Paper 1823-2014

Let SAS® Power Your .NET GUI

Matthew Duchnowski, Educational Testing Service

ABSTRACT

Despite its popularity in recent years, .NET development has yet to enjoy the quality and depth of statistical support that has always been provided by SAS®. And yet, many .NET applications could benefit greatly from the power of SAS and, likewise, some SAS applications could benefit from friendly graphical user interfaces (GUIs) supported by Microsoft's .NET Framework. The author aims to outline the basic mechanics of automating SAS with .NET and provide some specific strategies for maintaining parallelism between the two platforms at runtime. This will include embedding SAS scripts within a .NET project and managing communications between the two platforms. Specifically, the log and listing output will be captured and handled by .NET, and user actions will be interpreted and sent to the SAS engine. Visual Basic .NET is the author's language of choice, but an identical approach can be executed using C# syntax.

INTRODUCTION

This paper suggests a set of strategies for integrating .NET and SAS, but certainly not the only set in existence. The author discusses automating SAS 9.3 from a Windows Form developed within Microsoft Visual Studio 2008. The .NET code depicted in these illustrations is Visual Basic. Using these strategies the reader may, with few alterations, find success in automating other versions of SAS with other .NET solution types. The implementation can also be carried out with other .NET languages, most probably C#, and using other versions of Microsoft Visual Studio.

Regardless of the user's tool set, the goal here is to explore strategies for organizing and carrying out SAS/.NET integration so that the best of SAS and .NET can be combined for a powerful application.

DO YOU REALLY WANT TO DO THIS?

Automating one language using another is inherently glamorous to many programmers and therefore deserves some scrutiny. Integrating two languages might showcase your prowess as a developer, but does it present the best solution for your purpose? Here are some questions you should ask yourself before going forward:

Can this task be done just as well with one language alone?

The benefit of language integration comes from drawing on the best features of multiple languages. If one language has the power to accomplish all of your goals, pursue that solution. In particular, don't limit yourself by what you currently know. This might be the time for you to finally learn that set of tools that you've been avoiding.

Is this task fairly static?

In particular, if you are trying to "wrap" .NET code around a library of existing SAS scripts, it is recommended that these scripts be stable and time-tested. If the SAS code is particularly prone to receiving updates and changes, then managing the integration of the new code into the existing .NET solution might make more work than you are ready to take on.

Can you support your users?

You may find you don't have the necessary resources to support your users. For example, users may not have access to the version of SAS required by your application. If SAS privileges are handled by a separate entity, you might find it frustrating when you want to share your application with users who can't make use of it due to reasons beyond your control.

Generally speaking, if a single solution possesses multiple technologies integrated into one, there is an increased potential for end user configuration issues. These issues of course grow larger when you have a large number of users.

SETTING UP YOUR WINDOWS FORM PROJECT

To reproduce the code shared in this document, users of Microsoft Visual Studio first may need to create a new project (preferred) or open an existing one. Automating the SAS engine will rely on several dynamic-link library files (DLLs) that come standard with every SAS 9.3 installation. These files usually can be found in the "Program Files" under the SAS installation directory (see *Integration Technologies* folder). Readers who experience trouble finding these files should check with their system administrator.

The following three files will be required for our .NET integration:

- Interop.SAS.dll
- Interop.SASIOMCommon.dll
- Interop.SASWorkspaceManager.dll

It is recommended that these files be copied to the target Visual Studio project stored within their own folder. This can be done by right-clicking the project in the Solution Explorer and **Add → New Folder**. After creating the folder, title it appropriately (e.g "dlls") and deposit a copy of all reference files.

At this point, the reference files have not yet been added to the project. Adding the resources is a matter of clicking **Project → Add Reference ...** . Then, by choosing the **Browse** tab, the user can navigate to and select these three files.

Access to SAS objects is enabled through appropriate import statements located at the top of a .NET class or module:

```
Imports SAS
Imports SASWorkspaceManager
```

MAINTAINING PARALLEL ENVIRONMENTS

We are now ready to automate SAS from .NET, but before we do, we should give some thought to the underlying principles in our design.

Integrating two programming languages can be tricky and it is recommended that the code from each language be organized as independently from the other as possible.

One basic way to achieve this separation is to create a class in the Visual Studio project, which we shall call SASApp. This class can be added by selecting **Project → Add Class …** . This class will later handle all communication between SAS and .NET during application runtime.

The code presented below contains everything necessary for initializing, opening and closing an instance of SAS® 9.3 on the user machine. When a SASApp object is created, an instance of SAS will open in the background behind the application. The workspace is also assigned a Language Service object, which submits code to the engine at runtime. For the sake of thoroughness, a carriage return “Chr(13)” and newline character “Chr(10)” are assigned as the default line separators.

```
Option Explicit On

Imports SAS
Imports SASWorkspaceManager


Public Class SASApp

    Dim wsm As WorkspaceManager = New WorkspaceManager()
    Dim Sasworkspace As Workspace
    Dim WithEvents ls As SAS.LanguageService

    Dim MainForm As Form1

    Public Sub New(ByRef AppForm As Form)

        Sasworkspace = _
               CType(wsm.Workspaces.CreateWorkspaceByServer("", _
               SASWorkspaceManager.Visibility.VisibilityProcess, _
               Nothing, "", "", ""), Workspace)

        ls = Sasworkspace.LanguageService
        Sasworkspace = ls.Parent
        ls.LineSeparator = Microsoft.VisualBasic.Chr(13) & "" & Microsoft.VisualBasic.Chr(10)
        MainForm = AppForm

    End Sub

    Public Sub Close()

        Sasworkspace.Close()

    End Sub


    ' [Other Functions and Subroutuines may exist]


End Class
```

The SASApp constructor takes the application’s main form as a parameter and assigns it to the class variable `MainForm`. This will allow any public object on the main form to be accessed from the SASApp class using a simplified syntax (i.e. “MainForm” instead of “My.Forms.Form1”).

For simplicity, it is recommended that the application open one and only one SAS workspace during runtime. This is achieved by creating a SASApp object (called simply “SAS” in the example below) when the main form opens and terminating the workspace when the main form closes.

```
Public Class Form1

        Dim SAS As SASApp

        Private Sub Form1_Load(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) Handles MyBase.Load

                'Start an instance of SAS
                SAS = New SASApp(Me)

                'Submit trivial code segment so as to capture initial SAS log text
                SAS.Submit("")

        End Sub

        Private Sub Form1_FormClosing(ByVal sender As Object, _
                        ByVal e As System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing

                'Close SAS session
                SAS.Close()

        End Sub

        ' [Other Functions and Subroutuines may exist]

End Class
```

Because an instance of SAS will be running invisibly behind the .NET application, users might want to close all other instances of SAS before executing, as doing so will increase system performance and avoid ambiguity. Before instantiating the SASApp object, you may want to implement a check to see if SAS is already running and then either prompt users to close the workspace or completely prohibit the application from running if the condition is not satisfied. Checking for a running instance of SAS can be accomplished with the help of the function below.

```
Public Function IsSASRunning() As Boolean

   If System.Diagnostics.Process.GetProcessesByName("sas").Length > 0 Then
      Return True
   Else
      Return False
   End If

End Function
```

BASIC INTEGRATION TASKS

Now that a SAS workspace is established, SAS code can be submitted by using the SASApp object. For example, a SAS library definition might be desired according to a file path placed within a Textbox control. To submit the code, we have the need for a submit function. For example:

```
        SAS.Submit("libname dir '" & Me.TextBox1.Text & "'")
```

The SASApp class can handle this with the following `Submit` subroutine and `getLog` helper function. Also shown below is the helper function `getListing`, which retrieves the SAS listing output, if so desired.

```
    Public Sub Submit(ByVal sasCode As String)

        ls.Submit(sasCode)

        'Optional - Capture all code submissions to a multi-line textbox 'CodeTextBox'
        MainForm.CodeTextBox.AppendText(System.Environment.NewLine)
        MainForm.CodeTextBox.AppendText(sasCode)

        'Optional - Capture resulting log to a multi-line textbox 'LogTextBox'
        MainForm.LogTextBox.AppendText(System.Environment.NewLine)
        MainForm.LogTextBox.AppendText(getLog())

        'Optional - Capture all code submissions to a multi-line textbox 'CodeTextBox'
        MainForm.OutputTextBox.AppendText(System.Environment.NewLine)
        MainForm.OutputTextBox.AppendText(getListing())

    End Sub

    Public Function getLog() As String

        getLog = Replace(ls.FlushLog(1000000), Chr(12), "")
        If getLog = Nothing Then getLog = ""

    End Function

    Public Function getListing() As String

        getListing = Replace(ls.FlushList(1000000), Chr(12), "")
        If getListing = Nothing Then getListing = ""

    End Function
```

The `submit` routine has some additional features that are optional for the developer. By designating a multi-line textbox `CodeTextBox` on the main form, all code that is submitted to SAS can be captured for the user to view. Similarly, a textbox `LogTextBox` can be designated to capture SAS log with every submission made to the SAS engine. These textboxes can be prominently displayed for the user or kept in a more discrete place (such as a multi-tab view). Regardless of whether the user views this output, the developer will find this information invaluable while coding and debugging.

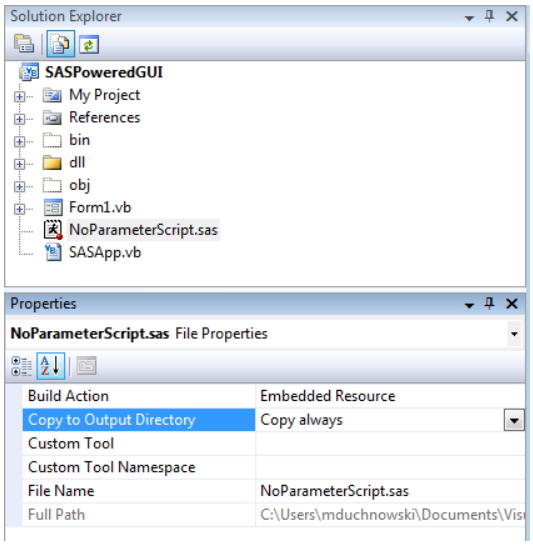
Also included in the `getLog` and `getListing` functions is a String `Replace` function, which attempts to rid the output of the notorious ASCII character that most users will recognize from their copy/paste efforts.

Using these optional functions in conjunction with the `submit` routine (as shown above) will also help the developer and GUI user to experience these actions as they would occur naturally when working in the SAS environment. If the .NET application evolves, the `submit` function can be parameterized later with options for the developer to suppress particular actions when appropriate.

MAINTAIN A SCRIPT LIBRARY

In order to avoid an abundance of `submit` statements littered throughout the .NET code, the developer may wish to maintain a library of SAS® scripts within the Visual Studio Project. Each script can then reside as an embedded project file and will be accessible at runtime.

To do this in Visual Studio, simply add a file by selecting **Project → Add New Item … .** Select "Text File" and change the file extension to "sas". The file can now be edited directly in Visual Studio and accessed by the SASApp object if the settings permit. In order for this access to be granted, it is recommended that, using the Solution Explorer, the "Build Action" be set to "Embedded Resource" and that "Copy to Output Directory" is configured as "Copy Always".



Display 1. Inserting SAS® script as "Embedded Resource" (NOTE: "Copy always" to output directory)

Developing the scripts might be best accomplished using the SAS IDE that comes with SAS. Developing SAS is much easier when key words are highlighted and colored. One downside of keeping the scripts embedded in VB is losing this formatting. The scripts will reside, however, in the .NET project folder and can be accessed directly with the SAS IDE. Once any changes are made and saved outside of Visual Studio, the developer will need to refresh the files in the Solution Explorer before recompiling the .NET application.

An entire script can now be accessed with a SASApp class function `getScript` as shown below. Note that this function requires additional imports to the SASApp class.

```
Imports System.IO
Imports System.Reflection

Public Function getScript(ByVal ScriptName As String) As String

   'Get the current assembly as object
   Dim oAsm As System.Reflection.Assembly = System.Reflection.Assembly.GetExecutingAssembly()

   'Read script using a stream reader
   Dim oStrm As IO.Stream = oAsm.GetManifestResourceStream(oAsm.GetName.Name + "." + ScriptName)

   'Read contents of embedded file using streamreader
   Dim oRdr As IO.StreamReader = New IO.StreamReader(oStrm)

   Return oRdr.ReadToEnd

End Function
```

Running an embedded script without modification can then be done simply as:

```
SAS.Submit(getScript("NoParameterScript.sas”))
```

When deployed, the application will, of course, only have the ability to execute a script that is embedded within it. Depending on the number of users and their location, it might appear advantageous to locate these scripts on a shared drive where they can be accessed at runtime and maintained without re-deploying the application. However, embedding the SAS scripts within the application and making use of a version control system prevents having the library and the .NET solution from becoming unsynchronized.

PASSING PARAMETERS TO SAS SCRIPTS

Most likely, your script will require parameters. If this is the case, one simple strategy is to make use of SAS comment syntax to insert strings into the SAS script.

To illustrate, the following SAS script is structured in this way:

```
/*

        This script runs a linear regression

*/

%let LIB = /*Library*/;
%let DS  = /*Dataset*/;

%let A = /*Regressor*/;
%let B = /*Predictors*/;

proc reg data=&LIB..&DS.;
        model &A. = &B.;
run; quit;
```

Passing the appropriate parameters becomes a matter of replacing the commented syntax within the body of the script. Shown below is an example function in the SASApp class that would execute the script.

```
Public Sub RunRegression(ByVal Library As String, ByVal Dataset As String, _
                             ByVal Regressor As String, ByVal Predictors As String)

   Dim Code As String = getScript("RunRegression.sas.")

   Code = Strings.Replace(Code, "/*Library*/", Library)
   Code = Strings.Replace(Code, "/*Dataset*/", Dataset)
   Code = Strings.Replace(Code, "/*Regressor*/", Regressor)
   Code = Strings.Replace(Code, "/*Predictors*/", Predictors)

   Submit(Code)

End Sub
```

The regression then can be called from the MainForm as in the following manner:

```
SAS.RunRegression("sashelp", "class", "height", "weight age")
```

Obviously, the parameters being passed as code to the SAS engine must all be strings. It may, therefore, be advantageous to create helper functions that interpret form controls as text. These helper functions, if moderate in number, can be stored directly in the SASApp class. Such a function might, for example, interpret the list of selected items in a CheckedListBox and reinterpret them as a delimited string.

```
Public Function getCheckedItems(ByVal CLB As CheckedListBox, ByVal Delimeter As String) As String

   Dim MyArray() As String
   ReDim MyArray(CLB.CheckedItems.Count - 1)

   CLB.CheckedItems.CopyTo(MyArray, 0)
   getCheckedItems = String.Join(Delimeter, MyArray)

End Function
```

GET TO KNOW IOM DATA PROVIDER

There will likely be occasions when information needs to flow from SAS back to the .NET application. This information may take the form of a dataset, but also simply may be a single string stored as a SAS macro variable. The information might be handled as an entire dataset displayed for the user, or it might be a single value known only to the system and be used to dictate the execution of subsequent .NET code.

Regardless of how it is being used, the single most useful technology in this regard is the SAS IOM Data Provider. Every reader who is not familiar with this technology is encouraged to research it separately, as a thorough discussion is beyond the scope of this paper. Simply put, SAS IOM provider allows .NET applications to connect to a SAS dataset by establishing it as a data source. One simple way to draw on the dataset then is by using a DataAdapter object and issuing a SQL select command.

The code below depicts an example of getting all records from a dataset and displaying it in a DataGridView control.

```
Public Sub LoadDataGrid(ByVal Library As String, ByVal Dataset As String, _
                                                          ByVal Grid As DataGridView)
   Dim obAdapter = _
      New System.Data.OleDb.OleDbDataAdapter("select * from " & Library & "." & Dataset, _
      "provider=sas.iomprovider.1; SAS Workspace ID=" & Sasworkspace.UniqueIdentifier.ToString)

   Dim obDS As New System.Data.DataTable
   obAdapter.Fill(obDS)

   Grid.DataSource = obDS

End Sub
```

This routine can be implemented from the main form using:

```
SAS.LoadDataGrid("sashelp", "cars", Me.DataGridView1)
```

USE THE SASHELP VIEWS

If the .NET application needs to access other information from SAS such as metadata or values stored in GLOBAL macro variables, you can make use of the SASHELP library views. A wealth of metadata is stored during each SAS session in the default SASHELP library and can therefore be accessed via IOM provider. In particular, readers are encouraged to familiarize themselves with the views VMACRO, VCOLUMN and VTABLE, as they have particular value for automation tasks.

For example, the .NET application may need a list of all datasets in a particular SAS library. The function below will retrieve this list and return it as a List of Strings.

```
Public Function getDatasets(ByVal Library As String) As List(Of String)

        Dim obAdapter = New System.Data.OleDb.OleDbDataAdapter("select " & _
           "memname from sashelp.vtable " & _
           "where memtype = 'DATA' and Upcase(libname) = '" & Library.ToUpper & "'", _
           "provider=sas.iomprovider.1; SAS Workspace ID=" & _
           Sasworkspace.UniqueIdentifier.ToString)

        Dim obDS As New System.Data.DataTable
        obAdapter.Fill(obDS)

        Dim ColumnValues As New List(Of String)
        For Each r As DataRow In obDS.Rows
            ColumnValues.Add(r("memname").ToString)
        Next

        Return ColumnValues

End Function
```

The List object then can be bound, for example, to a ListBox control located on the main form.

```
Me.ListBox1.Items.Clear()
Me.ListBox1.DataSource = SAS.getDatasets("sashelp")
```

Additionally, global macro variables can be evaluated easily and returned as text using a similar function that queries the VMACRO dataset.

```
Public Function Evaluate(ByVal VarName As String, _
                         Optional ByVal Scope As String = "GLOBAL") As String
   Dim value As String = ""
   Dim obAdapter = New System.Data.OleDb.OleDbDataAdapter("select value " & _
      "from sashelp.vmacro " & _
      "where scope = ' " & Scope.ToUpper & "' " & _
      "and upcase(name) = '"  VarName.ToUpper & "'", _
      "provider=sas.iomprovider.1; SAS Workspace ID=" & Sasworkspace.UniqueIdentifier.ToString)

   Dim obDS As New System.Data.DataTable
   obAdapter.Fill(obDS)

   If obDS.Rows.Count > 0 Then
      value = obDS.Rows(0)(0).ToString
   End If

   Return value

End Function
```

CONCLUSION

As previously mentioned, the set of strategies presented here offer a particular approach to integrating SAS and .NET, but certainly not the only approach. By using some of the methods presented here, programmers can integrate SAS and .NET in order to get the best out of both languages.

ACKNOWLEDGMENTS

I would like to thank all friends and colleagues at ETS who are endlessly willing to discuss topics such as those discussed here, particularly Sarah Venema, who gave valuable feedback on this paper. Thanks to John Bonett for sharing his knowledge of SASHELP Views. I would also like to thank Matthew Campbell for introducing me to the basic concepts of integration through a few discussions that took place many years ago.

I would like to especially thank my wife Robin, who provides me with unending support in all that I do.

REFERENCES

SAS Institute. (2009). *SAS global forum 2009: Table of contents.* Retrieved from support.sas.com/resources/papers/proceedings09/TOC.html

SAS Institute. (2011). *SAS 9.3 Integration Technologies - 9.3.* Retrieved from http://support.sas.com/documentation/cdl/en/itechwcdg/62763/PDF/default/itechwcdg.pdf

Zender, C. L. (2012). *Where's the LISTING window: Using the new results viewer in SAS® 9.3.* Retrieved from http://support.sas.com/resources/papers/proceedings12/250-2012.pdf