
Design of a C++ Software Library for Implementing EXPRESS: The NIST STEP Class Library

David A. Sauder - sauderd@cme.nist.gov

K. C. Morris - kc@nist.gov

1 Abstract

The NIST STEP Class Library is a C++ software toolkit designed to provide software “building blocks” for developing STEP software applications. It has evolved over past years along with ISO 10303-23 Product Data Representation and Exchange: C++ Programming Language Binding to the Standard Data Access Interface (SDAI) Specification. Experience gained in implementing SCL has been important for providing feedback into the C++ binding and SDAI. This paper presents basic data structures used in the SCL to implement an EXPRESS data dictionary and to instantiate instances of entities. Design decisions with respect to EXPRESS inheritance mechanisms which are difficult to implement in C++ are discussed. These include multiple inheritance and complex inheritance via AND and ANDOR constructs in EXPRESS. Design decisions are discussed that allow support of an early binding to an EXPRESS schema along with late binding style data access. Finally, implementation issues that have been overcome related to implementing EXPRESS aggregates using C++ templates are discussed.

2 Introduction

The NIST STEP Class Library (SCL) is a set of C++ class libraries designed to be used as a starting point for building applications based on an EXPRESS [ISO 10303-11] data specification¹. The software provides a dictionary of EXPRESS schema information and functionality for representing and manipulating instances of EXPRESS objects.

SCL was developed with several purposes in mind. Most notably it has been useful for validating emerging concepts for STEP implementation methods and for developing software for STEP-based applications. Developers of SCL have iterated through the following activities: STEP standards development, development of SCL based on STEP standards, and feedback into the standard as a result of implementation experience. Particular attention has been devoted to implementing STEP Part 21: Implementation Methods - Clear Text Encoding [ISO 10303-21], and Parts 22 [ISO 10303-22] and 23

1. This document assumes a basic knowledge of EXPRESS and C++ [Stroustrup90].

[ISO 10303-23] the Standard Data Access Interface (SDAI) and associated C++ language binding. Other purposes of SCL include to:

- help industry adopt STEP,
- help applications and tools conform to STEP, and
- stimulate commercial development of STEP.

This paper discusses some of the more advanced features of SCL and highlights design choices made and implementation challenges faced. The topics discussed are:

- the software environment including software libraries and components (section 3)
- the late-bound data dictionary implementation (section 4),
- generic constructs used to instantiate entities and attributes (section 5),
- schema-specific constructs used to instantiate entities (section 5),
- entity instantiation implementations according to EXPRESS inheritance used (ONEOF/OR, AND, and multiple) (section 5), and
- challenges in the implementation of aggregates using C++ template (section 6).

For more information on the SCL architecture and the overall design of the libraries the reader is referred to the following publications:

- Architecture for the Validation Testing System Software [Morris92]
- Validation Testing System: Reusable Software Component Design [Morris92b]
- Data Probe: A Tool for EXPRESS-Based Data [Morris93]
- Data Probe User's Guide [Sauder93]

3 Overview of the SCL Software Environment

SCL consists of schema-independent software and software that is generated for a particular schema. The schema-independent software components:

- generate schema-dependent software components,
- provide C++ base classes for schema-dependent generated classes to inherit from, and
- provide generic data structures to be initialized at run-time for a particular schema.

Figure 1 shows the four components used to generate schema-specific code for SCL. These components are:

- a file containing an EXPRESS specification,
- the EXPRESS to C++ translator (a.k.a. fedex_plus),
- the NIST EXPRESS Toolkit [Libes93a], and
- the EXPRESS Pretty Printer Toolkit [Libes93b].

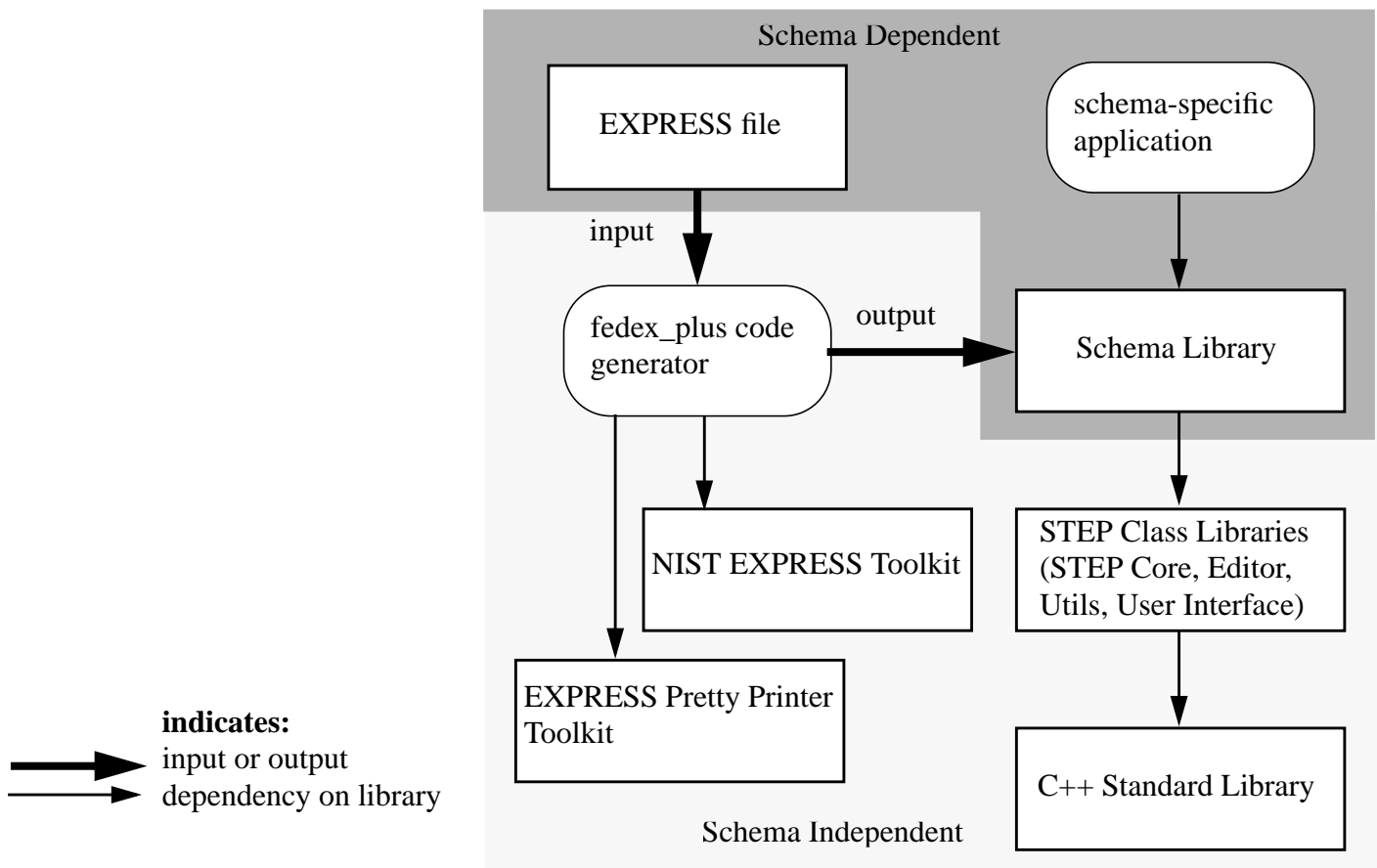
Fedex_plus generates a C++ class library representation of an EXPRESS schema with the help of the other two toolkits shown. Fedex_plus uses the NIST EXPRESS Toolkit to parse the EXPRESS schema, validate the EXPRESS syntax, perform limited semantic checking, and build an in-memory dictionary of the EXPRESS schema. Using the EXPRESS Toolkit schema dictionary and formatting routines from the EXPRESS

Pretty Printing Toolkit, fedex_plus then generates a C++ code representation of the schema called the Schema library.

The Schema library generated from fedex_plus uses schema-independent constructs from the STEP Core library. The STEP Core library is part of the STEP Class Libraries component shown in figure 1. The Schema library contains routines for initializing an EXPRESS-schema-specific dictionary at run-time using constructs from the STEP Core library. It also contains the schema-specific classes used to instantiate EXPRESS entities and types. Using inheritance these generated entity and type classes² extend constructs defined in the STEP Core library to be EXPRESS-schema-specific.

FIGURE 1

Overview of Components in the SCL Software Environment



SCL implements an early binding [ISO 10303-22 section 4.7] to an EXPRESS schema with early and late bound access to instantiations of EXPRESS entities and types. An

2. The terms entity class and type class will be used to refer to a class generated to represent an EXPRESS-schema-specific entity and type respectively.

early binding to an EXPRESS schema as defined by the SDAI means that schema-specific constructs are defined and bound to an application at compile-time rather than looked up and bound to generic constructs at run-time. The Schema library defines this early binding in the form of C++ class definitions representing EXPRESS-schema-specific types and entities. The early-bound class definitions representing entities include early-bound style attribute access functions for the entities. The names of the attribute access member functions are based on the attribute name as defined in the schema; they are bound to an application at compile-time.

SCL also provides late-bound style access functions for attributes of entity instances. The STEP Core library defines constructs to generically access attribute values and entities defined in the Schema library. All entity classes inherit this generic run-time access capability from a common supertype class. The generated entity class constructors initialize these inherited constructs necessary to provide late-bound style access. This late-bound access capability allows applications to be written generically so that the same application code may be bound to any number of schemas.

Schema-dependent or schema-independent application code may be developed using SCL; however, executable applications will always be schema-dependent. Applications developed using SCL early-bound access functions are by their nature schema dependent. Schema-independent applications may be developed by only using SCL late-bound functionality. Schema-independent applications may be compiled into schema-independent object files but must still be bound to a Schema library at link-time creating a schema-specific executable. Schema-independent SCL application object code may be linked with different Schema Class libraries to create different schema-dependent executables.

The following summarizes the libraries included in the SCL:

- Schema library - EXPRESS-schema-specific C++ code generated by `fedex_plus`. Defines schema-specific entity and type classes, attribute specific (early bound) access functions, and initialization functions for an EXPRESS dictionary. Based on the STEP Core library
- STEP Core library - schema-independent data structures for entities, types, and a data dictionary. Schema-independent code implementing reading/writing/validating functions for STEP data, and dynamic (late bound) attribute access functions
- Data Editor library - code supporting a STEP-specific data editor
- Utils library - general purpose utility code
- User Interface libraries - implement a graphical STEP specific data editor and schema information browsing tool based on SCL called Data Probe [Morris93, Sauder93].

4 The EXPRESS Data Dictionary

SCL provides a dictionary of EXPRESS schema information. The STEP Core library provides C++ container classes for creating a schema-specific dictionary at run time. A Schema library (generated by `fedex_plus` on a per schema basis) provides schema-specific initialization functions for creation of an EXPRESS dictionary based on STEP Core library constructs. The dictionary contains information from the EXPRESS model

on schemas, entities, defined types, enumerations, selects, lists, sets, bags, arrays, EXPRESS simple types³ (integer, real, number, string, binary, boolean, logical), and explicit, inverse, and derived attributes. Dictionary entries are pointed to by many constructs in the toolkit thereby avoiding unnecessary duplication of dictionary information. For example, all instantiations of an entity point to the dictionary entry for the entity. All attribute dictionary entries point to a type dictionary entry defining the attribute's type. These types of pointers provide many routes into the dictionary information.

The SCL dictionary is represented by the Registry class. Schemas, entities, and defined types are each contained in their own subdictionary. These subdictionaries can be accessed through the Registry. The Registry provides insertion, deletion, look-up, and traversal operations for each of the subdictionaries. The constructor for the Registry class accepts as an argument a pointer to a function from the Schema library. The function passed into the Registry constructor is invoked to initialize the dictionary.

The Schema library is generated to contain two types of initialization functions: a dictionary initialization function and one or more schema initialization functions. The dictionary initialization function initializes the EXPRESS dictionary for all the schemas in the EXPRESS model. It is always generated using the same function prototype so that generically written application code using the function only needs to be compiled once. At link time the function is bound to a Schema-library-specific implementation. To change a generically written application to be initialized for a different schema, the application need only be relinked so the dictionary initialization function in the new Schema library is bound to the application.

The dictionary initialization function calls the other type of initialization function, the schema initialization function, to populate the dictionary. Schema initialization functions are named based on the name of the schema they are generated to initialize. The dictionary initialization function calls a schema initialization function for each schema that is intended to be used with an application.⁴

The dictionary to be initialized (instantiated as a Registry class) is passed as an argument to each initialization function called. The dictionary initialization function accepts the dictionary to be initialized as a pointer to Registry argument. The dictionary initialization function calls all the schema initialization functions with the same Registry pointer as an argument. Each schema initialization function initializes the Registry dictionary with the EXPRESS schema information it was generated to initialize.

A Schema library contains several source files. For each schema from the EXPRESS the Schema library contains a header file containing schema-specific class definitions (.h), a corresponding file defining class member functions (.cc), and a file containing the schema initialization function (.init.cc). In addition to the schema-specific files three additional files are always included. The first file, schema.h, contains the dictionary initialization function prototype, the schema initialization function prototypes, and the include files for each of the separate schema header files. A developer using the early bound access functions need only include schema.h to use constructs from any schema.

3. The dictionary contains one entry for each EXPRESS simple type.

4. It is recommended that an EXPRESS long form be used. Use of multiple schemas sometimes results in compiler errors based on the ordering of generated C++ class declarations.

The second file, `schema.cc`, defines the dictionary initialization function. Lastly, as a convenience to developers a file, `make_schema`, is generated which contains a macro definition defining all the object files that need to be built for the schema. This file may be included in a generic Makefile for building Schema libraries.

The dictionary entries for entities and types⁵ may be used to instantiate instances of EXPRESS entities and types respectively. As part of the initialization of a dictionary entry for an entity or type, a pointer to a function is assigned a function which will create an instance of that entity or type. This additional dictionary functionality aides in the creation of schema-independent application code. It allows application programmers to create schema-specific classes through the generic function pointer rather than calling class constructors directly.

5 Entity Instantiation

The concepts of inheritance in EXPRESS and C++ differ. C++ and EXPRESS support the concepts of single and multiple inheritance. However, the default implementation of multiple inheritance in C++ differs from the multiple inheritance of EXPRESS. In C++ multiple paths in an inheritance hierarchy from a single child to a common ancestor lead to as many instantiations of the common ancestor as there are paths to it. Each instantiation in C++ of the multiple parent is qualified when accessed using the name of a child in a distinct path. EXPRESS on the other hand offers no duplication of ancestors no matter how many paths in an inheritance hierarchy lead to the common ancestor. EXPRESS also introduces the notion of AND inheritance which is not able to be supported directly using C++ inheritance.

The following subsections describe SCL implementation strategies used to support instantiation of entities using the different notions of inheritance in EXPRESS. Section 5.1 discusses the implementation of entities whose inheritance hierarchies include only single inheritance. Section 5.2 discusses the notion of complex inheritance. Section 5.3 discusses the implementation of entities that use multiple inheritance.

5.1 Instantiation of Single-Inherited Entities

This section describes the data structures used to instantiate EXPRESS entities (referred to as single-inherited entities) where:

- entities in the EXPRESS inheritance graph have no more than one supertype at any level in the graph, and
- an instance of a supertype is established through instantiation of only one of its subtypes at any level in the EXPRESS inheritance graph.

The data structures defined in this section provide a direct mapping between C++ inheritance and EXPRESS inheritance where the above holds true.

In SCL a Schema library contains one C++ class for each entity defined in the schema. Each entity class contains a data member for each attribute defined for that entity in the schema. The entity class also defines member functions for assigning values to and

5. Only those types implemented as C++ classes.

retrieving values from each attribute data member. These early-bound style access functions are named based on the name of the attribute. The entity classes use C++ inheritance to inherit from an entity class representing an EXPRESS-defined supertype, thereby allowing access to ancestor attribute data members and associated early-bound attribute access functions. For example, instantiations of `SdaiRectangles` in figure 2 cause instantiation of the supertype `SdaiShape` as well. `SdaiRectangle` inherits `SdaiShape`'s attribute data members and attribute access functions such as `_item_name` and its `Item_name()` access functions.

FIGURE 2

EXPRESS Entity and Corresponding Generated Entity Instance C++ Class Definition

```
TYPE label = STRING;
END_TYPE;


TYPE color = ENUMERATION OF (red, green, blue,
yellow, orange, white, black, brown);
END_TYPE;

ENTITY shape
SUPERTYPE OF (ONEOF (circle, triangle, rectangle));
    item_name : label;
    item_color : OPTIONAL color;
    number_of_sides : INTEGER;
END_ENTITY;

ENTITY rectangle
SUPERTYPE OF (square)
SUBTYPE OF (shape);
    height : length_measure;
    width : length_measure;
END_ENTITY;
```

```
class SdaiShape : public STEPntity {
protected:
    SdaiString _item_name ;
    SdaiColor _item_color ; // OPTIONAL
    SdaiInteger _number_of_sides ;
public:
    constructors, destructors, etc.
    const SdaiLabel Item_name() const;
    void Item_name (const char * x);
    const sdaiColor Item_color() const;
    void Item_color (sdaiColor x);
    const SdaiInteger Number_of_sides() const;
    void Number_of_sides (SdaiInteger x);
};

class SdaiRectangle : public SdaiShape {
protected:
    SdaiReal _height ;
    SdaiReal _width ;
public:
    constructors, destructors, etc.
    const SdaiLength_measure Height() const;
    void Height (SdaiLength_measure x);
    const SdaiLength_measure Width() const;
    void Width (SdaiLength_measure x);
};
```



Early bound access functions

The `STEPntity` class (see figure 3) from the STEP Core library is used as a base class for all classes used to instantiate entities. It defines general purpose functionality common to all entity classes. `STEPntity` also provides functions for reading instances from and writing instances to Part 21 files, validating entity attribute values, and providing generic (late-bound) access to all attribute values associated with an entity

instance. STEPentity provides class data members for containing error information, Part 21 comments, the Part 21 entity name (e.g. #1), and a pointer to the dictionary entry representing the entity.

FIGURE 3

Partial C++ Class Definition for STEPentity

```
class STEPentity : public Entity_instance {
public:
    STEPattributeList attributes; // generic attribute list
    int STEPfile _id;           // Part 21 entity name e.g. #2
    ErrorDescriptor _error;      // storage area for error information
    SCLstring *p21Comment;      // storage area for comments
    EntityDescriptor *eDesc;     // pointer to dictionary entry for entity
    int _complex;               // indicates this is part of a complex instantiation
    constructors, destructors, access functions, etc
    read, write, validate entity functions
}
```

The STEPentity class uses a list of attributes to provide generic access to the attributes specific to the entity classes. This is done through a linked list data member containing a generic attribute class instantiation for each attribute associated with the entity. The attribute list is used to implement the reading, writing, validating, and late-bound accessing functionality mentioned above. The list is built at run time by the constructors of entities inheriting from STEPentity. The ordering of the attribute class list entries corresponds to the order in which the attribute values would be read from a Part 21 file. The generic attribute classes on the attribute list are of type STEPattribute.

FIGURE 4

Partial C++ Class Definition for STEPattribute

```
class STEPattribute {
    ErrorDescriptor _error; // contains associated error information
    unsigned int _derive : 1; // has this attr been derived by another attr?
    const Attr *aDesc;       // attribute dictionary entry
    union { // discriminate use of ptr with underlying type info in aDesc
        pointer for each of the possible underlying attribute types
    } ptr;
    constructors, destructors, access functions, etc
    read, write, validate attribute value functions
}
```

The STEPattribute class (see figure 4) provides generic access to an attribute's value and its type. STEPattribute provides access to the attribute's dictionary information through a data member which points to the dictionary entry for the attribute. Information available from the attribute's dictionary entry includes:

- the attribute name,
- pointers to the dictionary entries for the attribute's type and owning entity, and
- unique, optional, and derived information.

STEPAttribute provides access to the attribute's value through a union data member containing pointers. The union data member contains a pointer to each possible underlying attribute type⁶. The pointer in the union data type is discriminated through the pointer to the dictionary entry which contains type information for the attribute. This union data member points to the storage area for the attribute defined in the attribute's defining entity class⁷.

The `_derive` data member flag in STEPAttribute indicates a different aspect of EXPRESS DERIVE information from the derive flag in an attribute's dictionary entry. The `_derive` flag in STEPAttribute is used to specify that an attribute is derived (by way of an EXPRESS DERIVE statement) in a subtype of the entity instance. A value of "true" in this field indicates that the value would be written as an asterisk in a Part 21 file. In contrast the derive flag in an attribute's dictionary entry specifies that the attribute is "deriving" a supertype's attribute value. An occurrence of a derive flag set to "true" in the dictionary would cause the `_derive` flag in the STEPAttribute for a supertype's attribute to be set to "true".

All SCL C++ classes used to instantiate entities (subclasses of STEPEntity), attributes (STEPAttribute), and types (e.g. for enumerations, aggregates, selects, strings, etc.) know how to read, write, and validate their values. This behavior makes reading, writing, and validating values conceptually very simple. For example, a class instantiating an entity calls the base class STEPEntity function to read an entity. The STEPEntity read function traverses its list of STEPAttribute classes directing each STEPAttribute class to read its value. Each STEPAttribute class looks at its attribute dictionary entry pointer to determine its type. For types implemented as C++ classes STEPAttribute directs the type to read its value. For types implemented as C++ built-in types STEPAttribute uses the C++ input functions to read the value. Writing and validating values are implemented in a similar manner. Reading, writing, and validating Part 21 files are implemented by an SCL STEPfile class in a similar manner by directing entities to read, write, or validate themselves.

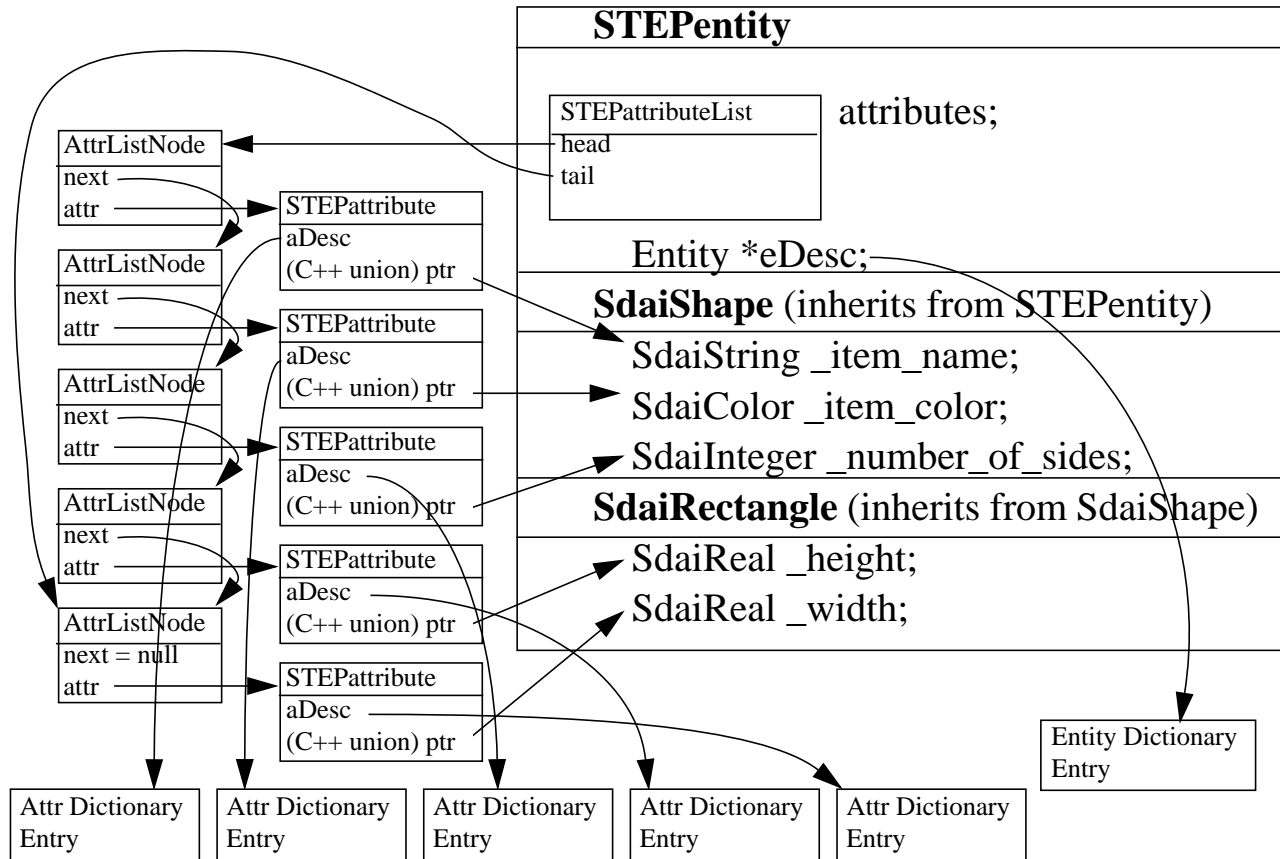
Figure 5 summarizes the relationships between the entity classes (for Shape and Rectangle), the base class STEPEntity, STEPAttribute, and the dictionary entity and attribute classes.

Application code to create an entity may be written using schema-dependent or schema-independent methods. Entity classes instantiated directly through use of the entity's class constructor would cause application code to become schema-dependent and thus could not be linked to work with multiple schemas. Schema-independent application code may be developed by using dictionary objects to create entities.

6. For EXPRESS types defined in the model only a pointer to the base class of the type is included. For example, each select type defined in EXPRESS inherits from a base class called SdaiSelect. The union data type contains a pointer to the base class SdaiSelect which at run time is used to point to any of the defined select subclasses.

7. The entity class defining the attribute's storage space is a subclass of the STEPEntity class containing the list of STEPAttributes.

FIGURE 5

C++ Class Relationships Involved in Single-Inherited Entity Instantiation⁸

The entity dictionary entries⁹ and the Registry class provide functionality necessary for writing application code in a schema-independent fashion. Both provide schema-independent methods for creating entity instances. Both methods ultimately create entity instances through use of a creation function found as part of each entity's dictionary entry. The dictionary entry contains a function pointer initialized to point at a creation function for the entity. Once the appropriate dictionary entry has been accessed the creation function may be invoked through the function pointer to create an entity instance. The new entity instance is returned through a pointer to the common entity base class **STEPEntity**. The entity dictionary entry is accessible through a pointer found in **STEPEntity** or through a lookup function in the Registry class (see section 4 for a description of Registry).

8. These relationships are also involved in multiple-inherited entity instantiation. See section 5.3.

9. Recall that each entity defined in EXPRESS has an associated dictionary entry class. Each entity dictionary entry is a schema-independent data structure created and filled in at run time for an entity from a particular schema.

The Registry class method available for schema-independent creation of entities is a create function that accepts the name of an entity in an argument of type string. Using the string entity name the Registry class locates the dictionary entry for the entity and calls the creation function associated with the dictionary entry. The resulting entity instance is returned through a pointer to the base class STEPentity.

5.2 Complex Entity Instantiation

This section describes the data structures used to instantiate entities where:

- an instance of a supertype is established through instantiation of more than one of its subtypes at any level in the EXPRESS inheritance graph, and
- Part 21 requires the use of the external mapping method.

The term complex will be used to refer to entities where the above applies.

Several factors influenced the choice of implementation of complex instances. These include initial SCL design considerations, compatibility with the existing SCL toolkit, and EXPRESS considerations regarding complex instances.

Two factors influenced initial design decisions in the development of SCL entity instantiation constructs. In an effort to provide an object oriented STEP toolkit, entity instantiation constructs for entities were designed to take advantage of benefits of using C++ inheritance. This was done in spite of the fact that C++ inheritance is not able to directly support complex inheritance in EXPRESS. These decisions were made knowing the implementation of complex inheritance would be affected. It was determined that the use of complex instances would be handled as exceptions. This decision was justified given the lack of support for complex inheritance offered by C++ and the fact that at the time complex inheritance was not commonly used.

Based on several requirements complex entity instances were implemented using a late-binding to the schema. When designing the data structures to support complex instances, the design for single inheritance was once again confirmed. Some levels of consistency in the implementation of entity instance data structures were sacrificed in order to preserve the object-oriented nature of the toolkit. Still, compatibility requirements with the existing SCL constructs remained for the complex instances solution. In addition, the design needed to limit the proliferation of the number of possible complex instance data structures which, unless carefully avoided in the data model, results from ANDOR inheritance being the default inheritance in EXPRESS. The resulting design provided for a late-bound style implementation based on constructs that already existed in the SCL with a few additions.

The late-bound implementation means that complex instances are not prespecified in the form of schema-dependent generated C++ classes but are built using generic data structures at run time (with the help of dictionary information).

Some consistency between complex and non-complex entity instances has been sacrificed while parts of the interface remain the same. The interface for reading, writing, and validating instances remain the same regardless of whether the instances are complex or not. The differences arise when directly accessing information from the underlying data structures. Since the complex instance implementation is late-bound, the early-bound attribute access functions are not implemented. These could be added later via generated casting functions. A function is provided for accessing entity element

parts of the complex instance data structures. Once an entity element is accessed the interface for late-bound style access to attribute values is consistent with non-complex instances.

FIGURE 6

Partial C++ Class Definition for STEPcomplex

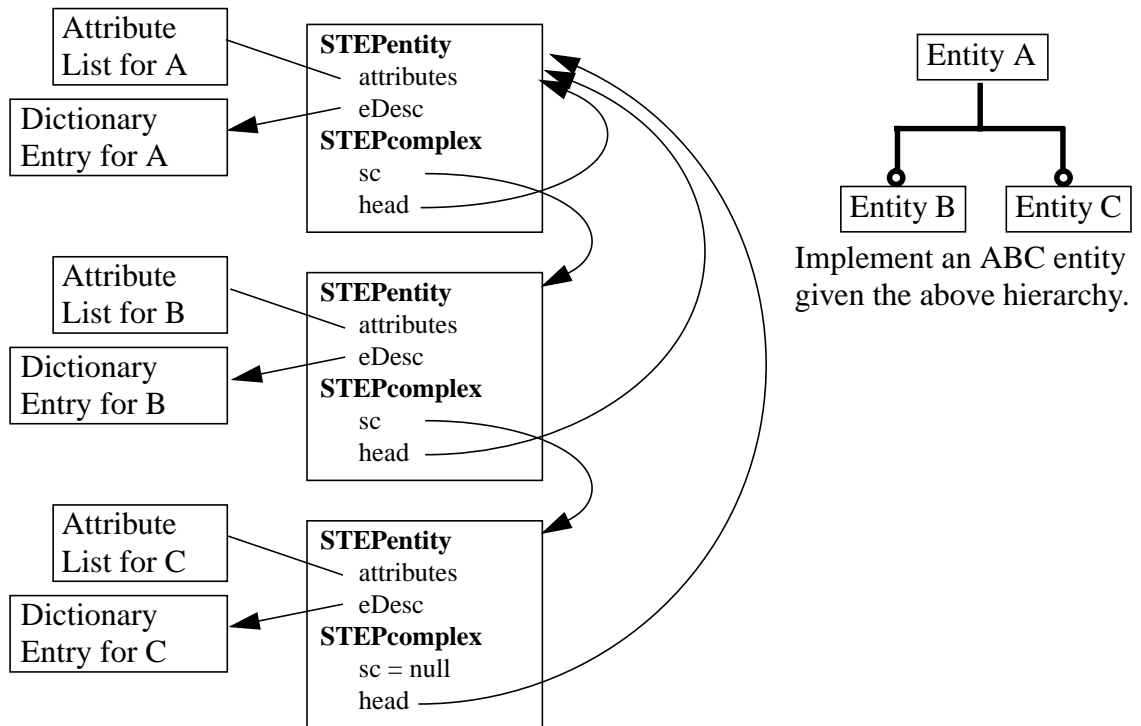
```
// STEPentity contains a boolean data member to indicate whether or not it
// is instantiated as a complex instance
class STEPcomplex : public STEPentity {
public:
    STEPcomplex *sc; // points to the next entity in the complex instance
    STEPcomplex *head; // points to the head entity in the complex instance
    int visited; // used when reading
    constructors, destructors, etc
    read, write, validate entity functions
}
```

Complex instances are implemented using the STEPentity (see previous section) and STEPcomplex (see figure 6) classes. STEPcomplex, a subtype of STEPentity, uses the constructs in STEPentity to define and manipulate the attributes defined for a single entity part of a complex instance. Complex instances are built at run-time from STEPcomplex classes. One STEPcomplex instance is created for each entity part included in the make-up of the complex instance. The attributes for each entity part are created within the STEPentity attribute list based on entity and attribute information in the appropriate dictionary entries. STEPcomplex contains constructs to link several STEPcomplex classes together to form an entire complex instance as the sum of the individual entities defining the complex instance.

STEPcomplex redefines virtual functions in STEPentity for reading, writing, and validating the entity's attribute values based on the configuration of STEPcomplex data structures. The STEPcomplex read, write, and validate functions perform the operations by calling the STEPentity read, write, or validate function for each entity part making up the complex instance. Figure 7 shows the relationships between the STEPentity and STEPcomplex data structures for a particular complex instance.

Complex instances are created at run time by calling the STEPcomplex constructor with a list of strings containing the names of the entities that make up the complex instance. The STEPcomplex constructor processes each name in the list creating a list of STEPcomplex classes one for each entity contained in the list. Each STEPcomplex class in the list initializes a pointer data member to point to the dictionary entry for the entity part it represents. It uses that dictionary entry to access the dictionary entries for the attributes and the attributes' EXPRESS types. STEPcomplex uses these attribute dictionary entries to create the list of STEPattribute classes where attribute values associated with the entity part of the complex instance are stored. Attribute types implemented as C++ built-in types are created directly by STEPcomplex. Attribute types implemented as C++ classes are created by STEPcomplex through a creation function associated with the type dictionary entry.

FIGURE 7 C++ Class Relationships Involved in Complex Entity Instantiation



5.3 Multiple-Inherited Entity Instantiation

This section describes the data structures used to instantiate EXPRESS entities where the following conditions hold:

- at least one entity in the EXPRESS inheritance graph specifying the entity to be instantiated has *more* than one supertype, and
- an instance of a supertype is established through instantiation of only one of its subtypes at any level in the EXPRESS inheritance graph¹⁰.

A multiple-inherited entity instance is represented by a generated class which inherits from STEPentity. The classes generated to represent multiple-inherited entities are identical to single-inherited entity classes except that the constructors of the multiple-inherited classes differ. The constructors for multiple-inheritance create additional structure for representing the multiple parents. The representation of multiple-inherited instances start with the same structure as single-inheritance entities but link themselves to additional single-inheritance entities to represent multiple parents.

The SCL implementation of multiple-inherited entities creates a single-inherited entity based on a primary inheritance path¹¹. The entity class associated with the primary path

10. If this condition is not met the entity is complex and is handled as described previously.

will be referred to as the primary entity. The C++ class constructors for the primary entity create additional single-inherited entities to represent the additional parents. These additional parent entities are linked through a list implemented using STEPentity data members nextMiEntity and headMiEntity. When these additional parents are created, multiple parents associated with them are again handled in the same manner.

The creation of a multiple-inherited instance is dependent upon the ordering of the C++ constructor calls and uses a constructor generated to handle the case of multiple inheritance. When non-virtual C++ inheritance is used, the ordering of the constructor calls is specified and guaranteed to start with the base class's constructor and proceed down through the parent classes' constructors until the constructor for the lowest-level class is called. Multiple-inherited entity classes are designed such that the resulting order of the constructor calls causes the primary entity's list of attributes to be created with a specific order.¹² When a multiple-inherited instance is created, all the constructors for entities in the primary inheritance path use the same constructor calls as single-inherited entities.¹³ When a constructor is reached for an entity whose EXPRESS definition is defined to be multiple-inherited, it instantiates new single-inherited classes representing its additional parents. The constructors used to create the additional parents accept as an argument a pointer to the primary entity. This new parent entity uses this overloaded constructor for each class created in its inheritance hierarchy. The constructors in this "secondary" inheritance hierarchy add the attributes associated with each entity in the hierarchy to the STEPattribute list of the primary entity, which was passed as an argument to the constructors. When a secondary parent entity reaches a constructor for an entity that is multiply inherited, it operates in the same manner creating additional parent entities for each additional parent and passing on the primary entity pointer. In this way a complete list of all the attributes is built up in the proper order in the primary entity's attribute list.

Figure 8 illustrates the basic components of a multiply-inherited instantiation of an entity G. The primary inheritance path is A, B, G. Entity G is instantiated in the same manner as it would be if it did not contain multiple inheritance. Four entities are instantiated: one each for entities G, D, E, and F. Entities G, D, E, and F are linked together to form a complete entity G through a linked list (implemented via the nextMiEntity pointers) located in G, D, E, and F's STEPentity base classes. The first entity element in this linked list is entity G followed by entities D, E, and F. Each of the four entities have a pointer, headMiEntity located in their STEPentity base class, which points at entity G. The headMiEntity data member is used by each entity making up entity G to indicate its ultimate identity as a G instance. Each entity instantiated to make up entity G also has a pointer to its dictionary entry so that it knows necessary information about itself used

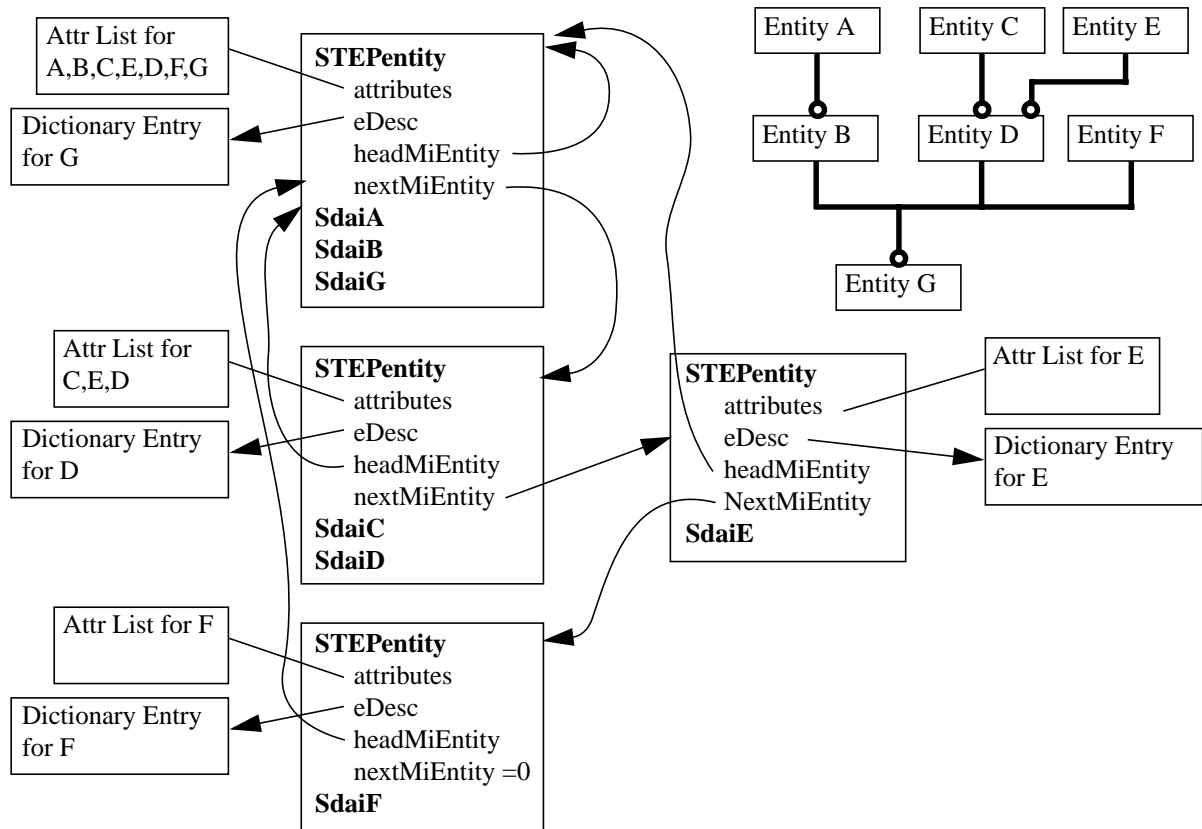
11. A single supertype/subtype path is followed in the EXPRESS inheritance graph from a root parent down to the leaf entity defining the multiple-inherited instance. This path in the graph is defined by proceeding upward from the multiple-inherited entity through the parents that are first mentioned in the EXPRESS SUBTYPE OF definition of the entity. This path will be referred to as the primary inheritance path.

12. The proper ordering of the attributes is the correct ordering for reading attributes from and writing attributes to a Part 21 file.

13. Recall that each of these constructors creates a STEPattribute class for each attribute it defines and adds the STEPattribute to the primary entity's STEPattribute list. Each STEPattribute class points to the storage area for the attribute which is defined within the class being constructed.

for reading, writing, validating, etc. itself. Entity G's STEPAttribute list contains the complete list of attributes for all of G's attributes (including ones inherited through B, D, and F). This attribute list is ordered according to how the attributes are read from or written to a Part 21 file.

FIGURE 8 C++ Class Relationships Involved in Multiple-Inherited Entity Instantiation



The creation of the data structures making up entity instance G involves the following relevant actions in order:

- G's STEPentity constructor creates its STEPAttribute list.
- A's constructor adds A's attributes to G's STEPAttribute list.
- B's constructor adds B's attributes to G's STEPAttribute list.
- G's constructor assigns its headMiEntity to point at itself and creates its alternate parent D providing a pointer to itself as an argument to D's constructor.
- D's STEPentity constructor is called.
- C's constructor (with a pointer to G argument) adds C's attributes to G's STEPAttribute list.
- D's constructor (with a pointer to G argument) makes its headMiEntity point at G and creates its alternate parent E sending its G argument pointer in as an argument to E's constructor.

- E's STEPentity constructor is called.
- E's constructor (with a pointer to G argument) makes its headMiEntity point at G and adds E's attributes to G's STEPattribute list.
- D's constructor (with a pointer to G argument) assigns E to its nextMiEntity pointer. It then adds D's attributes to G's STEPattribute list.
- G's constructor assigns entity D to its nextMiEntity pointer (now forming a list of G pointing to D pointing to E).
- G's constructor creates its alternate parent F sending a pointer to itself in as an argument to the constructor.
- F's STEPentity constructor is called and by default assigns nextMiEntity to be null.
- F's constructor (with a pointer to G argument) makes its headMiEntity point at G and adds F's attributes to G's STEPattribute list.
- G's constructor appends entity F to its nextMiEntity pointer list (now forming a list of G pointing to D pointing to E pointing to F).
- G's constructor adds G's attributes to G's STEPattribute list.

Attribute values are read, written, and validated in the same manner as single-inherited entity instantiations.

6 Template Issues in Dealing with Aggregates

This section describes two problems and solutions that came up when implementing aggregates in SCL using templates as specified in the first Committee Draft version of Part 23. The first problem occurred when trying to extend the Part 23 implementation. The second problem related directly to the Part 23 specified implementation.

Two requirements of the SCL aggregate implementation were that it match the Part 23 specification of aggregates and that it be extendable to accommodate reading and writing of STEP Part 21 exchange files. In addition, it was desired that the aggregate implementation be able to accommodate the SCL design where higher level objects could pass off tasks such as reading, writing, and validating to lower level or contained objects. For example, when the SCL file object STEPfile reads a file, it determines the entity to be read, creates the entity, and tells the entity to read its value. For each attribute associated with the newly created entity class, the entity tells the corresponding STEPattribute object to read its value. Each STEPattribute that has an underlying type representation as a class tells the type object to read its value. Following this style of implementation, the Part 23-specified aggregate template classes for set, list, bag, and array were extended to define template functions for reading, writing, and validating their own values. These aggregate functions were implemented such that aggregate template instantiations containing element types implemented as classes followed this same scheme using the element type classes' read, write, and validate functions to read, write, and validate the aggregate elements.

Since some EXPRESS types in the SCL (and in SDAI) are implemented as classes (e.g. selects) and some are implemented as C++ built-in types (e.g. int) a compile-time problem arises when an aggregate template instantiation attempts to invoke an element's class member function. The compiler expects template software code to be able to handle any possible template instantiation without respect to program logic. That is,

even though conditions could be added to the code that would not allow an element's function to be called when the underlying element is not implemented as a class, the compiler will not allow it.

The solution to the problem was to move the implementation of the desired activity from template functions to non-template functions. Each of these non-template functions shares the same name but is differentiated by the type of an argument which determines the version of the function that gets invoked at run-time. For example, the template function to read a value was changed to simply call a non-template function using a name that is common among all versions of the function. The argument passed into the read function is the template type-variable indicating the type currently associated with the template. Since the non-template read function is really several functions all sharing the same name, the template type that the template is currently associated with determines which non-template read function gets invoked at run-time. A read function that is invoked with a class as an argument then instructs the class to read its value. A read function that accepts an integer may simply call a function predefined to read an integer. The solution to this problem results from moving an action that would require one template function to deal with several types (which is what template functions must always do) in more than one way (individual types could not do this) to individual non-template functions (which are not required to deal with more than one type) that only deal with one type in their own way. The concept of polymorphism (one function with many forms) in C++ was used to separate the actions of reading different types into different functions at run-time.

A side effect of this solution is that it limits the general purpose nature of the template implementation. The result of overloading a function invocation to be bound to several function implementations based on EXPRESS types is that the template implementation may then only be instantiated for use with EXPRESS types. Instantiation of a template aggregate with a type not defined within EXPRESS would cause a compile time error. The template implementation may, however, easily be extended by defining the appropriate functions for dealing with the new type. (This limitation is of course not relevant to applications that only instantiate templates for use with EXPRESS types.)

The second problem that arose when implementing the aggregate templates according to Part 23 involved the function `CreateNestedAggregate`. Part 23 specifies that the `CreateNestedAggregate` function shall add an empty aggregate to the contents of an existing aggregate when the existing aggregate is defined to contain elements of type aggregate. If the aggregate's element type is not defined to be an aggregate, an error indicator shall be set.

The problem with implementing `CreateNestedAggregate` was similar to the first problem but the solution differed. The problem arose from trying to define a template function that would create an aggregate for some template instantiations and not create one for others. To create an aggregate element of type aggregate a create-aggregate function must be called on the aggregate element type. Since all aggregates are implemented as templates, the create-aggregate function is a template function. As a result the compiler requires that any possible type the aggregate could be instantiated for must have a create-aggregate template function. This is only possible when the element type used to instantiate the aggregate is an aggregate. Instantiation of the aggregate with any non-aggregate type causes a compilation error.

The solution to this problem involved creating template specialization functions. Template specialization functions are used to override template functions for certain types with which the template has been instantiated. The approach was to override the template functions for non-aggregate element types. The choice was to override these types since their number, though possibly large, equals the number of non-aggregate types that are defined in the EXPRESS and the number is finite. In contrast, although there are a finite number of aggregates defined in any one schema, the implementation of an aggregate is a template. Templates are meant to be instantiable with any defined type including templates (as is necessary when aggregates are nested). Thus, it is not possible to write template specialization functions to single out instantiations of templates that may be instantiated with templates and so on.

The template specialization functions define functions that will be called instead of the default function when the template has been instantiated with the type for which a specialization function has been written. These specialization functions simply set an error indicating that the CreateNestedAggregate function has been illegally called. The default CreateNestedAggregate function is then free to call the appropriate create-aggregate template function since the compiler expects that all types not implementing the create-aggregate function have been accounted for through template specialization functions¹⁴.

The specialization functions' discriminating argument type¹⁵ must match the template type exactly in order to be called. Entity, enumeration, and select types in SCL are all implemented as generated classes based on an EXPRESS specification. Each of these generated types inherits from its own common supertype. Specialization functions written to catch the common supertype were never invoked when the template was instantiated with any of the subtypes actually implementing the type. For this reason it was necessary to generate aggregate-specialization functions for the classes used to represent entity, enumeration, and select types.

A side effect of this solution is that it limits the general purpose nature of the template implementation. A template specialization function must exist for all non-nested aggregate instantiations or a compile-time error will occur when a template is instantiated for a type that is missing the specialization function. The result of requiring that a template specialization function must exist for all non-nested aggregate instantiations is that the template implementation may then only be instantiated for use with EXPRESS types. The template implementation may, however, easily be extended by defining the appropriate template specialization functions for dealing with the new type. This limitation is of course not relevant to applications that only instantiate templates for use with EXPRESS types.

Several other problems occurred with the use of templates to implement aggregates. The biggest of these problems was in dealing with various vendors' C++ compiler's handling of templates. In many cases, the implementation of aggregates using templates

14. Note - compilers handle these functions differently and thus this solution imposes portability problems.

15. The specialization function that gets invoked is determined by its function prototype. That is, the function is called with the current template type sent in as an argument. At run time, this causes the specialization function with the matching function prototype to be invoked.

as specified in Part 23 pushed the compilers beyond what they were able to support. Several rounds of bug reports, patches, and creative work-arounds had to occur in many cases even to get the implementation to build, much less work correctly. Another major impact on the development of the aggregate implementation had to do with the length of time and compiler options required to compile and link applications using templates. Different compilers required different compiler switches which often were not well-documented. The use of templates greatly increased the time it took to build code and test programs using templates thus trying the patience of developers and greatly increasing development time. Overall though, the SCL implementation of aggregates was useful for providing input back into the specified implementation for Part 23 and verifying that the ideas presented were implementable.

7 Conclusion

Implementation of the SCL software toolkit has been useful for validating Part 23 software implementation ideas and providing practical feedback into its development before obtaining approval and commitment for use as an international standard. The public availability of SCL and tools based on SCL has also been successful in lowering the cost of investment in the development and prototyping efforts for STEP by industry and academia.

The STEP Class Library and related tools are in the public domain and are available from NIST through the STEP On-Line Information Service (SOLIS) [Rinaudot]. SOLIS may be accessed by one of three methods:

- anonymous ftp: [ftp.cme.nist.gov](ftp://ftp.cme.nist.gov) (129.6.32.54)
- e-mail: nptserver@cme.nist.gov
- world wide web: <gopher://elib.cme.nist.gov> or <http://elib.cme.nist.gov:70/>

The software is located in the directory [pub/subject/sc4/tools/nist](#) and documentation is in [pub/subject/sc4/national/usa/nist/docs](#).

Contact the Manufacturing Systems Integration Division (npt-info@cme.nist.gov or 301-975-3508) for more information on how to obtain the software or for other information related to the work.

8 Acknowledgements

David Helfrick was the principal developer of the SCL aggregate implementation. He discovered the aggregate problems and solutions presented in this paper.

9 References

- ISO 10303-11:1994 *Industrial automation systems and integration Product data representation and exchange—Part 11: Description methods: The EXPRESS language reference manual.*
- ISO 10303-21:1994 *Industrial automation systems and integration—Product data representation and exchange—Part 21: Implementation methods: Clear text encoding of the exchange structure.*
- ISO 10303-22 *Industrial automation systems and integration—Product data representation and exchange—Part 22: Implementation methods: STEP data access interface.*
- ISO 10303-23 *Industrial automation systems and integration—Product data representation and exchange—Part 23: Implementation methods: C++ language binding to SDAI.*
- Libes93a Don Libes, *The NIST EXPRESS Toolkit - Design and Implementation*, Proceedings of the Seventh Annual ASME Engineering Database Symposium, San Diego, CA, August 9-11, 1993.
- Libes93b Don Libes, *Exppp - An EXPRESS Pretty Printer*, NISTIR 5292 (NTIS PB94 - 120797/AS), Gaithersburg, MD, November 1993.
- Morris92 KC Morris, *Architecture for the Validation Testing System Software*, NISTIR 4742, National Institute of Standards and Technology, Gaithersburg, MD, January 1992.
- Morris92b KC Morris, D. Sauder, S. Ressler, *Validation Testing System: Reusable Software Component Design*, NISTIR 4937, National Institute of Standards and Technology, Gaithersburg, MD, October 1992.
- Morris93 KC Morris, *Data Probe: A Tool for EXPRESS – based Data*, *Proceedings of the Seventh Annual ASME Database Symposium - Engineering Data Management: Key to Success in a Global Market*, American Society of Mechanical Engineers, New York, August 1993.
- Rinaudot Gaylen R. Rinaudot, *The IGES/PDES Organization: STEP On-Line Information Service (SOLIS)*, NISTIR 5511, National Institute of Standards and Technology, Gaithersburg MD, October 1994.
- Sauder93 David Sauder, *Data Probe User's Guide*, NISTIR 5141, National Institute of Standards and Technology, Gaithersburg, MD, March 1993.
- Stroustrup90 B. Stroustrup, ANSI X3J16/90-0020, *C++ Language System Reference Manual.*

Documents from the National Institute of Standards and Technology are available through the National Technical Information Service (NTIS), Springfield, VA, 22161.

To request the documents use the NISTIR number.