SCL Reborn -- The New SAS Component Language(SCL) in Version 7

Yao Chen, Display Products, SAS Institite, Inc. Cary, N.C.

Abstract

The SAS[TM] Component Language(SCL) in Version 7 is a genuine object-oriented programming language. In addition to inheriting all the existing functionality of the Screen Control Language(SCL) from Version 6, SCL now provides object-oriented programming constructs such as Class, UseClass, Interface, and dot notation. With these constructs, AF/SCL users can create object-oriented applications completely in SCL. In particular, they can use SCL to create and script the new SAS Component Objects. This paper will also address the topic of using the Interface model to design Model/Viewer applications, and a strategy for detecting Year 2000 problems by using the SCL Static Analyzer.

Introduction

With its powerful functionality, AF/SCL has become the language of choice for developing business solutions applications. For example, SAS/ASSIST[TM], SAS/EIS[TM], SAS/Warehouse-Administrator[TM], SAS/PH-Clinical[TM], SAS Enterprise Miner[TM], SAS/CFO-Vision[TM], SAS/Enterprise-Report, etc. are all written using Screen Control Language(SCL).

Two disparate seeds spawned the new SCL. First, a great deal of valuable feedback concerning AF/SCL was collected from the SASWARE ballot and other SAS local user groups. One of the most frequent requests was for a VB-like user-interface in the AF BUILD environment. Second, there has been a very strong interest within the software industry concerning the "thin"-client model. This model is characterized by fairly limited processing on the client (WEB) side with most of the heavy-duty actions (object method calls) being scripted on the server side. In the language of model/view, the user will deploy models on intelligent servers and use views on intelligent clients. In order to meet the requirements of both approaches, we created the new SAS Component Object Model (SCOM), which not only provides an object-oriented design model for SAS/AF BUILD users, but is also central to the server-side processing of the thin-client model. We have renamed Screen Control Language (SCL) to SAS Component Language (SCL) to reflect these new enhancements and SCL's full support of the SAS Component Object Model. A discussion of SCL's new features forms the basis of this paper. The topics are:

Table of Contents

General Enchancements for Version 7

```
 (1) Long Names/Long Labels(32 characters) and Long Strings(32K).

 (2) A large SCL programming model(No More 32K limitations).

 (3) Primitive data types(Numeric, Character, List, Object) and user-defined
     data types including class type and interface type.

 (4) Using the DCL statement to define variable types and to define local
     variables.
```

```
Using DCL statemnent to define SCL local variables.

Import Sashelp.Fsp.Collection.Class;
 DCL num n1;
 DCL num n2 n3 n4;
 DCL num n5, char c1 c2, list l1 l2 l3,
     object o1, Collection col1 col2,
     Sashelp.Fsp.Collection.Class col3;
 DCL num arr(3);
 DCL Char abc = 'def', Num def = 3;
 DCL List l = {1,2,sublist=('a','b'),num=3} ;
 if def = 3 then
   Do;
     DCL num local1;
     local1 = 0;
   End;
 m1: Method;
     DCL Num Local2;
     endMethod;
```

(5) Dot notation to simplify method calls and get/set of object attributes.

```
   obj.attr = 3;
   DCL Num  n1 = obj.attr;
   obj.method(a, b, c);
   obj.attr1.attr2.attr3.attr4 = 3;
   DCL Num n=obj.m1(a,b).attr2.mX(obj.attrX);
 /* For new V7 frame control, you can use
  * control name on dot directly */
 checkBox1.label = 'myLabel';
 /* For V6 frame control, you have to use
  * _getWidget method to get the objId first
  */
 dcl object objId;
 _frame_._getWidget('obj2', objId);
 objId._setlabel('myLabel');
```

(6) The Generic ProgramHalt Handler for Robust SCL programming.

The program Halt Class is designed to handle unexpected runtime errors. It contains methods that will be called when certain runtime exceptions occur. By overriding these methods, the user can gain control over how the exceptions are handled. For instance, the method _onZeroDivide will be called if a divide-by-zero occurs. The user can override _onZeroDivide

as follows, and perform any necessary actions inside the overriden method:

```
Class Work.A.Error.Class extends
      Sashelp.Classes.Programhalt.Class;
_onZeroDivide:method / (STATE='O');

   endmethod;
EndClass;
```

The halt handler gives SCL developers the capability of determining whether an application should continue executing or whether it should halt immediately. The halt handler saves any error messages generated by SCL and stores them in an SCL list. This list can then be mailed to the associated SCL developers, notifying them of potential problems in their code.

The SAS Component Object Model(SCOM)

Classes are the foundation of the SAS Component Object Model(SCOM). They contain definitions of methods, attributes, events, eventHandlers and interfaces. There are two ways to construct a class. You can use either the AF Class Editor or SCL class syntax to construct the class entry. The Class Editor provides a way of creating a class from a GUI (with table and tree views), while SCL class syntax provides a language-based method for constructing a class. The Class Editor is useful for obtaining a graphical view of a class and is appropriate for making simple changes to the class. However, larger changes that may require a great deal of editing (such as adding or deleting the signatures for several methods) are more easily achieved via the SCL class syntax. This paper will focus on this latter, language-oriented method of creating and modifying a class - using the SCL class syntax.

Converting a class entry to SCL class syntax using the CreateSCL fucntion

The quickest way to view the contents of a class entry is to use the new ClassToSCL function which can be used to convert any existing class (version 6 or version 7) into SCL class syntax. An example of the ClassToSCL function,

```
classId
= ClassToSCL('yourLib.yourCat.yourClass.class',
             'work.a.yourSCL,scl',
             'anyDesc');
```

This classToSCL function converts the yourLib.yourCat.yourClass.class to another SCL entry work.a.yourSCL.scl with the the description of anyDesc. If you open the entry work.a.yourClase.scl in the AF BUILD environment, you will see the attributes and methods defined for this class. This same thing can also be achieved through the AF Class editor by using the Save As command and then choosing the SCL entry that will contain the SCL class syntax of the converted class entry.

Constructing a class

You can construct a new class entry by using SCL Class syntax and the SaveClass command. A simple example of the class syntax is shown below,

```
 Class simple extends myParent;
```

```
Public Num num1;

m1: Method n:Num Return=Num
    / (SCL='work.a.uSimple.scl');
m1: Method Return=Num;
      num = 3;
      DCL Num n = m1(num);
      Return(n);
    EndMethod;
EndClass;
```

Entering the above SCL into the entry work.a.simple.scl and using the SaveClass command (or the Save as Class pmenu entry from File pmenu) will compile this SCL entry and store the resulting class in the new class entry work.a.simple.class. If the entry work.a.simple.class already exists, it will be erased and replaced by the new generated class. This new class 'simple' specifies work.a.myParent.class as its parent class by using the 'extends' clause. If you do not specify an 'extends' clause, the SCL compiler will assume Sashelp.Fsp.Object.Class is the default parent class. The '/' delimiters are used to provide optional information for the class, attributes or methods.

Defining Method Scope: Public/Private/Protected

All methods in version 6 were public methods, which means they could be accessed anywhere in your application. In version 7, SCL provides more flexibility regarding method scope. Public methods are the same - they can be inherited by subclasses and accessed anywhere the corresponding object exists. Private methods can only be accessed by other methods in the same class. They will not be inherited by a subclass. Protected methods can be accessed by the methods in the current class and any subclass. The Public/Private/Protected keyword must be listed before the keyword Method. The default is Public method.

```
Class scope;
  m1: Public Method n:Num Return=Num
      / (SCL='work.a.uScope.scl');
  m2: Private  Method :char
      / (SCL='work.a.uScope.scl');
  m3: Protected Method Return=Num;
      num = 3;
      DCL Num n = m1(num);
      Return(n);
    EndMethod;
  m4: Method; endMethod;
  EndClass;
```

The method m4 in the above class 'scope' is by default a public method.

Defining Parameter Types using the ":" operator

In Version 6, a parameter type can only be numeric and character. In version 7, a parameter type can be either a primitive data type such as numeric, character, list and object type, or a user-defined data type such as the class type or the interface type.

```
Import Sashelp.Fsp.Collection.Class;
 Class colon;
  m1: Method n:Num c:Char Return=Num
      / (SCL='work.a.uColon.scl');
  m2: Method s:Collection o:Object
```

```
      / (SCL='work.a.uColon.scl');
  m3: Method n(*):List
      / (SCL='work.a.ucolon.scl');
EndClass;
```

Defining Parameter Storage using Input/Ouput/Update

In version 6, parameter storage is always Update. This means the value of the caller's argument will be copied into (COPY-IN) the caller's corresponding parameter when the method is invoked. Then when the endMethod statement is executed, the value of the callee's parameter will be copied back (COPY-OUT) to the corresponding caller's argument. Some flexibility in this behavior has been introduced in version 7. Parameter storage can now be defined as Input and Output as well as Update. In SCL class syntax, this is done by using the symbol :I, :O or :U after the parameter name but before the :type (:I is input, :O is output, and :U is update). By using an Input parameter, you can avoid the COPY-OUT from the callee's parameter to the caller's argument when the endMethod statement is executed. By using an Output parameter, you can avoid the COPY-IN from the caller's argument to the callee's parameter when the method is invoked.

```
Import Sashelp.Fsp.Collection.Class;
 Class storage;
   m1: Method n:I:Num c:O:Char Return=Num
       / (SCL='work.a.uStorage.scl');
   m2: Method s:Input:Collection
       / (SCL='work.a.uStorage.scl');
   m3: Method s:Object
       / (SCL='work.a.uStorage.scl');
   m4: method
       / (SCL='work.a.uStorage.scl');
 EndClass;
```

Defining Return Type using Return= clause

One of the major complaints regarding the method syntax for version 6 SCL was the lack of a Return value. This issue has been addressed in version 7. You can now use RETURN=type in the method declaration to specify the return type. For example, in the previous class 'scope' example, the m3 method has Numeric return type. The existing Return statement in version 6 has also been enhanced with the following syntax:

```
Return(expression);
```

If a method has a return type, the Return(expression); statement is required in the associated method statement block. The type in the Return= clause must match the type of the expression in the Return (expression); statement. In the previous class 'scope' example, the method m3 has a return(n) statement, where n is numeric and which matches Return=Num specified in the method declaration.

Method Signatures and Method Overloading

Method overloading is the process of generalizing a method's name to include the method parameter list. This allows the user to create methods with the same basic name, but whose parameters differ in number, type, or both. This is especially useful for creating methods that are related conceptually, but have different parameter interfaces. The method signature is a datum representing a method's parameter types. It is basically shorthand for the method's type interface, and is used by the SCL

compiler (and runtime) to distinguish the different forms of a given overloaded method. In the following example, there are four overloaded methods. The first (public) m1 method has the signature (N)C where C is the return type for the method. The second (private) method m1 has the signature ([C)V where [C means an array parameter with char type, and V stands for Void which means there is no return value. The third (protected) method m1 has the signature ()V, which means it has no parameters and does not have a return value. The fourth m1 method has the signature (NC)V (which means it takes a numeric and a character and does not have a return value).

```
Class sig ;
   m1: Method n:Num return=Char;
             / (scl='work.a.uSig.scl');
   m1: Private Method N(*):Char /*array*/
             / (scl='work.a.uSig.scl');
   m1: Protected Method
             / (scl='work.a.uSig.scl');
   m1: method N:Num C:Char;
       endMethod;
   m4: method / (Signature='N');
         /* any SCL statement ..    */
       endMethod;
EndClass;
```

In version 6, there were no method signatures. There could only be one method with a given name. In version 7, this is equivalent to having the Signature='N' option set for a method, which indicates that there is no signature for the method (and so it cannot be overloaded). An example of this is the method m4 in the above example.

Method State and Overwritten Methods

Method definitions can be either new methods or "Overwritten" methods. A new method is one whose name and signature do not exist in any parent of the given class. An overwritten method is one which overrides a method with the same name and signature in a parent class of the current class. The _init method in the following example is an "Overwritten" method (which is indicated by state="O"). All other methods (which have no State= options) are "new" method methods by default. It should be noted that while the AF Class Editor will display all methods for a class, including all of its parents' methods, it is not necessary (and is in fact an error) to define parent methods in a given class's method statement block. Only those methods belonging to that particular class should appear there, not any of its parents methods. To improve readability, the naming convention for V7 system-provided methods is - capital letters should be used to separate compound words, and there should be only one "prefix underscore". AF/SCL users cannot create methods with "prefix underscore".

```
Class state ;
     m1: method N:Num C:Char;
         /* - - Any SCL statements - - */
         endMethod;
     m2: method N:Num return=Num;
             /* any SCL statement ...   */
          Return(3);
      endMethod;
     _init: Method / (State='O');
            _Super();
            /* Any SCL statements */
            EndMethod;
EndClass;
```

Other Method Options

Method implementations can be defined either inside the class statement block or outside the current SCL entry. You can use the option SCL=sclEntryName to specify the location of a method implementation. Usually, there is no need to distinguish the method name and label name, but if necessary you can use Label=option to specify a different label name for a method. The Enabled= option can be applied to limit the usage of a method - specifying Enable='N' makes a method inaccessible.

```
Class others ;
     m1: method N:Num C:Char
       / (SCL='work.a.others.scl,
          Label='mylabel',
          Enabled='N');
EndClass;
```

Creating Method implementations Using Class/UseClass Statement Block

The object-oriented model in version 6 applied a "dynamic binding" approach which associated the class meta information with the SCL method information during run-time. This late binding approach provided extensive flexibility in such areas as method delegation and per Instance methods, but a price was paid in performance because of the runtime association of meta and SCL information. Observations indicated that for the majority of class methods the meta information could be matched during compile-time. This concept of "Static-Binding" not only improves runtime performance by pushing the meta-SCL checks back to compile time, but also allows for the earlier (compile-time) detection of some errors. This approach requires that method implementations be coded inside a Class or UseClass statement block. The UseClass statement block is syntactically similar to the Class Statement block, with the exceptions of disallowing construction of class attributes and class events in UseClass. SCL users can directly reference attributes in a Class or Useclass statement block without specifying the object (similar to how regular SCL variables are referenced). Similarly, method calls can be coded without the object reference (just like regular function calls). This short-cut programming style tremendously improves the code readability and maintainability.

Approach 1: Method implementations directly coded inside a Class Statement block:

```
 Class one ;
    Public num sum;
    _init: Method / (State='O');
              _Super();
              sum = 0;
            EndMethod;
    sum: method N:Num Return=Num;
              sum = sum + N;
              return(sum);
           endMethod;
    sum: method N1:Num N2:Num
                 Return=Num;
              sum = sum + N1 + N2;
              return(sum);
           endMethod;

EndClass;
```

Approach 2: Method implementations coded inside a UseClass statement block (which is a separate SCL entry from the entry where the Class Statement block is defined).

Coding the following scl program in work.a.one.scl,

```
Class one ;
    Public num total;
    _init: Method
         / (State='O',
            SCL='work.a.uOne.scl');
    m1: method N:Num Return=Num
         / (SCL='work.a.uOne.scl');
    m1: method N1:Num N2:Num
                Return=Num
         / (SCL='work.a.uOne.scl');
EndClass;
```

Coding the following SCL program in work.a.uOne.scl,

```
UseClass one ;
    _init: Method / (State='O');
             _Super();
             total = 0;
           EndMethod;
    m1: method N:Num Return=Num;
             total = total + N;
             return(sum);
          endMethod;
    m1: method N1:Num N2:Num
                 Return=Num;
             total = sum(N1, N2) + m1(n1);
             return(sum);
          endMethod;
EndUseClass;
```

The first approach is simple and clean. It is appropriate for a small project where all the class methods are maintained by one or two developers. However, for a larger project which involves several developers maintaining a specific class, the second approach will probably be more appropriate. Each developer maintains a separate SCL entry which contains a UseClass statement block for the methods he or she is responsible for.

Comparing the method implementation style with the style used in version 6, we find there are several advantages to the Class/UseClass statement block:

(1) Parameters belong to the method scope.

(2) Method name Overloading

(3) Short-cut notation

(4) Error Detection in Compile-time

(5) Link label is not allowed.

(6) _Super call

(7) Performance Improvements

Instantation of a Class with the _NEW_ Operator and Importing a Class

To instantiate the class 'one' from the previous example in an application, the _NEW_ operator must be used. The _NEW_ operator is equivalent to _NEW_ method defined in the Sashelp.Fsp.Object.Class in version 6. To allow abbreviated class name usage in your SCL entry, you can use the Import statement. The Import statement defines class entry search rules (during compile-time), and allows shortened (typically one and two-level) names to be used for classes. By fully qualifying a class name in an Import statement, you can then use its one-level name anywhere else in the program. You can also use Import to specify a catalog in which to search for abbreviated class name entries.

```
Import work.a.one.class;
Init:
  /* short-cut notations of
   * DCL work.a.one.class
   *  obj= work.a.one.class();
   */
   DCL  one  obj  = _NEW_ one();
   DCL  Num  sum1 = obj.sum(3);
   Put sum1=;
   DCL  num  sum2 = obj.sum(3, 4);
   Put sum2=;
   Return;
```

The above example should print sum1=3 and sum2=10.

The above example should print sum1=3 and sum2=10.

Creating Class Attributes to Replace V6 Instance Variables

In comparing the SAS Component Object Model(SCOM) with typical object-oriented programming models, one finds that the major distinguishing feature is the attribute concept in SCOM. Most object-oriented programming languages do not provide an attribute concept in their language syntax, but require class developers to construct a class for each attribute. In SCOM, this is handled automatically. In the typical object-oriented language, the developer is forced to design methods to handle the actions of getting and setting an attribute, not to mention any associated actions that may be required. In the SAS Component Language, we have simplified these potentially complex steps by combining the concepts of class variable and attribute class construction into a simple SCOM attribute. In particular, SCL allows the user to get/set attributes by using dot syntax, and implicitly handles such actions as attribute value validations and event driven custom access method invocation (the associated actions mentioned above). By handling many of these actions automatically, and removing the burden from the user, the attribute concept greatly simplifies the AF/SCL object-oriented coding process.

Defining the Attribute Scope(Public/Private/Protected) and the type

Just as SCOM class methods have scope (PUBLIC, PRIVATE, PROTECTED), so do SCOM class attributes. By default, the attributes are assumed public. Public attributes can be accessed anywhere inside the application. Private attributes can only be accessed inside the current class. Protected attributes can be accessed by the current class and its subclass. The type of the attribute (which is given immediately following the scope) can be a primitive basic type (Num, Char, List, Object), or a

user-defined data type(Class or Interface). Attribute can also be defined as arrays.

```
Class myAttr;
  Public Num n;
  Private Char(20) c(3);
  Protected List l;
  DCL   object o;
EndClass;
```

The above example shows the attribute o has public scope by default. The private attribute c is a character array of three elements with each element 20 characters in length.

Autocreate for SCL list

When a class is instantiated, the storage for each attribute is also created. This includes attributes which have list type and four-level class type. The optional clause AutoCreate='Y' (which is the default) specifies that list attributes and class attributes will be created automatically (i.e. the listId or classId will be created). However, an attribute which has the generic object type will not be created (because the specific class is unknown). Conversely, the optional AutoCreate='N' will specify that list and class attribute should not be created when the class is instantiated. If this option is used, it is the user's responsibility to create any associated list or class attributes after the class is instantiated.

```
Import Sashelp.Fsp.Collection.Class;
  Class myAttr;
    Public List l1
      / (AutoCreate='N');
    DCL List s1;
    Public Collection c1;
    Public Collection c2
      / (Autocreate='N');
  EndClass;
```

The above example shows attributes s1 and c1 have the default AutoCreate='Y', and attributes l1 and c2 have the explicit AutoCreate='N'.

InitialValues and SCL List Initialization Syntax

Like SCL variables, all numeric attributes will be implicitly initialized to missing values and all character attributes will be implicitly initialized to blank strings. Attributes can also be explicitly initialized by using the optional initialValue= clause. List attributes can be initialized using the convenient and powerful SCL list constant syntax (which includes support for nested lists). To improve performance, the user should initialize values whenever possible.

```
Import Sashelp.Fsp.Collection.Class;
  Class myAttr;
    Public Num n1
      / (InitialValue = 3);
    DCL Char c1
      / (InitialValue = 'abc');
    DCL list  list1
     /  ( initialValue=
        {
          COPY = {
               POPMENUTEXT='Copy here',
               ENABLED='Yes',
```

```
            METHOD='_drop'                } ,
         MOVE = {
            POPMENUTEXT='Move here',
            ENABLED='Yes',
            METHOD='_drop'                } ,
         LINK = {
            POPMENUTEXT='Link here',
            ENABLED='Yes',
            METHOD='_drop'               }
       }
      );
  public list  list2
     / (initialValue=
       { 1, 2, 'abc', 'def' }
      );
EndClass;
```

ValidValues Syntax and Value Delimeters

The optional ValidValues= clause specifies that an attribute value is validated when dot syntax or short-cut syntax is used. If the valid values clause contains more than one value, you should use a space as the delimiter to separate valid values. If the valid values contain space, then you should use '/' as the alternative delimiter to separate the valid values.

```
Class business_graph_c;
 Public  Char statistic
    / (Validvalues=
 "Frequency/Mean/Cumulative Percent"
     )
    ;
 Public  Char highlightEnabled
    / (Validvalues="Yes No");,

EndClass;
```

Set/Get Custom Access Method(setCAM and getCAM)

The Custom Access Method (CAM) is a user-defined method that allows implicit actions to be associated with the getting and or setting of attributes. The CAM method must provide one parameter which stores the current value of the attribute. In the following example, the attribute A has the optional getCAM='M1' method specified. This means that the user-defined SCL method M1 will automatically be invoked after every fetch of the attribute A. Similarly, the method M2 will be invoked after every set of the attribute B.

```
Class CAM;
 Public  Char A
    / (getCAM='M1');
 Public  Num B
    / (setCam="M2");,
 M1: Method c:Char;
       put 'In m1';
     EndMethod;
 M2: Method n:Num;
       put 'in m2';
     EndMethod;
EndClass;
```

Version 6 Compatability: IV= and PureIV=

The object model in version 6 contains instance variables. These have been replaced in version 7 by attributes. In order to provide compatibility, the class loader will automatically convert version 6 class formats to the new version 7 SCOM format when the class is first loaded. This step includes converting all instance variables into protected attributes with the option IV='ivName'. An exception is pure instance variables, which will not be directly converted to attributes. To ensure compatibility, however, we've provided the optional clause pureIV='Y'. This directs the SCL compiler to construct the _IV_ list inside the class. In the following example, the variable B is not actually an attribute - it is a numeric list item stored in an _IV_ list. The attribute A has been converted from the instance variable 'abc'.

```
Class IV;
 Public  Char A
    / (IV='abc');
 Public  Num B
    / (PureIV="Y");,

EndClass;
```

Other Attributes Properties: Editor/Category/Linkable

Attributes can also be declared with the optional Editor='editorName' clause, which is used to specify the default editor (a frame program) to be used when the initialValues or validValues with ... options are invoked in the class editor or property sheet. The optional category='name' can be used to group attributes into a specific tree node inside the tree view when the class editor is displayed. The optional Linkable='Y|N' is used for attribute linking, but the linkTo information can only be specified from the Frame property sheet.

```
Class other;
 Public  List A
    / (Editor='myListEditor',
       Category='List');
 Public  list B
    / (Category='List');,

EndClass;
```

Creating User-Defined Events and EventHandlers

In SCOM, you can programatically design user-defined Events and EventHandlers. Events are essentially generic messages that are sent to objects from the system or from SCL applications. For example, these messages may direct an object to perform some action (e.g. run a method) based on the occurrence of some event. EventHandlers are methods that are invoked when a certain Event occurs. Attributes can be declared with the optional clause SendEvent='Y|N'. The default is SendEvent='Y' which implies that an attribute event definition will be automatically created when the attribute is created. This means that when that attribute is accessed via dot or short-cut syntax, an attribute event will be triggered. This event can be received by using an EventHandler, which can be created by using the new EventHandler syntax in the class statement block. An EventHandler is a essentially a 'listener' method which is executed whenever the associated event occurs. A user-defined event can be triggered by using the _sendEvent method. User defined events can be constructed using the EVENT syntax inside the class statement block.

For readability and for compatibility with version 6 events, the event name is a string. The following ehclass example shows two events which will be created when the class is constructed. One is the system-provided event for attribute n which has event name "n Changed" (SCOM will always create the suffix "Changed" after the attribute name to distinguish between system-provided events and user-defined events). The other event is a user-defined event and is called "myEvent". There are three eventHandlers defined - m1, m2 in ehclass.scl and m3 in ehclass1.

Edit the following program in work.a.ehclass.scl, using the Saveclass command to save the class.

```
class ehclass;
  public num n;   /* system event */
  event 'myEvent'
       / (method = "m2");

  eventHandler m1
       / (sender = "_SELF_",
          event='n Changed');

  eventHandler m2
       / (sender = "_SELF_",
          event='myEvent');

   m1: method a:list;
       put "Event is triggered by attribute n";
       endmethod;

   m2: method  a:string n1:Num n2: Num ;
         put "event is triggered by _sendEvent";
         put a= n1= n2=;
       endmethod;
endclass;
```

Edit the following program in work.a.ehclass1.scl, using Saveclass command to save the class. This class has an eventHandler m3 which listens for myEvent defined in the previous class. The sender='*' means the sender is determined at run-time.

```
class ehclass1;
  eventHandler m3
       / (sender = "*",
          event='myEvent');

   m3: method  a:string n1:Num n2: Num ;
         put "event myEvent is defined in
              another class which
              is triggered by _sendEvent";
         put a= n1= n2=;
       endmethod;
endclass;
```

The following test program will demonstrate event triggering for attribute and user-defined events. The m1 method in ehclass will be invoked first, followed by the m2 method in the ehclass, followed by the m3 method in the ehclass1.

```
import work.a.ehclass.class;
import work.a.ehclass1.class;
init:
   dcl ehclass obj = _new_ ehclass();
```

```
dcl ehclass1 obj1 = _new_ ehclass1();

/* (1) triggers the attribute event */
obj.n = 3;
/* (2) triggers the user-defined event */
obj._sendEvent("myEvent", 'abc', 3, 4);
return;
```

Implicit Actions When Executing Dot Syntax or Short-Cut Syntax

The SCL compiler will parse the short-cut syntax inside Class or UseClass statement blocks and generate internal SCL intermediate code as if regular dot syntax (with object references) had been used. In other words, the internal representations of short-cut and regular dot syntax are the same. Additionally, when an attribute is referenced via dot syntax (or equivalently via short-cut syntax), a get/set-attribute value method call is generated to perform the get/set. That is, the internal representation of dot syntax for attributes is a method call. Due to the complexity of the get/set attribute value method call, we will discuss the following associated actions:

Execution Sequence for Get/Set AttributeValue

For getAttributeValue, the attribute value is determined in the following way:

```
(1) If a getCAM is defined, then it is executed to determine the value.

(2) If the value has been set previously, then it is returned.

(3) If the value has not been set, then the class initial value is returned.
```

For setAttributeValue, the sequence of actions is as follows:

```
(1) Check if the value is contained in the valid values list (if the valid val
    metadata is an SCL entry, it is executed first to get the list of values t
    check against).

(2) Run the setCAM if it is defined (gives the user a chance to perform additi
    validation and process their own direct side effects)

(3) Store the given value on the object.

(4) If the send event metadata is set to 'Yes', then the '<attribute name> cha
    event is broadcast.
```

Object-Oriented Design Using Interface Model

The fundamental object-oriented design in SCL is based on public methods that can be invoked on objects. An alternative to this is to use something called Interfaces. An Interface typically consists of a set of definitions of abstract methods, which are essentially just prototypes for methods whose implementations are given in some other entry. An Interface contains only the information needed to call a method, it does not contain the actual implementation itself. An interface can be viewed is an expression of pure design, whereas a class is a mix of design and implementations. The syntax of an SCL interface is similar to the syntax of an SCL class. The Interface entry is a new catalog entry which has the entry type INTRFACE. You can use the previous mentioned classToSCL function to convert an INTRFACE entry into SCL Interface syntax and then use the SAVECLASS command to compile an SCL entry into INTRFACE entry.

Edit the following program in work.a.intface1.scl and use the SAVECLASS command to create the work.a.intface1.intrface entry. Methods defined in the interface statement block cannot contain method implementations. They cannot specify the SCL= clause either.

```
Interface intface1;
  getValue: Method return=Num;
  setValue: Method n:num c:string return=num;
EndInterface;
```

Edit another program in work.a.intface2.scl and use the SAVECLASS command to create the work.a.intface2.intrface entry.

```
Interface intface2;
  getName: Method return=Num;
  setName: Method n:num c:string return=num;
EndInterface;
```

Unlike SCOM classes, interfaces support multi-inheritance. The following interface intface3 inherits both intface1.intrface and intface2.intrface through the Extends syntax. The intface3 inherits all the methods defined in intface1 and intface2 with an extra "new" method m1b defined in the interface.

```
Interface intface3 Extends intface1, intface2;
  m1b: Method return=Num;
EndInterface;
```

Supported Interface and Required Interface

Interfaces describe "contracts" in a pure, abstract form, but an interface is interesting only if a class supports it. Two related concepts - Supported Interface and Required interface - can be used to simplify the design of the model/viewer environment. For example, the following class "model" supports the previous interfaces "intface1". and "intface2". The class "model" must implement all the methods specified in all supported interfaces.

```
Class model supported intface1, intface2;
  getValue: Method return=Num
      / (scl='work.a.uModel.scl');
  setValue: Method n:num c:string return=num
      / (scl='work.a.uModel.scl');
  getName: Method return=Num
      / (scl='work.a.uModel.scl');
  setName: Method n:num c:string return=num
      / (scl='work.a.uModel.scl');
endclass;
```

For an example of a Required interface, assume we have several models (say listModel class, SCLArrayModel class, and SASDataSetModel class) which support the interfaces "intface1" and "intface2". Assume further that we have the following class 'viewer' which has the Required interfaces "intface1" and "intface2". This means any of the above models can be used in conjunction with the class viewer (for instance, used as a parameter to a method in 'viewer'). By specifying which interfaces are Required, you allow the compiler to generate information that will be used to validate whether the actual model used at runtime matches the required interface.

```
Class viewer required intface1, interface2;
```

```
  /* - - Other implementations- -  */
endclass;
```

GUI Functions Enhancements

(1) new MessageBox function: allows the SCL user to call the host message bo
within SCL applications.

```
lid = makelist();
rc=insertc(lid,"Type:"||type);
rc=insertc(lid,"Line:"||lineNumber);
rc=insertc(lid,"Entry:"||entry);
rc = messageBox(lid);
```

(2) new OpenSasFileDialog, SaveSasFileDialog, OpenEntryDialog, and
SaveEntryDialog functions: OpenSasFileDialog and SaveSasFileDialog
functions are similar to the existing DIRLIST function which can open
and saveas all kinds of SAS files(two levels) with some user customized
capability. OpenEntryDialog and SaveEntryDialog functions are similar to
the existing CATLIST function which can open and saveas SAS catalog
entries(four levels) with some user customized capability.

(3) new GUI interface for SCL selection list functions: Most of the selector
new GUI look.

(4) new Dialog routine: provides a real modal application environment. It is
to CALL DISPLAY except that all other SAS windows will be disabled as DI
brings up the specified frame window. Note the DIALOG routine can also b
used as a function if the specified frame entry returns a value. The DI
routine/function can only be used to display a FRAME entry.

Non-GUI Functions/Classes Enhancements

(1) new IcCreate, IcDelete, IcValue and IcType functions:
allow users to create/delete/query the integrity constraints
to guarantee the correctness and consistency of the SAS data.

(2) new InitRow function: initializes the Table Data Vector to missing valu
This is useful when you are doing an APPEND with the NOINIT option while
you only want to explicitly set some of the table columns in the Table D
Vector, but not all of them.

(3) new Dcreate: allows the SCL user to create a new directory.

(4) new NameDivide and NameMerge functions: The NAMEDIVIDE function divides
a 2, 3, or 4-level SAS name and returns how many pieces are in
the name as well as each individual information in the variables
such as lib, cat, mem, type. The NAMEMERGE function is the opposite of
the NAMEDIVIDE function which takes the two, three, or four pieces of
a valid SAS name and concatenated them into a valid 2, 3, or 4-level
SAS name.

(5) new CompareList function:
gives the SCL programmer the ability to programmatically determine
if two lists contain the same information.

(6) New GetItemo, GetNItemo, SetItemo, SetNItemo, Inserto, Popo and

```
        Searcho functions: allow manipulating any list items of the new object
        data type.

    (7) Delete and Rename functions: both now support the FILE type.

    (8) new attribute name NLOBSF for Attrn function: returns the number of rows
        with WHERE clause being applied.

    (9) Lock function: no longer needs to run under SAS/Share.
```

SCL Debugging and Analysis Tools Enhancements

SCL debugger supports the dot syntax:

```
debug> examine obj1.attr1

debug> set obj.attr2.attr2 = 1

debug> describe a.b
```

Monitoring the SCL list contents changes

```
debug> calc setlattr(obj1, 'NOUPDATE')
```

The SCL Coverage Analyzer is a white-box run-time testing tool which measures SCL statement coverage, and can be used to verify the completeness of a test library. The SCL developer can use this tool to determine the frequency with which specific statements are executed, and add tests to the test library accordingly. Please reference the SCLPROF command under the release 6.12 to find related information.

Verifying Year 2000 Problems using SCL Static Analyzer

As the software industry's annus miserabilis approaches, much time and effort is being devoted to the problem, "Is My Application Year 2000 Compliant? ". For SCL application writers faced with such difficulties, the SCL Static Analyzer (which has been in production since 6.12) can prove to be an effective tool for identifying potential problems. SAS formats are saved internally as numeric doubles - which precludes any Year 2000 problems. Problems arise, however, when programmers convert the data into an mmddyy-related format and use the year information (2-digits) to compare with other data. The collection phase of the SCL Static Analyzer can be used to scan all SCL programs in a project to find every SCL source statement which references a SAS format/informat function, an INPUT/PUT related function, a SUBMIT, or simply a SAS data set. This information can be used as a starting point to identify any possible date format problems.

Conclusions

The SAS Component Language in version 7 contains many new features that result in more power and flexibility for the user. The old object-oriented programming style and its programming difficulties have been re-evaluated. The new object-oriented programming styles and language constructs have been illustrated. We encourage SCL users to fully apply the new object-oriented programming style to allow the SCL compiler to catch errors at compile-time rather than having them

deferred until run-time. With these new changes and enhancements, a user-friendly object-oriented application system will be much easier to build and maintain.

1. SAS Component Language: Reference Version 7(will be published this year)

Produced Tue Nov 3 13:11:53 EST 1998 by pub2html