) defines classes and member functions to represent the product information defined by STEP. The library is to be used as a basis for the development of tools for testing the validity of the STEP conceptual data models and as a basis for a prototype implementation of the STEP Data Access Interface Specification (SDAIS) [Briggs90]. These applications are able to use SCL classes directly or as a link between a proprietary system and the STEP exchange mechanism. For a system that provides its own STEP interface, SCL can be used as a reference standard for conformance testing.

Following a brief discussion of STEP, the paper provides a short introduction to Express, the conceptual modeling language used for STEP. Next the components of SCL are described along with examples of Express product information definitions translated into C++ classes. Finally, examples of projects using SCL are described, and the paper concludes with a discussion of SCL development to date and thoughts on future directions.

2.0 The Impact of STEP

Although computer technology has expedited many business transactions, sharing product information within and between organizations has been problematic. In 1979 the National Institute of Standards and Technology (NIST)\(^2\), in cooperation with representatives from industry, began investigating solutions to this problem [Bloom89]. The original effort resulted in the Initial Graphic Exchange Specification (IGES)\(^3\)[NCGA90] [Smith89], which is suitable for exchanging information among CAD systems. STEP is an outgrowth of IGES and is intended to provide a basis for sharing product information at all levels and stages in a product's life cycle. The key differences between IGES and STEP are the breadth

\[^1\] STEP is a project of the International Organization for Standardization (ISO) Technical Committee on Industrial Automation Systems (TC 184) Subcommittee on Manufacturing Data and Languages (SC4).[NCGA90] [Smith89]

\[^2\] The Omnibus Trade Act of 1988 changed the mission of the National Bureau of Standards (NBS) to include promotion of technology transfer between government labs and private industry. This redirection prompted the name change to the National Institute of Standards and Technology (NIST). With the change in mission, NIST is now pursuing an even more active role in promoting commerce in the U.S.

of the information covered and the switch in focus from merely exchanging data to sharing information.

One of the lessons of the IGES effort is that transmitting data is not the same as transmitting the information necessary to fully describe a product. The Helsinki Principle, quoted at the opening of this paper, points out what is missing. Without the "agreed set of semantics and syntactic rules," exchanging data is meaningless. STEP expanded on IGES by specifying a consistent use of the semantics of product data in addition to the specification of the data.

Initially, the STEP development effort focused on building conceptual data models. The requirement to support the models of existing CAD and CAM systems made this task difficult because the models overlapped and conflicted. For example, a curve through space can be represented as a b-spline, as a list of curve segments, or as a non-uniform-rational b-spline (NURB). The STEP modelers undertook the very difficult job of defining mappings between the different representations of the same information. STEP currently consists of a group of clearly and formally defined conceptual data models and a physical exchange protocol based on these models [Alte88b]. Ultimately, the conceptual models will be integrated into a single model, and the exchange protocol will take other forms such as a standard interface to a shared database and/or common memory.

STEP is an ambitious standardization effort that involves several hundred individuals in twenty-six countries. The initial draft of STEP [Smith88], totaling over 6000 pages, was submitted to ISO in December 1988. The draft specification is divided into twenty-eight application areas, providing comprehensive coverage of product-related data from geometry and structural tolerances to electrical design and configuration management. When translated into C++, the draft STEP model produces over 1300 class definitions that interweave to ensure consistent use of data and semantics in describing a product.

The National PDES1 Testbed2 was established at NIST in 1988 as a neutral testing site to provide national leadership in the STEP development and testing effort. The facility is used to test the quality of the evolving conceptual data models [Mitch90] and to investigate the suitability of new technologies to the application areas covered by STEP [Fowler90].

2.1 Motivation for the NIST STEP Class Library

The principal objectives for creating the STEP Class Library (SCL) are to support the development of conceptual data model testing tools, prototype applications, and eventually, conformance testing. A secondary objective of developing this library is to reduce the economic barrier of introducing STEP into the workplace and,

---

1. *Product Data Exchange using STEP (PDES)* refers to the United States development activity in support of STEP. NIST serves as the Secretariat of the IGES/PDES Organization, which coordinates the United States activity.

2. Funding for the Testbed has been provided by the Department of Defense's Computer-Aided Acquisition and Logistic Support (CALS) Office.

The NIST STEP Class Library
consequently, to accelerate the rate of growth of STEP-compliant applications. To facilitate application development, the STEP data exchange mechanism is built into member functions in the class library, thus eliminating the need to rewrite this code for each application.

In addition, the use of a standard class library can also reduce ambiguities that could potentially be read into the specification. With over 6000 pages in the specification, it is likely that some misinterpretation could occur. The value of a reference library in simplifying implementation and ensuring conformance was proven with the X Window System Project [Scheifler88]. The rapid acceptance and almost universal conformity to the X Window System protocol can be attributed largely to the fact that the protocol was completely covered by a layer of C library calls. For the X Window System, the reference implementation also provided a convenient mechanism for the testing of protocol implementations in other languages. SCL has been developed with the hope that it will have a similar impact on STEP implementation and acceptance.

3.0 The Approach to Implementing STEP

The STEP conceptual model is specified in the language Express [Schenck90]. Express is a conceptual modeling language that defines how data are interrelated. While Express resembles many computer languages and has an LRN(1) syntax, it is not an executable language. Several parsers for the language are available to check the syntax of conceptual data models written in Express and to serve as front ends for translators between Express and other languages. At NIST an Express parser is used to translate some of the constructs of Express into SQL (Structured Query Language) [ANS186], Smalltalk [Goldberg85], and C++ [Ellis&Strou90]. Eventually the standard will provide common memory and shared database implementation rules, but for now the definition is limited to a single exchange format -- that of an exchange file.

3.1 The Express Language as a Conceptual Modeling Language

The components of the Express language addressed in this version of SCL are schemas, types, entities, and constraints. The following sections describe each of these components in more detail.¹ The examples used in the balance of the paper are derived from schemas in the draft specification [Smith88].

3.1.1 Schema

A schema is a collection of the information needed to describe a model in the terms of a given discipline. A schema consists of types, entities, and constraints. They may be nested and/or include other schemas. A collection of schemas is needed to represent the information required for a complete product model.

¹ Functions and procedures will be mapped into C++ in a future release of SCL.
3.1.2 Type

In Express the concept of *type* is similar to that of a data type in a programming language. Express contains a limited set of built-in types from which other types can be defined. For example, the following statement declares `inspection_process` to be of type *STRING*.

```
    TYPE inspection_process = STRING;
    END_TYPE;
```

Express also provides enumerated types in a form similar to C and C++. The position in a list of enumerated items determines the value associated with the items. The value of the first item is less than the second; the second is less than the third, etc. The example below is an enumerated type in Express.

```
    TYPE coordinate_system_type = ENUMERATION OF
        (rh_rectangular, rh_cylindrical, rh_spherical,
        lh_rectangular, lh_cylindrical, lh_spherical);
    END_TYPE;
```

Several other constructs for *type* definitions are available for Express but will not be discussed in this paper.

3.1.3 Entity

An entity represents a data structure similar to a *table* in a relational database, a *struct* in a C program, or a *class* in a C++ program. Entities are organized hierarchically; an entity can have zero or more *subtypes* and/or *supertypes*. Three types of *attributes* are the components of entities: explicit, inherited, and derived. *Explicit* and *inherited attributes* indicate the set of data needed to represent an instance of an entity. *Explicit attributes* are specified inside of an entity declaration just as a member is declared inside of a C++ class definition. An *inherited* attribute is specified in a supertype entity in Express just as a member is inherited from a base into a derived class in C++. Finally, a *derived attribute* is calculated from other attributes by using an algorithm defined in the conceptual data model. In C++ a member function is the closest construct to a derived attribute. In the example shown in Figure 1, the explicit attribute `local_coordinate_system` would be an inherited attribute in the entity `cartesian_point`. 
ENTITY geometry
SUPERTYPE OF (point XOR
vector XOR
curve XOR
surface XOR
coordinate_system XOR
transformation XOR
axis-placement);
  local-coordinate-system : OPTIONAL coordinate-system;
  axis : OPTIONAL transformation;
END_ENTITY;

ENTITY curve
SUPERTYPE OF (line XOR
conic XOR
bounded-curve XOR
offset-curve)
SUBTYPE OF (geometry);
WHERE
  arc-length-extent(curve) > 0;
END_ENTITY;

ENTITY line
SUBTYPE OF (curve);
  pnt :cartesian-point;
  dir :direction;
WHERE
  coordinate-space(pnt) = coordinate-space(dir);
  coordinate-space(line) = coordinate-space(pnt);
  arc-length-extent(line) > 0;
END_ENTITY;

ENTITY point
SUPERTYPE OF (cartesian-point XOR
point-on-curve " XOR
point-on-surface
point-on-surface ")
SUBTYPE OF (geometry);
END_ENTITY;

ENTITY cartesian-point
SUBTYPE OF (point);
  x-coordinate : REAL;
  y-coordinate : REAL;
  z-coordinate : OPTIONAL REAL;
DERIVE
  space : INTEGER := coordinate-space(z-coordinate);
END_ENTITY;

Figure 1. Example of Express entity definitions from the Geometry model.

3.1.4 Constraint

Constraints both internal to an entity and between entities can be represented in an
Express conceptual data model. Internally, an entity may have its attributes
constrained in terms of uniqueness and existence through the use of key words in the entity definition. For example, the entity `product_assembly_definition` in Figure 2 uses the key word UNIQUE to constrain the attributes `document_number` and `schematic_reference`. A where clause can be used to further constrain the domain of an attribute's value. In the example, the entity is constrained by the where clause that requires that the entity not be in the `component_list`.

```plaintext
ENTITY product_assembly_definition;
  document_number : STRING;
  schematic_reference : schematic;
  component_list : LIST [1:] OF component_select;
UNIQUE
  document_number;
  schematic_reference;
WHERE
  NOT (product_assembly_definition IN component_list);
END_ENTITY;
```

Figure 2. Example of constraints on an entity from the PSCM Model

In addition to these constraints, which are applicable within an entity, there are constraints specified through rules. Rules are used to describe the relationships among the instances of entities. Figure 3 includes a rule named `product_item_and_version` that illustrates the nature of such constraints. This rule ensures that a `product_item` is only associated with a single `product_item_version`. The `product_item_and_version` rule is simple but carries important semantics about the relationship between the entities `product_item` and `product_item_version`.

```plaintext
RULE product_item_and_version FOR (product_item, product_item_version);
  IF (Instantiation(product_item_version, product_item) <> 1) THEN VIOLATION;
  (* A PRODUCT_ITEM_VERSION is associated with one PRODUCT_ITEM *)
END_IF;
END_RULE;
```

Figure 3. Example of a rule from the PSCM Model

3.2 C++ as an Implementation Language

C++ was chosen as the language of implementation for the STEP Class Library for several reasons. First, a language that supports the object paradigm [Cox87][Kim89][Meyer88] was desired. The STEP conceptual models are hierarchical, and the concept of inheritance is fundamental to their organization. Also, a language that would be able to handle large and complex data files or databases without paying a large performance penalty was needed. Another consideration was the need for an implementation language that was compatible with a wide variety of software packages (compilers, databases, debugging tools,
compatible graphics packages, etc.). Finally, it was desirable to use a standardized language or at least a language that had the promise of becoming a standard soon\(^1\). Portability and modularity are necessities when trying to implement a standard for data exchange.

4.0 The STEP Class Library Architecture

SCL is a collection of several component libraries. The \textit{STEP Schema Class Library} holds container classes that are directly mapped from the STEP models. The \textit{STEP Core Class Library} provides the context-independent data access mechanisms for the STEP data and the mechanisms for capturing semantic information in the STEP schemas. The \textit{STEP Data Probe Class Library} supports the context-independent browsing and transport control to the STEP Schema Class Library. These three low level libraries form the foundation for the future \textit{STEP Data Access Interface Specification} (SDAIS) library. SDAIS\cite{Briggs90} will provide a uniform interface for applications to create, retrieve, and manipulate STEP data. The \textit{STEP Data Probe Class Library} and the SDAIS are in the design stage and are not covered in this paper.

4.1 STEP Schema Class Library: Translation of the Conceptual Model

The \textit{STEP Schema Class Library} is the set of files that result from the translation of a STEP schema. These files are generated automatically using the \textit{Fed-X Toolkit} \cite{Clark90} for translating Express and are producible from an Express schema. The program \textit{fedex_plus}, which is a backend to \textit{Fed-X}, takes a conceptual data model written in Express as input and generates three C++ files for each schema. The C++ code in these files provides the class definitions and member functions for STEP entities needed by an application program.

4.1.1 Schemas

When a \textit{STEP Schema Class Library} is generated from an input file of Express text, each schema of the Express conceptual data model generates the following files: a header file of class definitions, a library file of class functions, and an initialization file. The classes defined in the header file correspond to the entities and types defined in that schema. Presumably, any application using any one of these entities will need to use several of the entities in the schema; therefore, they are all placed in the same file. If the schema includes any other schemas, the header files for the other schemas are included, using a \texttt{#include} statement, in the owning schema’s header file. The initialization file contains a function that must be called to initialize a program to use the particular schema. The files are named after the schema that they represent.

---

\(^1\) The ANSI X3J16 committee on “C++ Programming Language” was formed in December 1989. An initial draft based on the AT&T C++ Reference Manual is currently under review by the committee.
Express schemas are not completely self-contained. Figure 4 shows the schemas defined in the original draft of STEP. In order to create a global schema, references to entities from external schemas must be resolved. To implement this in C++ the header files that represent the external schemas containing the referenced entities are included in the schema’s header file. An application based on SCL includes the header files for the schemas needed by an application and link in the corresponding archive files. The SCL file structure ensures that the references to the schema are resolved.

<table>
<thead>
<tr>
<th>resources</th>
<th>life_cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>applications</td>
<td>geometry</td>
</tr>
<tr>
<td>topology</td>
<td>shape</td>
</tr>
<tr>
<td>design_shape</td>
<td>nominal_shape</td>
</tr>
<tr>
<td>solids</td>
<td>shape_interface</td>
</tr>
<tr>
<td>features</td>
<td>tolerances</td>
</tr>
<tr>
<td>material</td>
<td>presentation</td>
</tr>
<tr>
<td>product_manifestation</td>
<td>drafting</td>
</tr>
<tr>
<td>mechanical_product</td>
<td>pscm</td>
</tr>
<tr>
<td>aec</td>
<td>aec_core</td>
</tr>
<tr>
<td>ship_structure</td>
<td>electrical</td>
</tr>
<tr>
<td>electrical_functional</td>
<td>electrical_tic_schematic</td>
</tr>
<tr>
<td>lep</td>
<td>analysis</td>
</tr>
<tr>
<td>fem</td>
<td>data_transfer</td>
</tr>
</tbody>
</table>

Figure 4. Schemas from STEP draft

4.1.2 Entities

Every entity defined in an Express file is mapped into a corresponding class in C++. The supertype/subtype relationship of Express also maps into the base class/derived class relationship of C++. Express entity names are not case sensitive. To ensure that the names are consistently translated into C++ classes, the following rules are applied in the translation.

1. All characters in a name are translated to lower case.
2. Then the first letter in the name and any letter immediately following an underscore character "_" are made upper case.

The assumption is made that the Express schemas are logically divided so that no naming conflicts between schemas will arise in application software developed using these libraries. Figure 5 shows the header file of C++ class definitions that is created when fedex_plus uses the Express code from Figure 1 as input.
```cpp
#include "definedtypes.h"
#include "STEPentity.h"

class Geometry : public STEPentity {
  protected:
    STEPentity_*_local_coordinate_system; // OPTIONAL
    STEPentity_*_axis; // OPTIONAL
  public:
    Geometry();
    ~Geometry();
    char*Name() { return "Geometry"; }
    int opcode() { return 1; }
    class Coordinate_System*_local_coordinate_system()
      { return (class Coordinate_System*)_local_coordinate_system; }
    void local_coordinate_system(class Coordinate_System*x)
      { _local_coordinate_system = (STEPentity*)x; }
    class Transformation* axis() { return (class Transformation*)_axis; }
    void axis(class Transformation*x) { _axis = (STEPentity*)x; }
};

class Curve : public Geometry {
  protected:
  public:
    Curve();
    ~Curve();
    char*Name() { return "Curve"; }
    int opcode() { return 4; }
};

class Line : public Curve {
  protected:
    STEPentity_*_pnt;
    STEPentity_*_dir;
  public:
    Line();
    ~Line();
    char*Name() { return "Line"; }
    int opcode() { return 24; }
    class Cartesian_Point*pnt() { return (class Cartesian_Point*)_pnt; }
    void pnt(class Cartesian_Point*x) { _pnt = (STEPentity*)x; }
    class Direction* dir() { return (class Direction*)_dir; }
    void dir(class Direction*x) { _dir = (STEPentity*)x; }
};

class Cartesian_Point : public Point {
  protected:
    real _x_coordinate;
    real _y_coordinate;
    real _z_coordinate; // OPTIONAL
  public:
    Cartesian_Point();
    ~Cartesian_Point();
    char*Name() { return "Cartesian_Point"; }
    int opcode() { return 10; }
    real x_coordinate() { return (real)_x_coordinate; }
    void x_coordinate(real x) { _x_coordinate = x; }
    real y_coordinate() { return (real)_y_coordinate; }
    void y_coordinate(real x) { _y_coordinate = x; }
    real z_coordinate() { return (real)_z_coordinate; }
    void z_coordinate(real x) { _z_coordinate = x; }
};

Figure 5. Example of entities from Geometry schema translated into C++
```
4.1.3 Attributes

All Express attributes are implemented as C++ classes regardless of the data type of the attribute. Built-in Express data types are represented directly in the corresponding entity class; otherwise the attribute is implemented as a pointer to the appropriate C++ class.

4.1.3.1 Explicit Attributes

For each explicit attribute in an Express model there is a corresponding data member in the protected section of the C++ class definition. In addition, there are a pair of access functions for each data member: one for assignment and the other for retrieval of the data. In an attempt to isolate applications from changes to the conceptual model a data member is assigned or retrieved through access functions rather than being assigned or retrieved directly. This eliminates the maintenance problem that occurs when an application's software assigns or retrieves an attribute directly. Using an access function eliminates the need to update every reference to the data member in the software when the implementation details of the data member are altered. This approach increases schema independence and facilitates modularity of the software. From the example in Figure 5, the access functions

```cpp
real x_coordinate() { return (real) ._x_coordinate; }

void x_coordinate (real x) { _x_coordinate = x; }
```

are defined for the `x_coordinate` attribute of the `Cartesian_Point` entity. Similar functions are defined for the `y_coordinate` and `z_coordinate` attributes.

4.1.3.2 Inherited Attributes

Inheritance of attributes in Express resembles the inheritance supported by C++. For the time being it is sufficient to represent inherited attributes through the standard C++ mechanisms; however, translating the other types of inheritance defined by Express into C++ is also being investigated. This issue is addressed later in this paper. In Figure 5 the access functions

```cpp
class Transformation* axis() { return (class Transformation*)_axis; }

void axis (class Transformation* x) { _axis = (STEPentity*)x; }
```

are inherited down into the `Cartesian_Point` class as inherited attributes.

4.1.3.3 Derived Attributes

Derived attributes are implemented as member functions.
4.2 STEP Core Class Library: Context Independent Classes

The STEP Core Class Library (SCCL) is a collection of context independent class definitions used by the schema dependant classes that are found in the STEP Schema Class Library. Classes found in the SCCL include a common base class for all STEP entity class definitions and classes to maintain meta information from the schemas. After a brief description of some problems solved by the SCCL, this section will conclude with definitions of the major classes in this library.

A problem with any translation of a conceptual model into an implementation language is in the translation of the semantics conveyed by the conceptual model. The symbolic names used in a model store some of the meaning intended by the modelers. Consider the following type definition.

```
TYPE inches = INTEGER;
END_TYPE;
```

To a human reading a conceptual model, the term *inches* conveys more information than the term *integer*. At first glance, it may seem as if inches can simply be translated to an integer and all would be well; however, this approach loses the semantics captured by the term inches. Furthermore, many of the tools that are being developed explicitly require that the symbolic information be available.

To capture the symbolic information several classes have been created. The `STEPentity` class captures information pertaining to the entities of the conceptual model; the `STEPattribute` class handles the descriptions of the entity's attributes. The `STEPenumeration` class currently stores the symbolic name of enumerated values. A class to retain the definitions `types` specified in the conceptual data models is the subject of future work.

4.2.1 The STEPentity Class

Meta information for every Express entity is stored in the base class `STEPentity`. This class is the root of each tree of classes corresponding to the entities in the Express conceptual data model. The data members and member functions for the class `STEPentity` are:

<table>
<thead>
<tr>
<th>Data Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance_id</td>
<td>a global identifier represented as an integer assigned to each instance of an entity</td>
</tr>
<tr>
<td>STEPfile_id</td>
<td>an identifier assigned to an instance in the input STEP exchange file</td>
</tr>
<tr>
<td>reference_count</td>
<td>an integer referring to the number of references to a particular instance</td>
</tr>
<tr>
<td>application_marker</td>
<td>an integer reserved for use specific to an application</td>
</tr>
<tr>
<td>attributes</td>
<td>the list of pointers to the standard data members specified in the STEP conceptual data model</td>
</tr>
</tbody>
</table>
Member Functions | Description
--- | ---
Name | the virtual function that returns the entity name for an instance of a class.
opcode | the virtual function that returns an integer assigned to represent a STEP entity.
STEPwrite | prints out an entity using the STEP exchange protocol.
begSTEPwrite | prints out any unprinted entities referenced by the entity being printed.
STEPread | reads an input stream of data in the STEP exchange format and assigns the values to the data members of the class instance.

The exchange protocol implemented by `STEPwrite` and `STEPread` in the current version of SCL is defined in an ISO document [Alte88a]. The ISO TC184/SC4/WG1 working group, which is investigating mechanisms for sharing data between applications, developed the protocol. Eventually, the standard will provide implementation rules for common memory and shared databases, but for now the definition is limited to this single exchange format.

A key design goal for SCL was to isolate the implementation of the exchange protocol from the class definitions in the STEP Schema Class Library. By doing so, it is possible to change the exchange protocol without disturbing code that uses the STEP Schema Class Library. This also hides the details of the protocol from the application developers. The following details of the current protocol should never be seen directly by the developers. It has been included for reference purposes only.

The exchange file has a syntactic format based on an Express schema. The file is a series of sets of data values. The format of the data sets is based directly on the entity definitions of the corresponding conceptual data model. Figure 6 shows an example of the file exchange format. The numbers preceded by the symbol @ are instance identifiers, which are assigned to the data member `STEPfile_id` by the member function `STEPread`.

```
STEP;
HEADER;
FILE_IDENTIFICATION(IBMTPRTZ,'1990 01 24 18 30 17','LMICKEE','COMPANY 3','1','1','PDES');
FILE_DESCRIPTION('SIMPLE PART');
IMP_LEVEL(USER DEFINED ENTITIES ONLY);
ENDSEC;
DATA;

@19=DIRECTION(0.7071067845031212,0.7071067845031212,0.7071067845031212,0.);
@20=DIRECTION(-0.7071067845031212,0.7071067845031212,0.7071067845031212,0.);
@21=DIRECTION(0.0,0.0,0.0,1.00000000000000000000000000000000);
@22-CARTESIANPOINT(0.0625,21.3794594354248047,11.5299987239711914);
@23-TRANSFORMATION(D19,M21,D22);
@24-COORDINATE_SYSTEM(,0.0,0.0,21.3794594354248047,11.5299987239711914);
```

Figure 6. Excerpt from a STEP exchange file based on the Geometry model
An issue of concern in this implementation was whether to make the \texttt{STEPentity} class a virtual or base parent of the schema classes. Currently the \texttt{STEPentity} class is implemented as a root node of each entity hierarchy. However, a schema is not required to be a true hierarchy; therefore, it would be more general to implement the class as a virtual parent of each \texttt{STEP} class. There are several reasons why this implementation was not chosen. First, it is less generic in the sense that the ability to cast a pointer to a virtual parent to a pointer to the appropriate class is not built in to C++. Second, virtual classes were first supported in C++ version 2.0 which supports multiple inheritance. Version 1.2 of C++ does not support this, and several of the database systems being considered for incorporation into the software currently run only with Version 1.2. The problem of a non-hierarchical entity structure has not been a concern with the models used to date.

### 4.2.2 \texttt{STEPAttribute}

The data member \texttt{attributes} of the class \texttt{STEPentity} is a list of pointers to data members that are \texttt{STEP} attributes. These pointers are represented by the class \texttt{STEPattribute}. A \texttt{STEPattribute} object contains the following data members:

<table>
<thead>
<tr>
<th>Data Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{shared}</td>
<td>a Boolean value indicating whether the attribute is also usable by another instance of a \texttt{STEPentity}</td>
</tr>
<tr>
<td>\texttt{nullable}</td>
<td>a Boolean value indicating whether the attribute needs to be populated for the instance to be a valid \texttt{STEP} instance</td>
</tr>
<tr>
<td>\texttt{type}</td>
<td>an enumerated value that indicates the data type of the attribute</td>
</tr>
<tr>
<td>\texttt{name}</td>
<td>the name of the attribute as specified in the \texttt{STEP} conceptual data model</td>
</tr>
<tr>
<td>\texttt{ptr}</td>
<td>the pointer to the data member representing the attribute</td>
</tr>
<tr>
<td>\texttt{type_name}</td>
<td>name of a type as defined in the Express conceptual data model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{aread}</td>
<td>reads in an attribute from an istream in the format specified by the \texttt{STEP} exchange file</td>
</tr>
<tr>
<td>\texttt{screen_read}</td>
<td>reads in an attribute’s value from standard input</td>
</tr>
<tr>
<td>\texttt{aprint}</td>
<td>prints out an attribute in \texttt{STEP} exchange format</td>
</tr>
</tbody>
</table>

### 4.2.3 \texttt{STEPAttributeList}

The \texttt{STEPattributeList} class is the key to providing common functionality for members of the \texttt{STEPentity} class. A \texttt{STEPattributeList} is a list of \texttt{STEPattributes}. The list can be used to traverse the \texttt{STEP} data members of any entity instance. For example, \texttt{STEPwrite} can traverse the \texttt{STEPattributeList} for any \texttt{STEPentity} and print the value of each element in the list in the format of the \texttt{STEP} exchange file.
The \textit{STEPwrite} function is only defined in one place rather than being redefined for each entity. The \textit{STEPattributeList} could also be used to print the names of the attributes.

4.2.4 \textit{STEP}enumeration

Enumerated data types are handled by the base class \textit{STEP}enumeration. This class maintains a list of the symbolic values that an enumeration is able to assume. An enumeration has two constructors: one accepts an integer value and the other takes a character string, which is the enumeration's symbolic value. The constructors check to make sure that the given value is in the specified domain. It may seem that an enumerated type in the conceptual data model should be directly translated to an enumerated type in C++; however, this translation loses the semantics of the enumerated type. The symbolic values of the enumeration are both necessary and desirable. These values are needed to interpret a \textit{STEP} exchange file in which enumerated values are represented by their symbolic names. When building interactive data editors that prompt the user for values, it is desirable to have such information available.

5.0 Example Applications

Several prototype applications have been started using SCL and several others are in the planning stage. The following sections describe a few of these first attempts at using the library to implement \textit{STEP}.

5.1 \textit{STEP} Data Probe

The \textit{STEP} Data Probe is being developed in the National PDES Testbed to support validation testing of the \textit{STEP} model. Test data will be created, viewed, changed, and deleted using the \textit{STEP} Data Probe. The probe can also be used to browse the \textit{STEP} schemas.

In some cases, using the probe directly will have limited practicality, such as when trying to locate a specific point or edge in a large mechanical structure. In these circumstances the probe will be integrated into higher level viewing aids, such as a solid model browser. In this configuration, the Data Probe will be given a reference from the graphic browser when an edge or a corner is selected. The probe will present a view of the \textit{STEP} data that corresponds to the item that was graphically selected. In this role the probe will provide a practical and consistent mechanism for viewing \textit{STEP} data. To incorporate the \textit{STEP} Data Probe functionality into an application, the developer will link in the \textit{STEP} Data Probe Class Library (see Section 4.0).

5.2 IGES to \textit{STEP} Translator

Many existing CAD systems currently support IGES and not \textit{STEP}; therefore, it was useful to create a translator between the two specifications. The resulting tool provides a convenient mechanism for generating data for testing the \textit{STEP} conceptual data models.
The translator includes a parser for the IGES file format and classes that represent IGES entities. When the IGES file is parsed, the data are placed into a list of instances of the IGES classes. Each IGES class has a member function called `makepdes` that translates the IGES entity into the corresponding STEP entities.

Figure 7 shows the definition of a class representing the IGES construct `LineEntity`. Figure 8 shows the definition of the `makepdes` virtual member function that does the translation into the corresponding STEP objects. The code that cycles through an IGES file and builds the STEP file uses a `linked list` class, a `string` class, fewer than 100 lines of additional code in parser routines. The function `translate`, shown in Figure 9, uses the virtual function `makepdes` to cycle through the various IGES objects and translate them into STEP objects.

```cpp
class LineEntity : public DeNode {
  public:
    real x1;
    real y1;
    real z1;
    real x2;
    real y2;
    real z2;

    void populateparameters(String);
    void makepdes();
};

void LineEntity:: makepdes(){
  PolyLine *line = new PolyLineO;
  Transformation *local_transformation;
  if (this->transformation_matrix == 0) local_transformation = NULL;
  else
    local_transformation = ((TransformationMatrixEntity*)
      mylist(this->transformation_matrix))->PdesTransformation;
  line->axis = local_transformation;
  CartesianPoint *point = new CartesianPointO;
  point->x_coordinate = x1;
  point->y_coordinate = y1;
  point->z_coordinate = z1;
  line->points.push(point);
  pdeslist.push(point);
  point = new CartesianPointO;
  point->x_coordinate = x2;
  point->y_coordinate = y2;
  point->z_coordinate = z2;
  line->points.push(point);
  pdeslist.push(point);
  pdeslist.push(line);
}
```

Figure 7. Example of the C++ representation of an IGES entity

Figure 8. Example of `makepdes` member function for an IGES entity
void translate()
{
    int index;
    DeNode *entry = mylist.First();
    while (mylist.AtEnd() != 1) {
        index = entry->sequence_number;
        entry->makepdes();
        if (index != entry->sequence_number) entry = mylist.Index(index);
        entry = mylist.Next();
    }
}

Figure 9. Function translate from the IGES to STEP translator

5.3 Databases

STEP provides the definition of product information, but it does not define how the data are stored. Some applications will rely on exchange files, others may require relational databases, and yet other STEP applications may use an object-oriented database.

The implementation of a database is not the target of the standardization effort. Within the Testbed, however, the conceptual data modeling and validation activities need access to a data storage and query capability. Consequently, storage tools have been investigated. At present, a limited SQL-based implementation of a STEP database has been developed. The SQL statements to generate the database tables were directly translated from Express using the Fed-X parser.

An object-oriented database implementation of STEP is being considered for the Testbed, but this effort is currently limited by the lack of an object-oriented equivalent of the SQL standard. There is interest in STEP among object-oriented database vendors, and several vendors are investigating approaches to implementing STEP in their databases.

5.4 Process Planning Tool

The STEP model has incorporated many of the features of a process planning language called ALPS (A Language for Process Specification) [Catron&Ray90], which has been developed at NIST. An Express language based conceptual model for ALPS has been translated into C++ using fedex_plus and the resulting classes are being used with SCL as part of a C++ implementation of a shop floor controller. Future plans for this project include extending the exchange mechanism of SCL to store the process work plans in an object-oriented database.
6.0 Summary

The existing version of SCL only partially meets the needs of a class library for implementing or testing STEP. Several of the significant tasks that will be addressed in the future development of SCL are examined below.

SCL has not addressed enforcement of some of the constraints on STEP entities. Without enforcement of these constraints, the semantics of STEP will not be fully enforced by the library. The most difficult challenge in constraint checking will be in implementing the rules and where clauses. The current plans for implementation call for a virtual member function, named \texttt{STEPvalidate}, in the \texttt{STEPentity} class. For each entity class in the STEP Schema Class Library a specialized version of \texttt{STEPvalidate} would be generated to enforce the constraints of the entity.

The mapping of inclusive subtypes from Express into C++ must also be resolved. While the C++ inheritance model is sufficient for representing generalization relationships [Smith&Smith77] where the subtype declarations are all mutually exclusive, it does not directly support the type of inheritance representation found in the inclusive subtype construct. Another method must be developed for dealing with this construct.

Functions and procedures from the Express language will be mapped into C++ in a future release of SCL.

The existing implementation of SCL only provides an exchange protocol for STEP data. Future releases may incorporate a database system, an SDAIS class implementation, and a shared memory implementation.

In the long term it would be helpful to further develop the STEP Schema Classes themselves by adding functionality which, although useful and necessary to an application, is not directly derivable from a specification written in Express. For example, a \textit{move} function could be added to the Geometry class to relocate it in space. This function would be generally applicable to any object of this type.

7.0 Conclusion

An alpha version of the STEP Class Library has been used in several prototype applications. This version meets many of the interface requirements of the draft specification of STEP. The results of this proof-of-concept development effort demonstrates that C++ provides an effective implementation mechanism for STEP and that the library mechanism developed for this project provides a manageable development tool. The project will proceed with plans to use SCL as the basis for tools in the Testbed, application prototypes, and a reference implementation.
References

[Bloom89] Bloom, H. The Role of the National Institute of Standards and Technology as it Relates to Product Data Driven Engineering, NISTIR 89-4097, National Institute of Standards and Technology, Gaithersburg, MD, April 1989.
[Fowler90] Fowler, J., STEP Production Cell, NISTIR, National Institute of Standards and Technology, Gaithersburg, MD, forthcoming.


This paper describes a C++ class library that implements the Standard for the Exchange of Product Model Data (STEP). The STEP class Library (SCL) is under development at the National Institute of Standards and Technology as part of the National PDES Testbed and provide the core set of STEP functions for tools that are used to validate the STEP information models and for STEP based application prototypes. The library also lowers the economic barrier to developing STEP based applications. The current version of the library provides a STEP compliant file exchange mechanism, consequently, users of the class library will not have to create an input/output mechanism for STEP.

The paper provides an introduction to STEP and the specification language, Express, in which it is written. The supporting classes that provide the STEP compliant input/output mechanism and the mapping between the Express specification and the C++ classes are described. The paper concludes with descriptions of some applications that use the class library and a discussion of future directions for the class library.

C++; CAD; CALS; CAM; CAPP; Core Library; Data Exchange Standards; Express Language; IGES; National PDES Testbed; PDES; SCL