

National PDES Testbed  
Report Series

**Validation Testing  
System:  
Reusable Software  
Component Design**

Katherine C. Morris  
David Sauder  
Sandy Ressler



**NISTIR 4937**

## National PDES Testbed



# Validation Testing System: Reusable Software Component Design

**Katherine C. Morris**  
**David Sauder**  
**Sandy Ressler**

U.S. DEPARTMENT OF  
COMMERCE

Barbara H. Franklin,  
Secretary of Commerce

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

October 1992



---

## Contents

---

<b>1</b>	<b>Introduction .....</b>	<b>5</b>
<b>2</b>	<b>Overview .....</b>	<b>7</b>
<b>3</b>	<b>VTs Specific / Model Independent Layer .....</b>	<b>9</b>
3.1	STEP Class Library .....	9
3.2	Data Editor Library.....	17
3.3	VTs Interface Library.....	21
<b>4</b>	<b>Application Model Specific Layer.....</b>	<b>25</b>
<b>5</b>	<b>Data Probe: An Example Application.....</b>	<b>26</b>
<b>6</b>	<b>Conclusions, Summary and Future Directions .....</b>	<b>30</b>
<b>7</b>	<b>References.....</b>	<b>31</b>
<b>Appendix A</b>	<b>EXPRESS Model of Registry Classes .....</b>	<b>33</b>
<b>Appendix B</b>	<b>VTs Document Series.....</b>	<b>37</b>



---

# Validation Testing System: Reusable Software Component Design

---

---

## Abstract

---

Data sharing is a difficult problem with a variety of issues. There is a need to share data across multiple enterprises, different hardware platforms, different data storage paradigms, and a variety of network architectures. The ISO Standard for the Exchange of Product Model Data (STEP) addresses this need by providing information models which clearly and unambiguously describe data. The validity of these information models is essential for success in sharing data in a highly automated business environment.

The design of software, which supports the testing of information models for validity and correctness, is described in this document. This design follows requirements and an architecture described in previously published Validation Testing System (VTS) project documents. The collection of these documents provides a basis for software development within the National PDES Testbed. The Testbed is used to validate information models for STEP. The scope of this document is limited to the design of those components of VTS software scheduled for development in the initial phase of the project.

---

## 1 Introduction

---

The software described in this document supports the Validation Testing System (VTS) at the National PDES Testbed<sup>1</sup>. The Testbed is used by researchers to test the validity of application protocols, or application models<sup>2</sup>, which are being proposed for STEP<sup>3</sup>.

---

1. Funding for this work and the Testbed, located at NIST, has been provided by the Department of Defense's Computer-Aided Acquisition and Logistic Support (CALS) Office. PDES, Product Data Exchange using STEP, is the U.S. activity in support of the international STEP standard.

2. The term *application model* will be used throughout this paper to refer to the domain specific schema which is being evaluated whether that be an application protocol (AP), an application resource model (ARM), a context driven integrated model (CDIM), or some other form of an application schema.

3. The Standard for the Exchange of Product Model Data (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).

The testing process is described in a paper outlining the methodology for Application Protocol validation [Mitch91].

The design architecture [Morris92] provides a framework for the software developed within the VTS project. The software described in this document will be designed and implemented incrementally as resources are available. Likewise, this document will be updated regularly as the design of the various software components is completed. For a complete description of the VTS document series please see Appendix B<sup>4</sup>.

The initial emphasis of the VTS software is the support of Test Data Generation software, more commonly known as the Data Probe. Therefore, the software components detailed in this document directly support the requirements for Test Data Generation.

The audience for this document includes software designers and developers. This document builds on the software architecture covered in the VTS architecture document [Morris92]. This approach is relevant to those software developers concerned with the development of STEP-based schema driven software for production oriented environments.

### A Note on Document Style

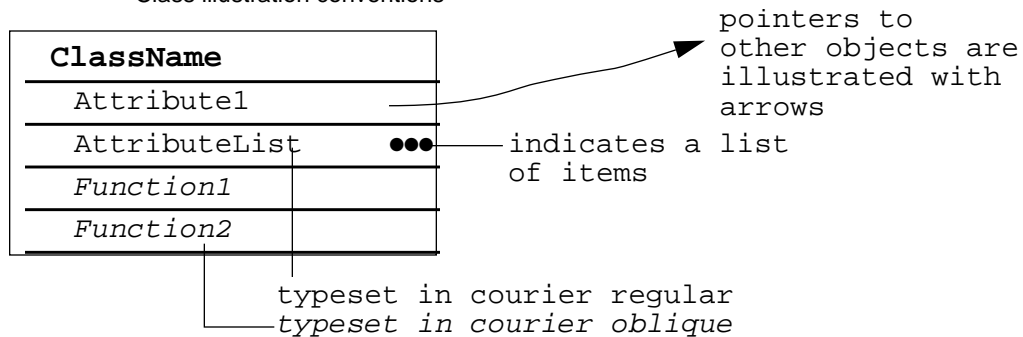
Throughout this document we have used a number of typographic practices to improve clarity, as indicated in FIGURE 1. All references to the names of software modules, and libraries are in *italics*, classes and specific fragments of code (EXPRESS, C or C++) are set in `courier`. The actual names of attributes and functions in these diagrams do not necessarily correspond to existing code. This is not a software reference manual, and names are meant to be descriptive in nature.

Class diagrams are drawn with the following convention:

---

**FIGURE 1**

Class illustration conventions



---

4. No approval or endorsement of any product by the National Institute of Standards and Technology is intended or implied. The work described is funded by the United States Government and is not subject to copyright.

---

## **2 Overview**

---

The VTS software architecture integrates modular software libraries. These libraries will be incorporated into a single system which supports functions needed for the validation process [Mitch91]. The use of object-oriented techniques and standard interfaces will enable software reusability. The system will use software as available from external sources. When such software is unavailable, the necessary software will be developed. Support for the implementation methods specific to STEP will need to be developed.

Since STEP is a developing standard, the mechanisms for its implementation are not stabilized. Furthermore, since these mechanisms are developing concurrently with the application models which the VTS software is used to validate, it is unlikely that externally developed software will be available in the necessary time frame. These mechanisms include the specification language for the application models -- EXPRESS [ISO11] -- and the data interface formats, such as the exchange file format [ISO21]. The VTS must be responsive to changes in these implementation mechanisms by providing software which is quickly and easily adaptable to new versions of the mechanisms.

The following goals have influenced the design of the VTS software architecture:

- minimize the impact of changes to the standard on implementation software
- permit easy transition of the software to support a new application model,
- minimize the need for data translation by providing an integrated system which supports a broad range of functions,
- provide a single end-user program,
- enable the development of different style user interfaces,
- allow for the integration of externally developed software into the system, and
- develop reusable software.

All the software components needed to support the testing process involve the computerized manipulation of data based on a common schema or application model. Therefore, the approach taken has been to develop a library of data structures and access functions for representing a schema which can be shared among the different components. These data structures must be rich enough to store data and provide some semantic information from the application models at runtime. Semantic information needed at runtime includes the names of the structures, their fields, and, when possible, acceptable values for the fields and rules governing instance structure. In the initial implementation the latter information is available only by displaying the application model, as specified in EXPRESS, to the user.

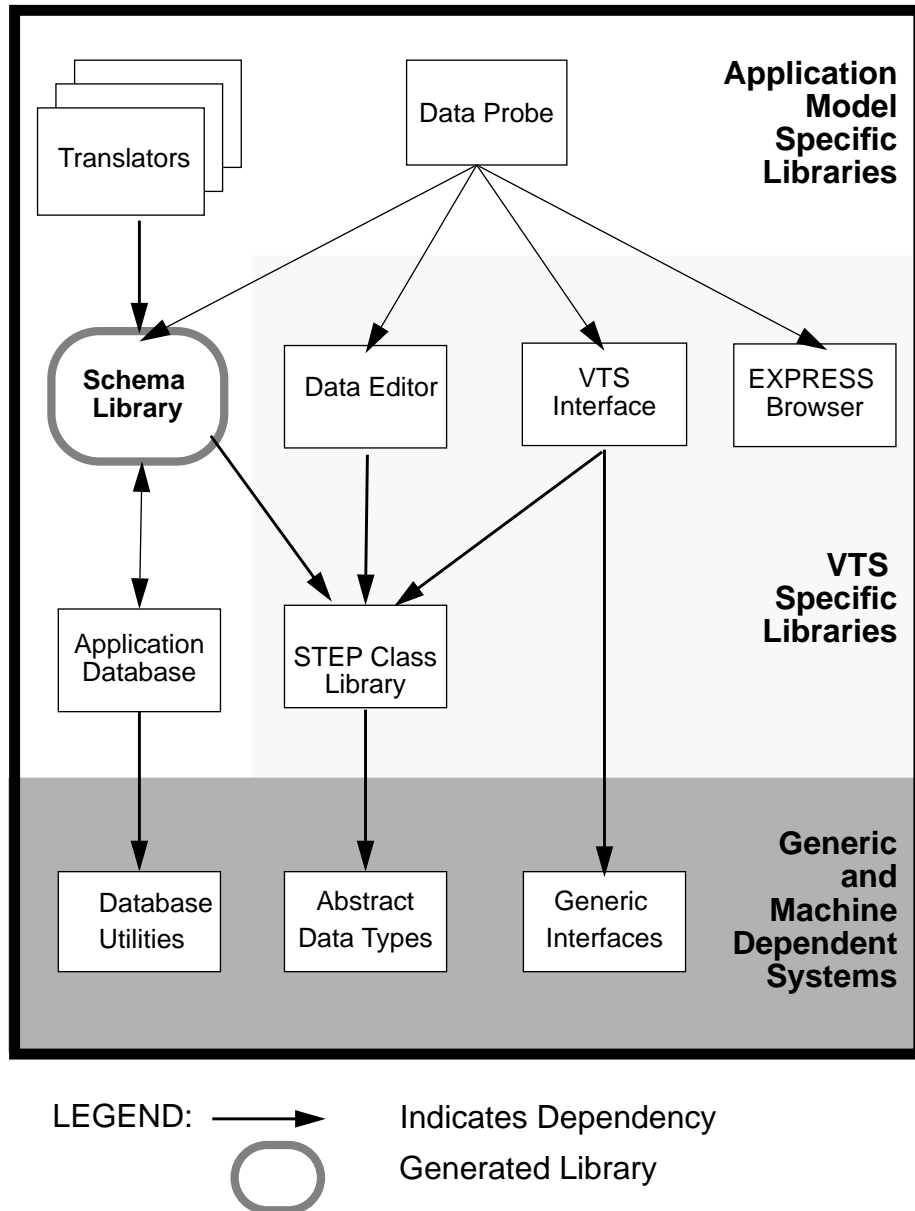
### **Relationships between the VTS Software Components**

This section discusses relationships between the VTS software layers as specified in the VTS architecture document [Morris92]. The application model specific libraries, the VTS specific libraries, and the generic systems each contain functional elements of this software. The VTS software design is confined initially to the components in these layers. Only the first 2 layers are discussed here, since software in the "generic" layer is

expected to come from outside (i.e. commercial or public domain) sources. Dependencies between the software in these layers are shown in FIGURE 2.

FIGURE 2

VTS Software Component Relationships



The division of the VTS software into component libraries enables code reuse by encapsulating the functionality needed for different aspects of the validation process. Several considerations are influential in defining the components of the VTS software, but two functional requirements are of particular importance:



- the need to easily transition the software to a new application model, and
- the desire to enable different user interfaces to be developed.

The conceptual division of VTS software is based on these requirements. The information from a particular application model is encapsulated in the *Schema Class Library*. The *Data Editor Library* encapsulates the information and operations needed for manipulating and editing that information. The *VTS Interface Library* contains the operations needed to present that information to the user. The presentation format in the user interface is not influenced by the specific content of the information but uses generalizations, or *abstractions*, about the structure of the information. These abstractions are based on EXPRESS and are supported by the *STEP Class Library*.

The VTS architecture layering supports modularization of the software. Component libraries in each layer are dependent on some of the libraries in the next, more general layer. However, the more general layers are not dependent on the less general layers. For example, the *VTS specific* layer depends on the *generic systems* layer, but not vice versa. Dependencies are limited to the interface between layers. The exception is in the database management system. The database system will come from an external source and will have its own particular structure.

The VTS architecture is structured to enable reuse of the software. A different style of user interface could be developed for the VTS software by replacing the *VTS Interface Library*. This can be done without affecting the data editing and representation capabilities of the system. For example, two different systems could be developed by replacing this library. One could support window-based interfaces and the other ASCII-based interfaces. These systems would provide the same functions in terms of editing and representational capabilities, since those components of the software would not change.

Other programs may use parts of the VTS software. In particular, the *STEP Class Library* and *Schema Class* libraries may represent the application data structures. Stand-alone translators could be developed using these libraries. These translators would not be dependent on the *VTS Interface Library* which may require a sophisticated windowing system. But they would share the same data representation capabilities of the VTS software and would be able to directly access the same database using the interface provided in the *Schema Class Library*. The interface to the database would be transparent to the translators. Likewise, parts of the *VTS Interface* and *Data Editor* libraries could be used for any general purpose editor of highly structured information.

---

### **3 VTS Specific / Model Independent Layer**

---

Software in this layer is specific to the needs of validation testing but independent of any particular application model. These include the data structures needed to represent entities. In addition, the functionality for editing the data contained in these data structures is in this layer.

#### **3.1 STEP Class Library**

The *STEP Class Library* (SCL) is a collection of application independent class definitions. They are used by the application dependent classes found in the *Schema Class Library*. Classes found in the SCL include a common “base” class for all entity class

definitions and classes to maintain meta-information from the schemas. After a brief description of some problems solved by the *SCL*, this section will conclude with definitions of the major classes in this library.

The *STEP Class Library* (see FIGURE 3) provides functionality for supporting a *Schema Class Library*, a dictionary of the application model, and data files. The *SCL* is dependent on the external C++ libraries for standard input and output [McLay90].

The problem with any translation of a conceptual model into an implementation language is 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:

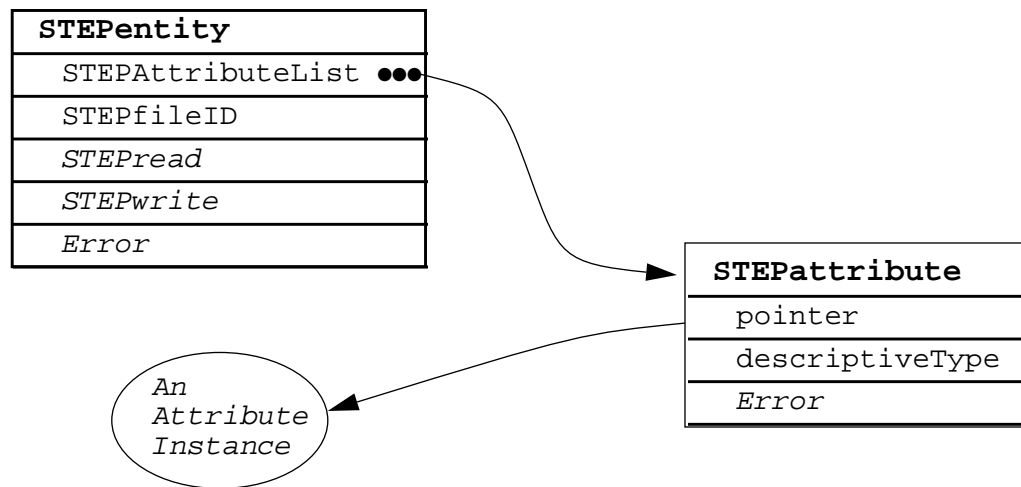
```
TYPE inches = INTEGER;
END_TYPE;
```

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 does not maintain the semantics captured by the term *inches*. *Inches* is a specific unit of measurement not simply a number.

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.

FIGURE 3

STEP Class Library - major classes and relationships



### 3.1.1 Class Relationships within STEP Class Library

The *STEPentity* class contains the functions *STEPread* and *STEPwrite* which encapsulate the behavior required to read and write an entity encoded in the STEP exchange file format.

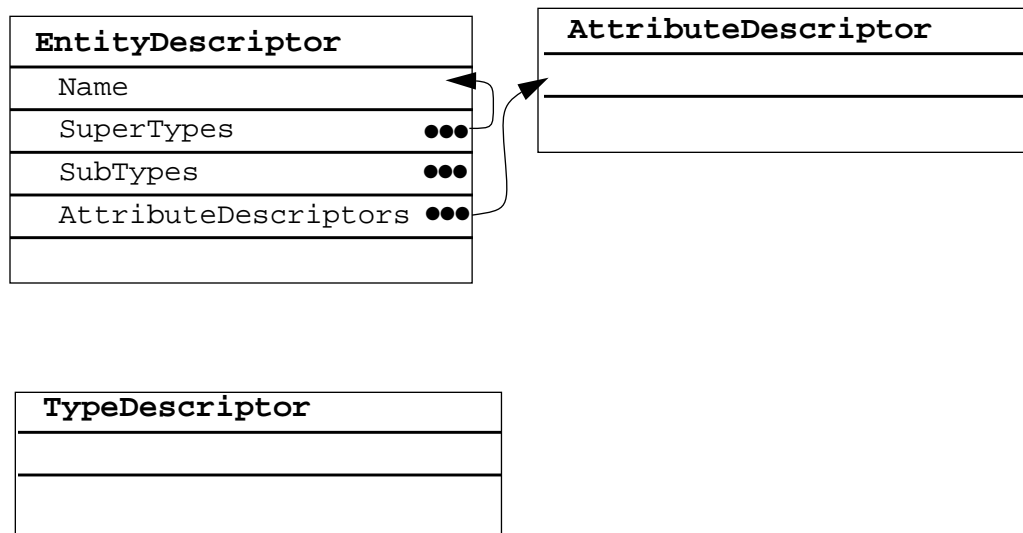
The `STEPAttributeList` is a list of pointers to the instances of the attributes of an entity. Functions can traverse the list of attributes without any knowledge of attribute types. Information in the `STEPAttribute` class describes the type of an attribute for a given instance of the attribute. Using this information the `STEPAttribute` can “validate” itself by ensuring that the data pointed to by the `STEPAttribute` corresponds to the correct type.

The descriptive information from a schema is encapsulated in a set of classes, which populates a `Registry` (see **FIGURE 4**). The `Registry` is used as a dictionary to the contents of the schema for the application model. The `Registry` is also used to create new instances of the `STEPEntity` class at run-time based on the entity’s symbolic name.

The objects contained in the `Registry` are instances of `Entity Descriptors` and `Attribute Descriptors`. These classes maintain information about an entity such as its name, the supertype(s) and subtypes, and the names of its attributes. The information in the `Registry` mirrors the particular schema being used.

**FIGURE 4**

Registry classes



### 3.1.2 Relationships to other Libraries

The `STEPEntity` class is the link to the other libraries within the VTS software. The *VTS Interface Library* provides the interactive user’s view of an instance of an entity. The view is dependent on the `STEPEntity` class to provide its structure.

Classes in the *Data Editor Library* manage sets of instances of the `STEPEntity` class. The *Data Editor Library* is unaware of the contents of the instances but is able to generically manipulate them. Likewise, the `STEPfile` class is able to generically operate on instances of the `STEPEntity` class, leaving the details of the file syntax to be handled by the `STEPEntity` and `STEPAttribute` classes.

The Registry is used by the *Data Editor Library*. The Registry provides functions for inserting, deleting, and retrieving entity descriptors. All of the entity descriptors in the Registry must be unique. To support this functionality the nodes in the Registry have a key. In EXPRESS a unique key is determined by the combination of the entity name and schema name, but in this implementation uniqueness is determined by entity name alone. Entity names may not be used in more than one schema.

### 3.1.3 Classes

The *STEP Class Library* consists of two parallel sets of classes (see TABLE 1): one dealing with instances and the other dealing with symbolic descriptions of the instances, the schema. Each set contains a class for representing entities and a class for representing attributes. The library also contains parallel sets of classes which are specialized for different types of attributes (i.e. aggregate attributes.) These specialized classes are not described in detail in this document. (A model written in EXPRESS describes all the classes for the Registry of EXPRESS. See the Appendix for details on this model.)

TABLE 1

Primary Classes of the STEP Class Library

instance	schema
STEPentity	EntityDescriptor
STEPattribute	AttributeDescriptor
	TypeDescriptor
STEPattributeList	Registry

The separation of sets of classes is guided by the multiple uses for the library. One category of functions that the library is intended to support involves interactive browsing of an application model; the other category of functions involves translations of data into a particular application model. For interactive tools the symbolic descriptions of the application model is needed. For translators a programming interface to the application model is sufficient. Many tools (i.e., a data editor), however, support combinations of these functions and, therefore, use both sets of classes.

#### STEPentity

The abstract base class, STEPentity, enables the generic manipulation of the specific classes which correspond to entities in the application model -- the classes generated (by the fedex\_plus utility) from an EXPRESS schema. The STEPentity class encapsulates the access to information about an instance of entity, including both values for its attributes and symbolic information describing the entity's type. The STEPentity class does not maintain this information directly. The values for the attributes are maintained by the appropriate subclass of the STEPentity. The symbolic information is provided by the Entity Descriptor. However, all this information can be accessed through the STEPentity class.

In addition, the `STEPentity` class provides the necessary functions for implementing the STEP file exchange protocol [ISO21] and initiating instance validation. In particular, the `STEPentity` class contains an instance identifier, error state information, and a list of its attributes. The identifier is used to maintain the instances's identifier for an exchange file. The error state is used to report invalid or missing data values. As the ISO standard develops, the `STEPentity` class may be used in the implementation of other data sharing features such as an identifier which has a broader context than an exchange file.

A key design goal for the *SCL* was to isolate the implementation of the exchange protocol from the generated class definitions for a particular application model. By doing so, it is possible to change the exchange protocol without modifying code that uses the *Schema Class Library*. This approach also serves to isolate the details of the exchange protocol from the users of the software.

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 application model. The fragment below shows an example of the file exchange format. The numbers preceded by the symbol @ are instance identifiers. The following fragment is from a STEP exchange file based on the Geometry model [ISO42] (this example is from a very early version of the file format):

```
STEP;
HEADER;
FILE_IDENTIFICATION('IBMPRT2','1990 01 24 18 30
17','L.MCKEE'),('COMPANY 3'),'1','1','PDES');
FILE_DESCRIPTION('SIMPLE PART');
IMP_LEVEL('USER DEFINED ENTITIES ONLY');
ENDSEC;
DATA;
.
.
@19=DIRECTION(,,0.7071067845031212,0.7071067845031212,0.);
@20=DIRECTION(,,-0.7071067845031212,0.7071067845031212,0.);
@21=DIRECTION(,,0.,0.,1.0000000308363815);
@22=CARTESIAN_POINT(,,0.0625,21.3794994354248047,11.5299997329711
914);
@23=TRANSFORMATION(,,#19,#20,#21,#22,);
@24=COORDINATE_SYSTEM(,,#23);
```

### **STEPattribute**

Like the `STEPentity` class the `STEPattribute` class encapsulates the access to information about values for an entity's attributes and symbolic information describing the attribute. The actual values for an entity's attributes are maintained by the appropriate class in the *Schema Library*; however, the `STEPattribute` class maintains a pointer to the value for an attribute. This feature is used to access an attribute's values without knowing the nature of the attribute. The descriptive information about the attribute is maintained within the `AttributeDescriptor` which is accessible

through the instance of the `STEPAttribute`. The `STEPAttribute` class maintains error state information for the attribute. The error state is accessed by the `STEPEntity` class during instance validation.

### **STEPAttributeList**

The `STEPAttributeList` class is the key to providing common functionality for subclasses of the `STEPEntity` class. The list can be used to traverse the attributes of any entity instance. Using the instances of the `STEPAttribute` both the values for and the descriptive information about the attribute can be accessed. For example, the list is traversed by the function which implements the exchange file format for reading and writing of the values for an instance. Similarly, the list is used to display the names of an entity's attributes to an interactive user.

### **Registry**

The purpose of the `Registry` is to provide access to the descriptive information about schemas, types and attributes. This information is used to present meaningful information, such as names, to users. In addition the `Registry` has a mechanism for the creation of instances based on the entity name. The traversal of the schema hierarchy via pointers to the parent and subtype entities is also facilitated by information in the `Registry`.

A `Registry` provides two important functions:

- it contains symbolic information about the structures (the entities and their attributes) described in the application model.
- it provides a mechanism to create new instances of an entity type given an entity's symbolic name.

The `Registry` used in the VTS software contains entries associated with an application model written in EXPRESS; however, the `Registry` class may alternatively be populated with information from models specified in a different data modeling language. To use the `Registry` for a different data modeling language, appropriate descriptor classes (analogous to the Entity and Attribute Descriptors described below) must be defined as subtypes of the `RegistryEntry` class.

Support for multiple schemas at runtime is one requirement which led to the design of the `Registry` class. The `Registry` class provides a mechanism to instantiate a program with a particular application model. In the design of the initial VTS software only one application model, and therefore one instance of a `Registry`, will be used. However, it is envisioned that multiple registries would be used to support multiple schemas in a single program. By doing so the user would be able to switch "contexts." In this way the application model visible to the user could be switched at runtime.

An aspect of the STEP application models, currently not addressed in the VTS software design, relates to the organization of the application models. The VTS software currently is not addressing these requirements because the organization of these models and the relationship of this organization to the EXPRESS language is in a very active stage of development. It is anticipated that a mechanism for supporting multiple, and possibly overlapping or nested, application models in a single program will be needed to

support the emerging organization. As these requirements are defined and become more stable, more support for these will need to be incorporated into the VTS software design. The `Registry` class encapsulates the mechanism to support future application model needs. This encapsulation will isolate the changes to the VTS software needed to support the new organization.

One problem area is entity referencing between schemas. Two EXPRESS language constructs in the interface specification portion of EXPRESS, `USE FROM` and `REFERENCE FROM`, allows schemas to use entities from other schemas. This is not handled in a robust way by the current version of *fedex\_plus* [McLay 90] or the VTS libraries. This area is in flux in the EXPRESS committee and other committees within the STEP community.

As an aside, the class descriptors used by the `Registry` were developed by using an early version of the Data Probe (see the section Data Probe: An Example Application). A model of EXPRESS in EXPRESS was created, translated to C++ and a Data Probe of this EXPRESS of EXPRESS schema was created. Data entered in this particular Data Probe was used to prototype the class information which we desired to represent in the `Registry`. This iterative design process has proven quite useful [Kramer92].

### **EntityDescriptor**

For each entity in the application model an entity descriptor is created. An entity descriptor encapsulates the minimum set of symbolic information needed by the VTS software at runtime. The entity descriptor is represented by the class `EntityDescriptor`. This class is used to populate an instance of the `Registry` class and is the `Registry` entry for an EXPRESS information model. Instances of this class contain the following information:

- name
- schema
- subtype(s)
- supertype(s)
- attribute(s)
- creation function

One entity descriptor is created for each entity in the application model. The entity descriptor is accessible in a number of ways.

- The entity descriptor is represented as a global variable in the Schema library.
- Those subclasses of the `STEPEntity` class which are instantiated contain a pointer to an `EntityDescriptor`. Each instance of a `STEPEntity` points to the entity descriptor for the entity type.
- When an entity descriptor is stored in a `Registry`, it is accessible from the `Registry` given the entity name. The entity name serves as the key into the `Registry`.

### **AttributeDescriptor**

A description of each attribute is also contained in the schema dictionary or Registry. These descriptions are most easily accessible through the entity descriptions which contain direct pointers to the descriptions of the attributes. The EXPRESS definition of the attribute descriptor is as follows:

```
ENTITY attribute_descriptor;  
    name: STRING;  
    type: type_descriptor;  
    optionality: BOOLEAN;  
    uniqueness: BOOLEAN;  
    owner: entity_descriptor;  
END_ENTITY;
```

This structure describes the basic properties associated with an attribute. In the initial implementation this information is used to present a simple description of the attribute to the user; in later implementations it may be used in conjunction with constraint checking on values of attributes.

The initial implementation of the VTS software does not provide support for derived attributes. The requirements which support derived attributes have not been identified for this design. However, it is anticipated that the *AttributeDescriptor* concept will be extended to support these attributes.

### **TypeDescriptor**

Much of the information which describes an attribute is contained in the attribute's type. To support this, another type of entity is introduced: the *type\_descriptor*. Due to the variety of types available in an EXPRESS schema, there are several subtypes of *type\_descriptor*. The following EXPRESS definitions represent the basic *type\_descriptor*:

```
TYPE base_type =  
ENUMERATION OF (  
    INTEGER_TYPE, STRING_TYPE, REAL_TYPE, ENTITY_TYPE,  
    AGGREGATE_TYPE, ENUM_TYPE, REAL_PTR_TYPE, INTEGER_PTR_TYPE,  
    SELECT_TYPE, BOOLEAN_TYPE, LOGICAL_TYPE, NUMBER_TYPE, BINARY_TYPE,  
    UNKNOWN_TYPE );  
END_TYPE;  
ENTITY type_descriptor;  
    name: STRING ;  
    fundamental_type: base_type;  
    description: STRING;  
END_ENTITY;
```

The name, *fundamental\_type* and *description* are currently implemented in the VTS software. The subtypes of the *type\_descriptor* are not implemented in the initial implementation of the VTS software. The level of detail provided by this design is not a priority for the VTS software. The information encapsulated in the *type\_descriptor* represents a minimal set of information which is needed at runtime.



The `type_descriptor` encapsulates the area which will be expanded for future implementations to support access to more robust type information. The expanded type information is particularly necessary for the software to evolve to support tighter domain restrictions for attribute values and other types of constraints.

### 3.2 Data Editor Library

One of the goals of the VTS software design is to provide a clean separation of the software's functionality and the interface. Support for editing is contained within the *Data Editor Library*. This library includes functions for editing individual instances and for manipulating groups of instances. Group manipulation includes merging exchange files, searching sets of instances, and checking sets of instances for completeness with respect to the application model. The *Data Editor Library* also maintains the state of an interactive editing session.

The *Data Editor Library* manages information from two areas. First, it is responsible for managing information about instances of `STEPentity` with regard to the editing process. This information is used as the subject material for the visual objects (views of the subject) that are shown to the user with the VTS user interface. (A visual object refers to an object in the *VTS Interface Library* which can actually be seen on the computer screen.) The information in the visual objects can then be viewed, edited and copied back to the underlying subject. The second area of information managed by the data editor is information about the display, such as:

- which `STEPentity` is currently being displayed,
- which of the `STEPentity` instances have been displayed,
- in what order `STEPentity` instances are displayed in the list of `STEPentity` instances read from an exchange file,
- in what order available entity types are displayed,
- how a `STEPentity` instance has been saved,
- which `STEPentity` instances are marked to be deleted,
- what is the state of a visual display of a `STEPentity` instance (is writable or read-only)

It is important to note that the visual objects are not actually created or placed on the screen by the *Data Editor Library*. The *Data Editor Library* maintains information about which `STEPentity` instances have an associated visual object and the state associated with those objects. The *Data Editor Library* is only loosely tied to a specific user interface library.

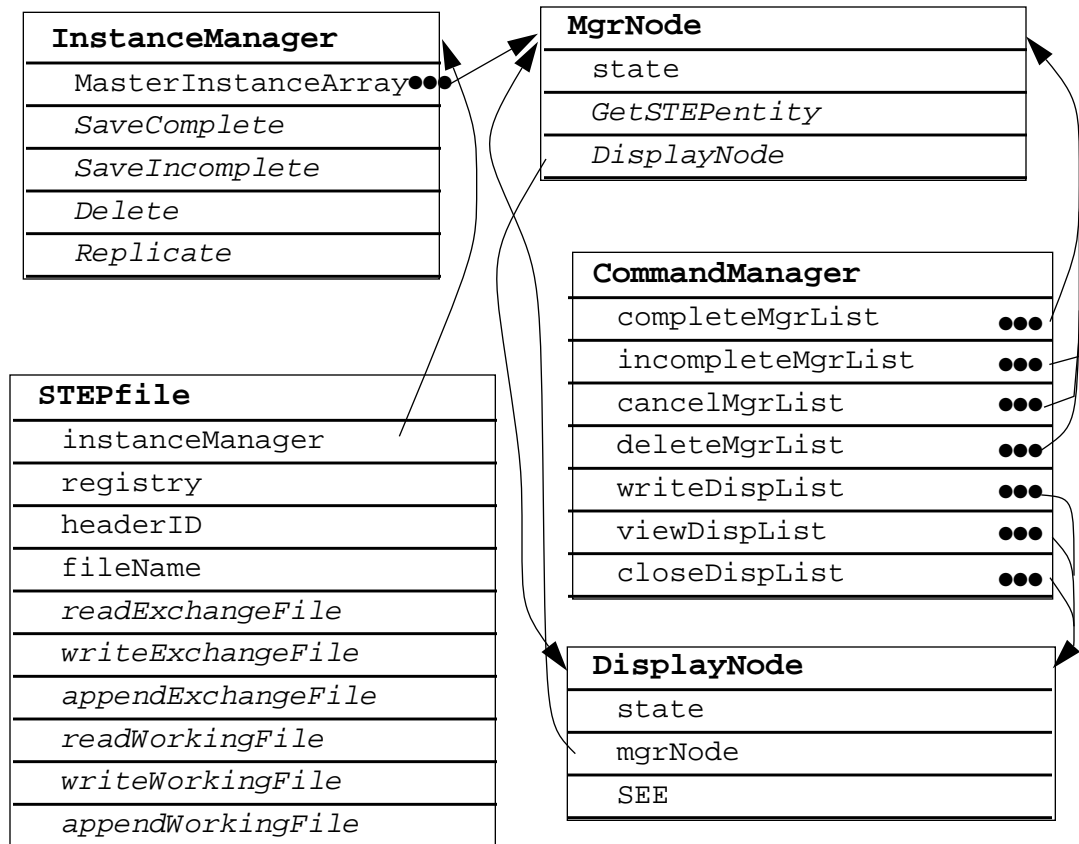
#### 3.2.1 Class Relationships within Data Editor Library

The primary classes in the *Data Editor Library* are the `InstanceManager`, the `ManagerNode`, and the `STEPfile` class. The library also relies on the `Registry` and `STEPentity` classes, which were described in the previous section. The prime use of the `InstanceManager` is to maintain a list of the entire set of instances of `STEPentity` for a particular editing session. The `InstanceManager` is also used

by the `STEPfile` class to maintain information about the header section associated with a particular file.

FIGURE 5

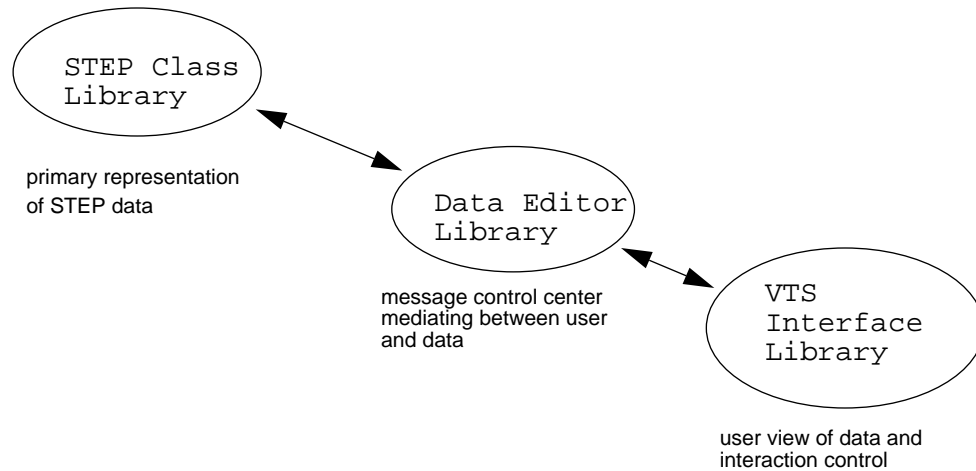
Data Editor classes



### 3.2.2 Relationships to other Libraries

Classes in the *Data Editor Library* work closely with the classes from the *STEP Class Library* and *VTS Interface Library* to implement a graphical editor for STEP entities. The *VTS Interface Library* defines visual classes. The *Data Editor Library* serves as a mediator between the *STEP Class Library* and the *VTS Interface Library* (FIGURE 6).

---

**FIGURE 6** Data Editor Library as Mediator


### 3.2.3 Classes

The classes in the *Data Editor Library* primarily perform management functions. They implement the control necessary to pass control messages through a variety of data structures.

---

**TABLE 2** Primary Classes of the Data Editor Library

---

InstanceManager	ManagerNode
STEPFile	

---

#### **InstanceManager**

The InstanceManger maintains a set of instances.

The primary purpose of the InstanceManager class is to keep a master list of instances in the current editing session. It also maintains an index into that set for fast look ups. Whenever a new instance is created during the editing session, either interactively or by reading in a data file, the new instance is added to the master Instance Manager.

The InstanceManager class also maintains other instances. For example, an Instance Manager instance is used by the STEPfile class to maintain the header information associated with a particular file. Future implementations may allow the user to indicate groupings of instances. Such a feature could be implemented using the InstanceManager class.

### **ManagerNode (MgrNode)**

The `ManagerNode` class maintains an association between an object of the `STEPEntity` class and a visual object in the user interface. The `ManagerNode` class allows the application model to remain independent of the *VTS Interface Library*. The `ManagerNode` maintains an editing state for a `STEPEntity` object and a state for the display object. The editing states are

- saved complete,
- saved incomplete,
- marked for deletion, and
- new

The *saved complete* state indicates that the instance has all necessary values (i.e. all non-optional attributes have values) and, as far as can be determined, these values are correct. The *saved incomplete* indicates that the instance does not have all its necessary values. This state can be set in two ways: 1) An error resulted when the validation function from the `STEPEntity` class was executed. 2) Or the user interactively set the state to “incomplete” through the user interface. The *new* state indicates that the instance has been recently created interactively and has not been edited.

The Manager Node maintains information about which instances have corresponding Display objects and the state of those Display objects. Available states for Display objects are

- editable display object
- view-only display object
- unmapped display object
- no display object

An *editable display object* is visible on the screen and may be edited by a user. A *view-only display object* is visible on the screen but may only be viewed by the user. An *unmapped display object* refers to a display object which has been created but is not currently visible on the screen. (An Unmapped display object is a mechanism for buffering the implementation of the display.)

In addition, the Manager Node also maintains *intended* states for the instances. Intended states are used to mark instances for attempted state changes which can then be applied at one time. For example, several instances can be marked for deletion or display, and messages to initiate the appropriate operations can be sent to the respective objects at one time. This functionality is intended to aid editing across the Internet or modems where display performance is a problem.

### **STEPFile**

The `STEPFile` class implements two important functions:

- controls the reading and writing of data files, and
- interfaces with the file system for opening and closing files.

The `STEPfile` class relies on the `STEPentity`, the `Registry`, and the `InstanceManager` classes for implementing the functions which it initiates.

For instance, for the reading and writing of data files the `STEPfile` class serves as a driver. The `STEPfile` class opens the file, controls the parsing of the sections of the file, initiates the creation of new `STEPentity` instances (the `Registry` class actually creates the instances), initiates the reading or writing of the instances (the `STEPentity` class actually parses the instances), updates the `InstanceManager` during the process, and finally closes the file. The parsing of the file involves two passes over the Data Section (to resolve forward references.) These passes are controlled by the `STEPfile` class. Throughout the process the `STEPfile` class monitors the error state and provides appropriate messages as errors are encountered.

The `STEPfile` class also implements functions to save the working state of an editing session. The state of the session is saved into a file similar in structure to an exchange file: a *working session file*. The working session file stores the state of the instances with the editing session as indicated by the `ManagerNode`. Unlike the exchange file, the working session file does not require `STEPentity` instances to be a complete and valid state (in other words, attribute values, required by the application model, can be missing).

### **CommandManager**

The `CommandManager` isolates and encapsulates editing functionality. Functionally the command manager contains commands which act upon the selected entities. For example, a set of entities which are to be saved in a complete state are maintained by the `CommandManager`. Other operations handled by the `CommandManager` include save incomplete, cancel, and delete. These commands are subsequently processed by the `Probe` class (defined in the VTS Interface Library, see FIGURE 7).

### **DisplayNode**

The `DisplayNode` is the class which contains display information for an entity. The `DisplayNode` encapsulates all display information. Furthermore `DisplayNodes` only exist for those entities which are actually displayed on the screen. This is a significant performance issue as we expect to have data files with thousands of entities. The display node points to the actual display object which could be replaced by other display objects in the future if a new user interface is chosen.

## **3.3 VTS Interface Library**

The *VTS Interface Library* was designed with the user in mind. Users are expected to be people familiar with the STEP file format [ISO 21], and the EXPRESS language [ISO 11]. Users interact primarily with three different types of windows. First the `EntityType` window (see FIGURE 10) which displays a list of all entity types for the schema. The other window is the `Entity Instance` window (see FIGURE 11) which displays entity instances. Display of these instances is in the STEP physical file format, a format familiar to the users. The third window type is the STEP Entity Editor (SEE) window used for the actual editing of data.

Each time the user creates a new instance or views an existing one a SEE window (see left side window of FIGURE 10) is created and displayed. In the context of the current implementation which uses the X windows system each SEE is created as a separate window which can be moved and iconified. This approach allows the user to interact with the windows without introducing any new unknown window management commands.

Where possible the key bindings used by the display and editable objects match those found in GNU emacs [Schoo92]. This library works with a window manager for the X window system environment. Windows inserted onto the screen will generally be decorated and placed by the window manager. Moving and resizing of windows will be done with the aid of the window manager.

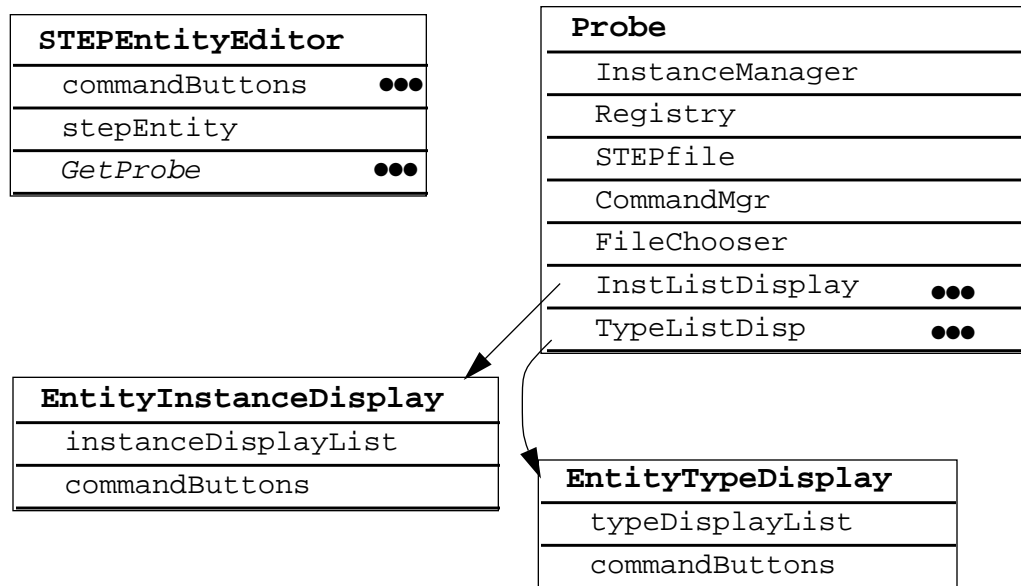
### 3.3.1 Class Relationships within Library

A SEE is made up of attribute rows and buttons. The attribute rows pass messages to the SEE, as appropriate.

The information communicated between the different windows in this library goes through the `Probe` class. For example when the save button on a SEE is hit, the information goes through the `Probe` to update the same information in the Instance List.

FIGURE 7

VTS Interface Library classes



### 3.3.2 Relationships to other Libraries

Much of the lower level user interface functionality is derived from the Interviews class library, a public domain C++ library available from Stanford [Linton91]. The `Probe`

class manages the interaction with the data objects managed by classes of other libraries. The *Probe* class manages the interaction between the displays. Its main function is to manage the display information and to keep it consistent with the underlying information objects. (The same information in different windows is kept consistent.)

The *Probe* receives an interpretation of keystroke or button commands and performs the function. For example, to save the instances, the *Probe* receives the list of commands to execute from the *CommandManager* (in the *Data Editor Library*).

When a button is pushed the *Probe* executes a single command for the selected instance and is “notified” by the generic *Interview* class *Subject* when it is set by the button object.

In general, display objects such as those in the *SEE* point to the actual information objects instantiated from other classes such as the *STEP Class Library*. The *SEE* itself points to an instance of a *STEPentity* (from the *STEP Class Library*). A particular row of a *SEE* which displays the attribute information for a particular entity points to a particular attribute and an instance of the class *STEPattribute* (also from the *STEP Class Library*).

### 3.3.3 Classes

The classes in the *VTS Interface Library* group the functionality of existing *Interviews* library classes into a meaningful interface.

---

**TABLE 3**

---

Primary Classes of the VTS User Interface Library

---

<i>StepEntityEditor</i> ( <i>SEE</i> )	<i>Probe</i>
<i>EntityInstanceList</i>	<i>EntityTypeList</i>

#### ***StepEntityEditor* (*SEE*)**

The *StepEntityEditor* is a visual representation of an underlying *STEPentity*. The *SEE* allows a user to interactively edit a *STEPentity* on the screen. The *StepEntityEditor* class is independent of the particular *STEPentity* type. The *SEE* sizes itself appropriately based on the number of *STEPattributes* contained in the underlying *STEPentity*. When the user is editing through a *SEE*, the current row corresponds to an attribute of the underlying entity.

*SEE* windows are created and appear when a command for creating or editing an entity instance is given. They disappear (unless “pinned”) when a command is given to save the instance, delete the instance, or close the window. A single *SEE* window is shown in **FIGURE 8**.

FIGURE 8 STEP Entity Editor (SEE) window

dp - #2957 Cartesian\_Point

editors

Label: importantPoint

x_coordinate	21.235	Length_Measure
y_coordinate	20.0933	Length_Measure
[z_coordinate]	5.89939	Length_Measure

message line

M save i c d r e m l

The SEE window for an instance lists the name of the type of entity and the identification number of the entity at the top of the window. Identification numbers are assigned to entity instances either as read from an input file or in numerical order. The remainder of the window consists mostly of rows for the attributes of the entity, one attribute per row. Each row has three sections: the name of the attribute, a space for the user to enter the value of the attribute, and a description of the required type of the attribute. When an entity instance is first created, the middle section of each line is blank. When an entity instance is edited, blank values may be filled in or existing values changed.

The message line in a SEE is a place where the software can give the user feedback. For example if the user attempts to save an entity such as the one illustrated above without a valid `x_coordinate` the message: missing value for `x_coordinate`, would appear in the message line.

Each SEE window has a “save” button at the bottom. When this button is selected, the window disappears, and the contents of the window are transcribed to the Entity Instance List.

The SEE window helps prevent errors by refusing attempts to enter invalid data for attribute values and by checking data types again when an instance is transcribed from the SEE window to the Entity Instance List. Invalid data is not transcribed.

Operations which a user may perform via a SEE can:

- activate operations using an emacs-like key binding interface,
- create new instances and automatically make the connection to the “current” attribute,
- provide both key bindings and visual buttons for the user.

The StepEntityEditor provides the following editing operations on an entity instance:

- Save Complete
- Save Incomplete



- Cancel
- Delete
- Replicate
- Edit Attribute's Instance
- Mark (select marked instance for current attribute)
- List Values (for an enumeration)

### **EntityInstanceListDisplay**

The `EntityInstanceListDisplay` class is a display object. It contains an object which is a list of string representations of the actual underlying data entities. This scrollable list is the primary user interface mechanism for selecting particular entities. It also contains a collection of buttons which the user can use to perform operations on the underlying entities. From the user interface, the user can mark an entire set of instances, delete or save, and invoke these operations at once via an execute button (or keystroke). Another field allows the user to search for particular entities by entering a substring.

### **EntityTypeListDisplay**

The `EntityTypeDisplayList` has many of the same functional mechanisms as the `EntityInstanceDisplayList`. It presents the user with a list of types which the user may select. The user may elect to create a new instance of the selected type or display more semantic information about the type. This window also contains a searching function.

### **Probe**

The `Probe` is the main grouping object. It contains pointers to the `InstanceManager`, `Registry`, `STEPfile`, `Command Manager`, `FileChooser`, `InstListDisplay`, `TypeListDisp`. It also has a number of menu objects. In a sense the `Probe` object is the object oriented equivalent of a main routine in a C program.

---

## **4 Application Model Specific Layer**

---

The *application model specific* layer represents the components which are tailored to the application model undergoing validation. These components are updated each time the application model changes.

The *STEP Schema Class Library* is the set of files that result from the translation of an EXPRESS schema. These files are generated automatically using the *Fed-X Toolkit* [Clark92] for translating EXPRESS and are producible from an EXPRESS schema. The program *fedex\_plus*, which is a backend to *Fed-X*, takes a conceptual data model written in EXPRESS as input and generates three C++ files for each schema [Morris92]. The C++ code in these files provides the class definitions and member functions for STEP entities needed by an application program.

Applications in projects working on process planning research and an IGES to PDES translator have used the C++ code produced as the output of *fedex\_plus*. The *STEP Schema Class Library* is the result of a mapping process between EXPRESS and C++. The mappings occur as follows:

- EXPRESS entities are translated into C++ classes derived from the class *STEPEntity*.
- EXPRESS attributes are translated into data members.
- Public access functions are automatically created allowing values to be read and written to the data members.
- Instances of Entity and Attribute descriptors are created for populating a Registry.

---

## **5 Data Probe: An Example Application**

---

The Data Probe is the major application which uses the various libraries described in this document. It is intended to allow people involved with the creation and testing of STEP Application Protocols to examine and populate data files corresponding to those models.

New versions of the Data Probe may be automatically generated for an EXPRESS schema. Each Data Probe executable is specific to a particular schema. A UNIX shell script called *mkprobe* takes as input an EXPRESS schema and outputs a new Data Probe specific to that schema. The process of creating a new Data Probe is completely automated through this process. It is anticipated that users will not be working with very many different schemas and will instead concentrate on one schema at a time.

FIGURE 10 shows the main windows of the Data Probe built for Part 42 of STEP. The editor consists of four types of persistent windows, all of which may be manipulated in the typical ways (move, open, close, hide, expose, resize, etc.):

- 1 Data Probe management window (FIGURE 9),**
- 2 STEP Entity Editor (SEE) windows (left side of FIGURE 10),**
- 3 an Entity Type List window (right side of FIGURE 10),**
- 4 and an Entity Instance List window (FIGURE 11).**

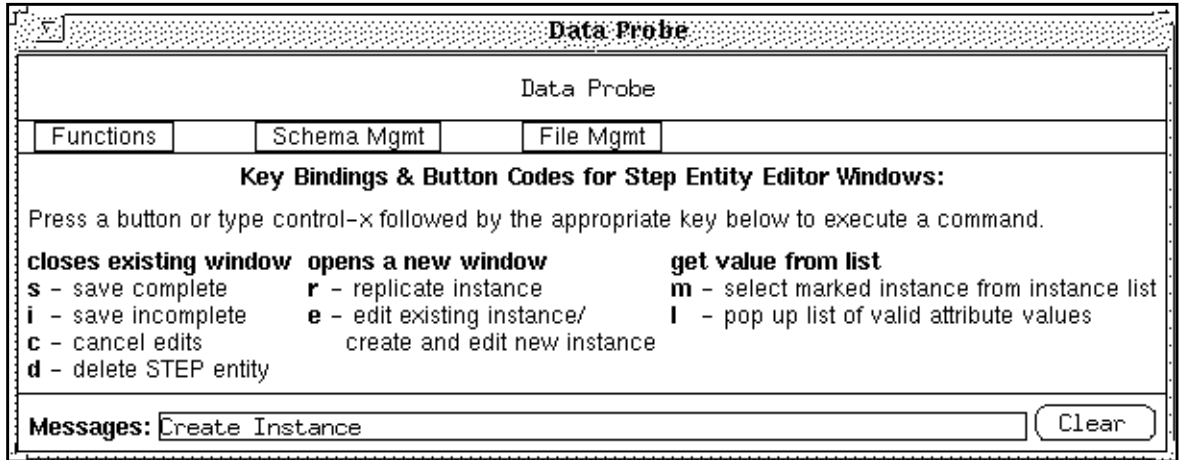
Any number of temporary STEP Entity Editing (SEE) windows may be created and destroyed as the editor is being used. A single SEE window is shown in the left side of FIGURE 10, partly obscuring the Entity Type List window.

### **Data Probe Management Window**

The Data Probe management window provides the user with systems functions such as saving or appending files, clearing the Entity Instance List, and quitting the editor. There

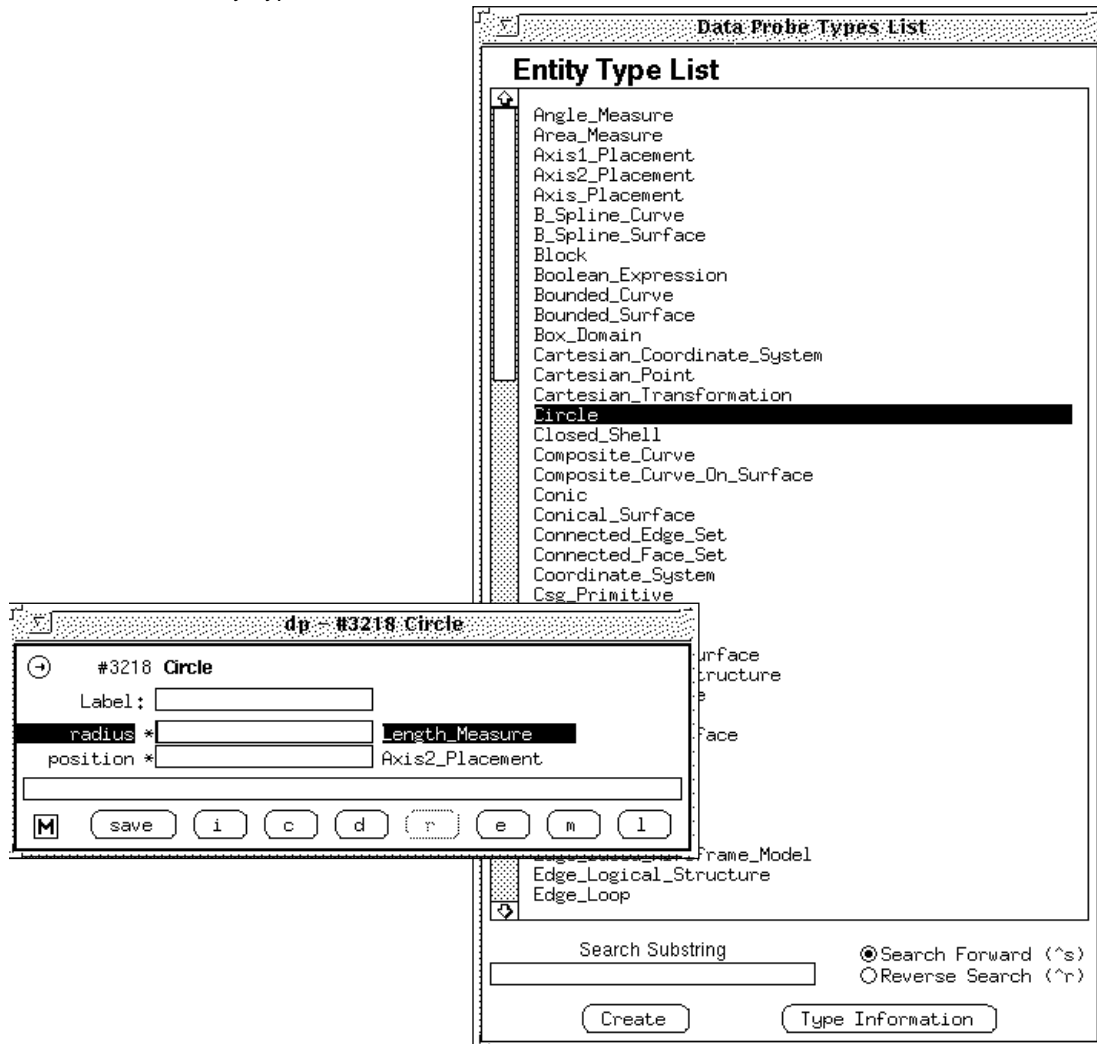
is also a subwindow for brief messages to the user. In FIGURE 9 the status window provides feedback that an instance has just been created.

FIGURE 9 Data Probe Management Window



### Entity Type List Window

The Entity Type List window contains a scrollable list of all the entities defined in the EXPRESS schema used to create the editor. Instances of these entities make up a STEP file. In FIGURE 10, the first few entities in this window are angle\_measure, area\_measure and axis1\_placement. The circle entity is highlighted in this window because the user has selected it. Below the list of entities there are some command buttons.

**FIGURE 10** Entity Type List window and new SEE

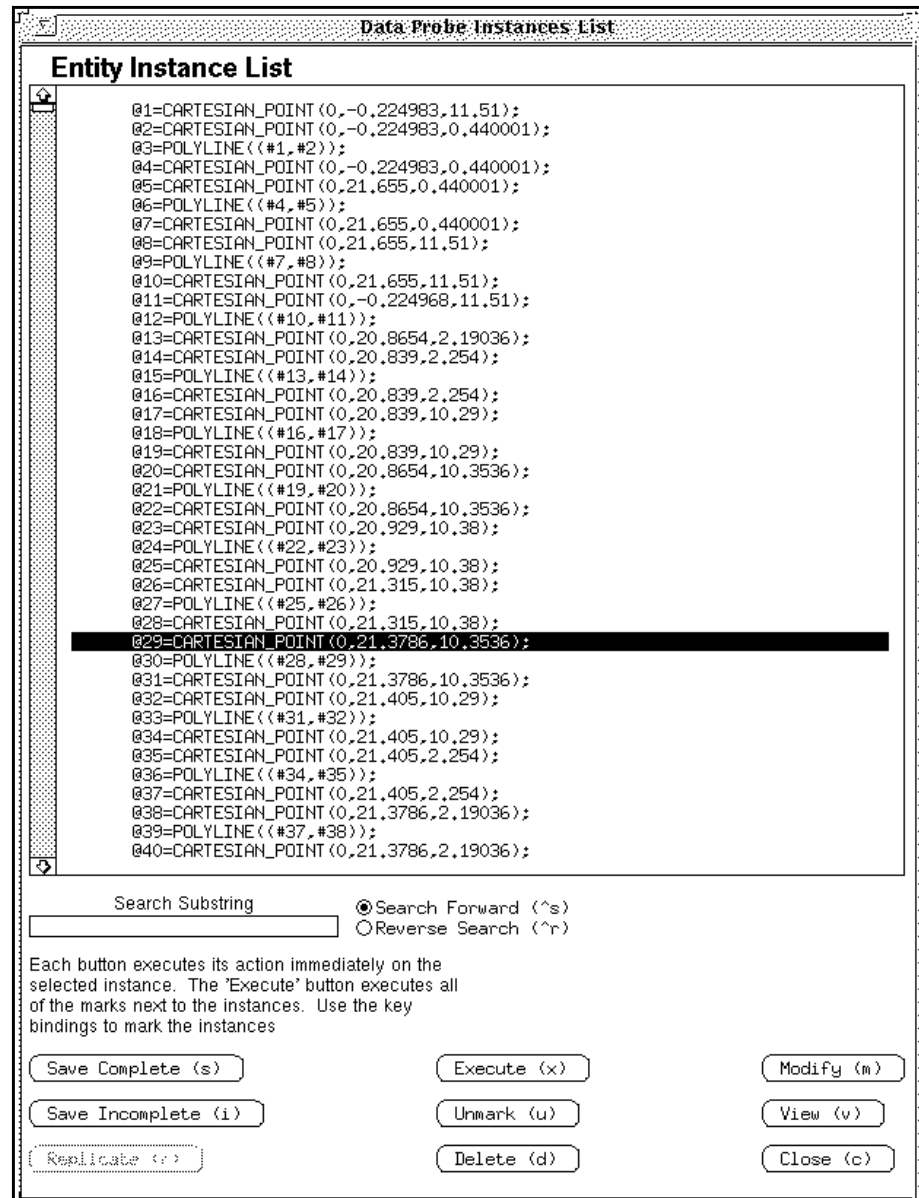
This window is used to select the type of entity to create when the user decides to create an instance of an entity. The user simply selects the name of the entity type to be created (by pointing at the name with the mouse cursor and clicking a mouse button) and then selects the “create” command button in a similar manner. This causes a SEE window to pop up for a new instance of that type of entity. The example illustrates an instance of “Circle” has just been created.

### Entity Instance List Window

The Entity Instance List window has two main parts. The top part has the appearance of the “DATA” section of a STEP file. Entity instances are listed, one instance per line. An indication of the editing status (such as marked for deletion) of an instance may be displayed to the left of the instance.

FIGURE 11

Entity Instance List window



The bottom part of the window contains the controls for a searching utility that may be used to find text strings. The bottom part also contains a set of command buttons to perform actions on instances such as: delete, modify, view, save, etc. To use the command buttons, an instance is selected with the mouse, and then the desired command button is selected with the mouse. If either viewing or modifying is selected, a SEE window pops up.

## 6 Conclusions, Summary and Future Directions

---

A functioning system called the Data Probe has been designed and implemented. At the time of writing this document it is under preliminary testing by knowledgeable users. The implementation of a set of class libraries has proven to be a flexible and robust approach. The libraries have been designed to allow the software developer the freedom to choose user interface styles and systems.

The initial implementation of the Data Probe is limited to the use of one input file at a time. The underlying data structures however will allow the use of multiple files and this extension is planned for the future.

Future work will include a more sophisticated use of schemas with more display of schema information and integration of a document browsing utility for on-line access to the ISO documents. Another future activity is the move to an SDAI (STEP Data Access Interface). SDAI will be a standard mechanism to access STEP data. As the standard is defined data accessed with the Data Probe will move through an SDAI interface. The intent is to use the same SDAI interface with persistence data repositories such as an object oriented data base for the persistent storage of STEP data.

---

**7 References**

---

- [Clark92] Clark, S.N., Libes, D., Fed-X: The NIST Express Translator, NISTIR 4822, National Institute of Standards and Technology, Gaithersburg, MD, April 3, 1992.
- [ISO11] ISO 10303 Industrial Automation Systems -- Product Data Representation and Exchange -- Part 11: Description Methods: The EXPRESS Language Reference Manual, Draft International Standard, ISO TC184/SC4, Spiby, P., ed., July 15, 1992.
- [ISO21] ISO 10303 Industrial Automation Systems -- Product Data Representation and Exchange -- Part 21: Clear Text Encoding of the Exchange Structure, Draft International Standard, ISO TC184/SC4, Van Maanen, J., ed., July 15, 1992.
- [ISO42] ISO 10303 Industrial Automation Systems -- Product Data Representation and Exchange -- Part 42: Integrated Generic Resources: Geometric and Topological Representation, Committee Draft N121.5 ISO TC184/SC4/WG4, Goult, R., ed., May 14, 1992.
- [Kramer92] Kramer, T., Morris, K.C., Sauder, D. A Structural EXPRESS Editor, NISTIR 4903, National Institute of Standards and Technology, Gaithersburg, MD, July 1992.
- [Linton91] Linton, M., InterViews Reference Manual Version 3.0-alpha, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Silicon Graphics, January 1991.
- [McLay90] McLay, M.J., Morris, K.C., The NIST STEP Class Library, C++ at Work-'90 Conference Proceedings, (reprinted as NISTIR 4411,) September 1990.
- [Mitch91] Mitchell, M., A Proposed Testing Methodology for STEP Application Protocol Validation, NISTIR 4684, National Institute of Standards and Technology, Gaithersburg, MD, September 1991.
- [Morris91] Morris, K.C., McLay, M., Carr, P. J., Validation Testing System Requirements, NISTIR 4676, National Institute of Standards and Technology, Gaithersburg, MD, September 1991.
- [Morris92] Morris, K.C., Architecture for the Validation Testing System Software, NISTIR 4742, National Institute of Standards and Technology, Gaithersburg, MD, January 1992.

- [Schoo92] Schoonover, M., Bowie, J.S., Arnold, W., GNU Emacs UNIX Text Editing and Programming, Hewlett-Packard Press, 1992.
- [Strang86] Strang, John, Programming with Curses, O'Reilly and Associates Inc. 1986.



## **Appendix A**

### **EXPRESS Model of Registry Classes**

---

The following EXPRESS schema is a model of EXPRESS. This model was developed as a basis for a skeleton of schema information from an EXPRESS model available at run-time using the VTS software. (The VTS software components involved are the STEP Class Library and a schema library generated from EXPRESS using fed-x-plus.)

With these exceptions this schema includes everything covered by EXPRESS-G:

- intra-schema relationships
- cardinality
- derived attributes

```
SCHEMA    express_meta_model;
```

```
TYPE base_type =
ENUMERATION OF (
    INTEGER_TYPE, STRING_TYPE, REAL_TYPE, ENTITY_TYPE,
    AGGREGATE_TYPE, ENUM_TYPE, REAL_PTR_TYPE,
    INTEGER_PTR_TYPE, SELECT_TYPE, BOOLEAN_TYPE,
    LOGICAL_TYPE, NUMBER_TYPE, BINARY_TYPE, UNKNOWN_TYPE);

(*
attribute_type =
SELECT (
base_type, type_descriptor );
*)

END_TYPE;

ENTITY type_descriptor
(*
    SUPERTYPE OF ( ONEOF ( real_type_descriptor,
        string_type_descriptor, array_type_descriptor,
        bag_type_descriptor, list_type_descriptor,
        set_type_descriptor, enumeration_type_descriptor,
        select_type_descriptor ))
*)
;

    name:STRING ;
    fundamental_type:base_type;
    (* the fundamental type will default to certain values
    according to the subtype of the type_descriptor *)
    referent_type:type_description;
    description:STRING;

END_ENTITY;

ENTITY real_type_descriptor
```

```
SUBTYPE OF (type_descriptor);
precision_spec: OPTIONAL INTEGER;
END_ENTITY;

ENTITY string_type_descriptor
  SUBTYPE OF (type_descriptor);
  width: OPTIONAL INTEGER;
  fixed_size: BOOLEAN;
END_ENTITY;

ENTITY aggregate_type_descriptor
  ABSTRACT SUPERTYPE
  SUBTYPE OF (type_descriptor);
  (* The bounds spec have been simplified to integer values -- they
  can actually have functional values *)
  bound_1: INTEGER;
  bound_2: INTEGER;
  unique_elements: BOOLEAN;
  aggr_domain_type: type_descriptor;
END_ENTITY;

ENTITY array_type_descriptor
  SUBTYPE OF (aggregate_type_descriptor);
  optional_elements: BOOLEAN;

  (* WHERE exists (bound_1 && bound_2) *)
END_ENTITY;

ENTITY bag_type_descriptor
  SUBTYPE OF (aggregate_type_descriptor);
  (* unique_elements: always TRUE *)
END_ENTITY;

ENTITY list_type_descriptor
  SUBTYPE OF (aggregate_type_descriptor);
END_ENTITY;

ENTITY set_type_descriptor
  SUBTYPE OF (aggregate_type_descriptor);
  (* unique_elements: always TRUE *)
END_ENTITY;
```

---

## References

---

```
ENTITY entity_type_descriptor
  SUBTYPE OF (type_descriptor);
  entity_type: entity_descriptor;
END_ENTITY;

ENTITY enumeration_type_descriptor
  SUBTYPE OF (type_descriptor);
  elements: LIST OF UNIQUE STRING;
END_ENTITY;

ENTITY select_type_descriptor
  SUBTYPE OF (type_descriptor);
  elements: LIST OF UNIQUE type_descriptor;
END_ENTITY;

ENTITY attribute_descriptor;
  name:STRING;
  domain_type:type_descriptor;
  optional_value:BOOLEAN;
  unique_value:BOOLEAN;
  owner:entity_descriptor;
END_ENTITY;

ENTITY inverse_attribute_descriptor
  SUBTYPE OF (attribute_descriptor);
  inverse_attribute : attribute_descriptor;
( *
role_attr: ExplicitAttribute;
min_cardinality: bound_description;
max_cardinality: bound_description;
duplicates: OPTIONAL BOOLEAN;
( *      3-Apr-1992 kcm
WHERE
uniqueness_correct: (EXISTS (duplicates) AND max_cardinality > 1)
OR
(max_cardinality = 1 AND NOT EXISTS (duplicates);
* )
* )
END_ENTITY;

ENTITY schema_descriptor;
  name:STRING;
END_ENTITY;
```

---

```
ENTITY entity_descriptor;
    name:STRING;
    originating_schema:schema_descriptor;
    abstract_entity: BOOLEAN;
    subtypes: OPTIONAL LIST OF UNIQUE entity_descriptor;
    supertypes:OPTIONAL LIST OF UNIQUE entity_descriptor;
    explicit_attr:OPTIONAL LIST OF UNIQUE
    attribute_descriptor;
    derived_attr: OPTIONAL LIST OF UNIQUE STRING; -- not
implemented yet
    inverse_attr: OPTIONAL LIST OF UNIQUE
inverse_attribute_descriptor;

    UNIQUE name;

    (* functions
istypeof (entity_descriptor):
create_instance:
*)
END_ENTITY;
END_SCHEMA; --express_meta_model
```

---

**Appendix B**

---

---

**VTs Document Series**

---

This document complements others in the National PDES Testbed Report Series which provide detailed technical information relating to the Testbed software. Those documents which specifically address aspects of the Validation Testing System are described below.

*Validation Testing Systems Plan* lays out the tasks and the overall approach for the initial implementation of the Validation Testing System. (NISTIR 4417)

*Proposed Testing Methodology for STEP Application Protocol Validation* describes the complete process used to develop and validate application protocols. This methodology document focuses on the analysis of application models and planning for validation testing. (NISTIR 4684)

*Validating STEP Application Models at the National PDES Testbed* describes a strategy for automation based on an analysis of the information flow in the application protocol development and testing process, and on initial experiences with automation for validation testing at the National PDES Testbed. (NISTIR 4735)

*Validation Testing System Requirements* describes functional requirements for automation of the VTS. This document also provides an overview of the VTS software environment. Requirements for the VTS system are driven by the STEP development effort and reflect the needs of the National PDES Testbed users. (NISTIR 4676)

*Architecture for the Validation Testing System Software* describes an architecture for software which supports the testing of information models for validity and correctness. The architecture provides a basis for software development within the National PDES Testbed. (NISTIR 4742)

*Validation Testing System: Reusable Software Component Design* provides guidelines for the implementation of the VTS software. The document describes the design of software libraries which fit within the VTS architecture. These libraries enable the creation and support of tool which support the VTS testing methodology. Designs for the components of the software are also provided. (NISTIR 4937)