TPS Development using the Microsoft .NET Framework

Teresa P. Lopes
Assembly Test Division
Teradyne, Inc.
North Reading, MA USA
teresa.lopes@teradyne.com

Yönet A. Eracar
Assembly Test Division
Teradyne, Inc.
North Reading, MA USA
yonet.eracar@teradyne.com

***Abstract*— Managed or .NET languages are widely used in system software development but are often overlooked for TPS development. This paper looks at using managed languages and the .NET Framework for TPS development. In addition to describing the extensive library support for math, file I/O and string manipulation built into the framework and the powerful application development and debugging support in Visual Studio, the paper also explores how one would define and combine test specific constructs.**

Keywords-TPS Development; Microsoft .NET Framework

I. INTRODUCTION

General purpose programming languages such as ANSI-C and C++ are commonly used for TPS development when test specific languages fall short or where TPS developers and system integrators want to take advantage of the commercial tools available for these languages. This paper explores adding .NET languages, specifically C#, to the list of general purpose programming languages used for TPS development.

The paper starts with an overview of the .NET Framework including a description of the support for legacy technologies. A description of common problems with non-.NET languages and how .NET languages and tools address these problems follows. Finally, the paper concludes with examples of test development in .NET.

II. OVERVIEW OF .NET FRAMEWORK

The .NET Framework [1] is a development and execution environment that allows different programming languages and libraries to work together to build, integrate and deploy software components. It is the next step in the evolution of Microsoft component technology: Win32 -> COM -> .NET. Unlike previous component technologies which were built on top of existing platforms and shoehorned into existing languages, the .NET Framework was designed from the ground up to provide a platform that would make building, managing and deploying Windows-based components easier.

The architecture of the .NET Framework is shown in Figure 1. Microsoft .NET Framework Architecture and the CLR.

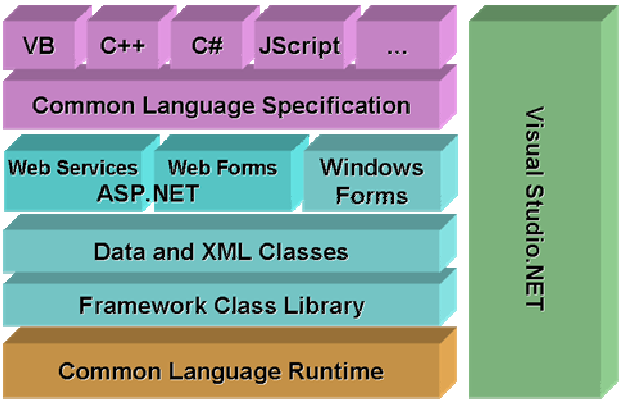

Figure 1. Microsoft .NET Framework Architecture and the CLR

The foundation of the .NET Framework is the Common Language Runtime (CLR). All code in .NET is executed and managed by the CLR. The CLR is analogous to the Java Virtual Machine, except that the CLR always runs fully compiled code; the code is never interpreted. The CLR's main responsibility is to convert code modules compiled into Intermediate Language (IL) into native machine instructions. IL is an abstract, intermediate language that is independent of the original programming language, target machine and operating system that is managed by the CLR. The first time a method is called, the Just-In-Time (JIT) Compiler translates IL to native code. This process is shown in Figure 2. CLR Execution Model

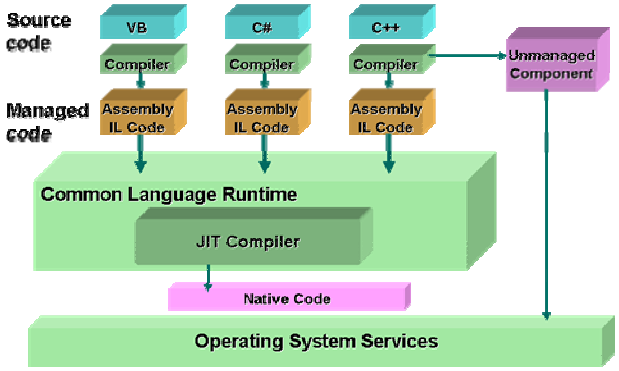

Figure 2. CLR Execution Model

A. Common Language Runtime

The CLR makes extensive use of metadata. Metadata describes the items that make up an application. The metadata is emitted by the compilers. It provides the CLR with complete knowledge of running code. This metadata is a vast improvement over the type libraries used in COM. Metadata can be read and written at runtime. The CLR uses the metadata to perform runtime checks on the code it is executing with regards to type safety and security. Checks are made to verify:

- That an application only attempts to access data that belongs to it
- That every method is called with the correct number of parameters
- That every method's return value is used properly
- That every method has a return statement
- That code is executed based on the code access policy associated with the location from which it was executed. For example, code downloaded from the internet executes with a different set of permissions than code installed natively.

This runtime checking results in making software that is safer and more robust.

In addition to the runtime checking performed by the CLR, several other features help make the .NET Framework a more robust development environment. These are:

- Memory management, via a garbage collection system
- Unified error handling
- Support for inter-operating with legacy components
- Reflection

1) Garbage Collection

Garbage collection is the process of automatically freeing up memory when the object for which it was allocated is no longer being used. When you create an object, memory is allocated and assigned to that object. When the system detects that the object is no longer being used the memory allocated for that object is freed. While you can control when memory for an object is allocated, when it is created, you do not have control over when the memory allocated for that object is freed. The life of an object is not tied to the method that creates it. An object can exist beyond the single method call that created it. The system detects when an object is no longer being used and frees it. A garbage collection prevents the following errors:

- Forgetting to free objects
- Attempting to free the same object more than once
- Freeing an active object

2) Unified Error Handling

Another feature that stands out in .NET is unified error handling. All error reporting is done via exceptions. Contrast this to COM and Win32 that provided a variety of error handling mechanisms:

- HRESULT
- Structured exception handling
- C++ Exceptions
- Return Codes
- GetLastError / SetLastError

.NET exceptions cannot be ignored, and can be thrown and caught across languages.

3) Reflection

Reflection allows code to discover information about other managed code. This is done by reading the metadata emitted by the compiler. This allows for runtime discoverability of data types and for a whole new class of debugging tools.

B. Framework Class Library

The Framework Class Library (FCL) can be thought of as an object-oriented version of the Win32 API. The FCL provides infrastructure services such as file I/O for all .NET components. The Data and XML Classes provide functionality for data access. ASP.NET provides the core Web infrastructure such as Web Forms for web-based UI development and Web Services for programmatic interface development. Windows Forms provides the core functionality for developing Windows- based user interfaces.

C. Common Language Specification

The Common Language Specification (CLS) provides a set of rules that apply to "externally visible" items in the programming languages supported by the .NET Framework. As a result, multiple components created using different .NET languages can interact seamlessly. Several programming languages are supported by the .NET Framework. Any language that is implemented to the CLS is automatically supported by the CLR. Figure 1. Microsoft .NET Framework Architecture and the CLR shows the languages developed by Microsoft; there are others developed by other companies.

.NET achieves much of its language independence by defining a unified type system. In addition to supporting all the standard data types (integers, doubles, Booleans, etc.), the CLR type system adds native support for a string data type. This alone makes a move to the .NET Framework worth it. No more BSTRs; no more forgetting to allocate an extra byte for the null-termination; no more buffer overruns. Because the string data type is defined by the CLR's unified type system, it means that strings can be used interchangeably across all languages. The type system is a function of the runtime, not of the compiler.

III. STANDARDIZATION

Microsoft showed incredible openness during the design and development of the .NET Framework and the C# language, making specifications that the details of the inner workings of the runtime available to the development community.

This openness was taken to a new level in November 2000 when Microsoft, along with co-sponsors Intel and HP, officially submitted the specifications for the C# language, a subset of the FCL, and the runtime environment to ECMA for standardization. [2]

The ECMA standards cover three areas:

- C# Language [3]
- Common Language Infrastructure (CLI) [4]
- C++/CLI Language [5]

IV. Legacy Technology Support

The dream of all developers is to work solely on a completely new code base unencumbered by the constraints of legacy code. Sadly this dream can never become a reality and thus for any new software framework to be acceptable it needs to support existing programming technologies. Expected levels of support would be in both directions where the .NET code has the ability to call legacy libraries as well as the reverse where legacy applications can call into .NET libraries.

.NET provides three different approaches for interoperability: COM, Platform Invoke, and C++/CLI. Each approach has certain benefits as well as drawbacks.

A. COM Interoperability

Because of the object orientation of COM, providing a connection between .NET and COM is rather straightforward and mechanized. Because the conversion process from one to the other is mechanical, the engineers at Microsoft developed tools that provide the necessary conversion code without significant effort on the part of the user.

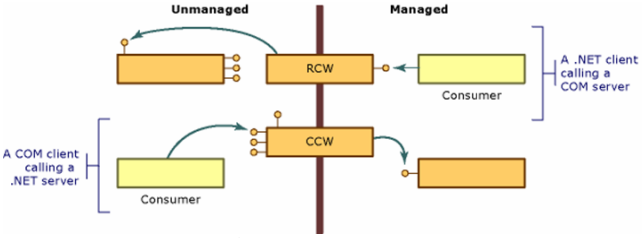

Figure 3. COM Interop

These tools basically generate two kinds of wrappers. Runtime Callable Wrappers (RCW) provide the necessary linkage to call a COM object from .NET. The tool reads the COM type library and generates all the necessary code required to create and manage the object as well as code required to marshal the arguments passed to the objects methods. This conversion tool is available either as a part of the .NET IDE, as a batch tool, or as a runtime callable converter class.

The other wrapper that is generated is a COM Callable Wrapper (CCW). This wrapper is automatically available, however to make a .NET class useful through COM; some minor decoration needs to be added to the classes methods. In COM adding new functionality was always difficult and required a significant amount of additional work; however, in .NET the tool takes care of this drudgery.

B. Platform Invoke (P/Invoke)

In much the same way as traditional C++ can directly call a C function; P/Invoke allows .NET code to call into a function within an unmanaged library. The user need only provide the function's declaration so that the necessary marshalling and linkage can take place. While most of the time the automatic parameter marshalling is satisfactory, there is also the ability to provide all the necessary customization both for data conversion as well as performance. P/Invoke also allows the bi-direction marshalling of an entire managed object across the managed/unmanaged boundary. Callbacks are also possible from both directions.

C. C++/CLI

C++ in the .NET environment consists of the ANSI definition of the language that programmers know and love along with necessary extensions to allow it work fully within the CLR and handle managed objects, known as "C++/CLI".

Since C++ has the ability to read in a header file that fully defines an object and its methods, the P/Invoke available in C# and Visual Basic is greatly simplified, to where the developer need not do anything except make the desired call. Because of this simplicity, it is referred to as "It Just Works" or IJW. That is, code written using C++/CLI can call both managed and unmanaged code without additional runtime generated wrappers or user defined P/Invoke declarations.

V. Common Problems with Non-Managed Languages

Non-managed languages are usually **weakly-typed**, which means that type checking happens at run-time. This means that programming errors due to mismatching types are caught at run-time, which is too late in the development process

.NET languages are strongly typed, this allows tools to report errors as code is entered. The type information is available at runtime making easier to build tools.

Non-managed languages leave the responsibility of **memory management** to the programmer. Programmers have to track memory allocation and de-allocation throughout their program. While the ability of programming with pointers provides flexibility and sometimes increased performance, it can also lead to common programming errors like memory overruns, which are difficult to find and fix.

The .NET Framework provides garbage collected managed memory system. This means that user no longer needs to track memory allocations. The system tracks the usage and when all the references to allocated memory go away the memory is automatically reclaimed. Also, because the system tracks memory, it knows the size of the buffers and can report accesses outside the allocated range right when they happen.

In non-managed languages, like C and C++, the use of functions and variables should come after the declarations of these functions and variables. Header '.h' files need to be used to refer to declarations in other source files. Classes and functions should be defined in a careful order to avoid situations where classes are circularly-dependent on each other.

Unlike non-managed languages, all variables and member fields are initialized to type-specific default values when they are declared in managed languages. In non-managed languages it is programmer's responsibility to initialize variables at declaration time. However, the .NET language compilers do recognize when a variable for member field is being used without being explicitly set and generate compiler errors.

Strings are first class data types in the .NET languages. No more buffer overruns because an extra byte wasn't allocated for the null-terminator.

In non-managed languages, the standard libraries for file I/O, string manipulation, network support, XML file manipulation and advanced data structures like collections, hash tables, stacks, and queues are limited at best.

VI. TOOLS

An important benefit of using .NET Framework and languages is the availability of development and debugging tools as well as a wide network of users and programmers sharing code examples and building class libraries. Microsoft VisualStudio is the front runner of commercially available development tools. MONO [6] and SharpDevelop [7] are examples of development environments, which are available as open source.

A. Microsoft VisualStudio

Starting with Version 2002, Microsoft extended its existing Application Development Environment product VisualStudio to support .NET Framework and languages. In later versions of VisualStudio, Microsoft added more development and debugging tools which greatly increased the productivity of a developer and reduced the average time in debugging problems. The following is a list that highlights a few development and debugging features available in VisualStudio.

1) Development Features

The latest VisualStudio **Text Editors** provide a number of productivity features;

1. Intellisense for member information, parameter information, word completion, and automatic brace matching
2. Auto-indentation
3. Background compilation which allows for underlining of syntax errors in the editor as you type
4. Auto-generation of stubs for new functions
5. Live display of semantic errors
6. Automatic replacement of renamed functions and variables
7. Change tracking

Navigation to the definition and to all references of a function or variable **Intellisense** provides a convenient method to access information about classes, functions and parameters. It can speed up development by reducing the amount of information the programmer needs to remember and by reducing the amount of keyboard input. It is more streamlined alternative to Function Panels in LabWindows/CVI. With intellisense, the user types the first few characters of the function or the variable name and hits ALT-TAB to list all possible completion choices available within the current programming scope. After selecting the desired one from the list, she hits enter to complete the name. This feature not only improves productivity by minimizing the keyboard input required, but also eliminates the need for memorization of function and variable names.

2) Debugging Features

The latest VisualStudio includes the following advanced debugging features in addition to the usual features available in other debugging tools;

1. A Threads window that tracks all threads (for multi-threaded applications) and provides a quick way to review the call stacks associated with each thread
2. A Modules window that monitors all the DLLs loaded during the execution of the debugged application
3. Customizable Watch windows with the ability to associate type-specific visualizers for viewing the contents of variables
4. Ability to change the value of a variable at a breakpoint
5. Ability to set/change the next statement to execute
6. Ability to attach the debugger to a process that has already started
7. An **Immediate window** that is used to debug and evaluate expressions, execute statements, and print variable values.

A **Visualizer** is a debugging component of VisualStudio. A visualizer creates a user interface to display a variable in a manner appropriate to the variable's data type. VisualStudio includes five visualizers; text, HTML, XML, WPF, and dataset visualizers. VisualStudio also provides an interface to write your own type specific visualizers that integrate with VisualStudio.

B. Open Source Tools

Mono [6] is an open source project to create an ECMA standard compliant, platform independent .NET framework and a set of tools to support it. A variety of operating systems is supported, althought the primary users of Mono are Linux developers.

SharpDevelop [7] is an open source integrated development environment for .NET languages.

C. Code Analysis Tools

Free tools like FxCop and StyleCop are used to enforce style and programming guidelines. FxCop provides a default set of rules based on Microsoft's Design Guidelines for Class Developers [8]. Additional rules can be added using FxCop's SDK. Some versions of VisualStudio has FxCop-like code analysis tools built-in. These tools can be run as part of the buid process to enforce style and programming guidelines.

Commercial tools are also available, including: CodeIt.Right, CodeRush, Parasoft dotTest.

Other tools using the reflection technology of .NET Framework have the ability to perform static analysis without source code.

VII. TEST DEVELOPMENT IN .NET

.NET languages and development tools provide a richer and safer environment for TPS development. The VisualStudio application development environment is the premier development tool on the Windows platform. Other development tools try to emulate the capabilities of VisualStudio but often fall short.

A. *Test Development Infrastructure*

Using VisualStudio and the .NET Framework for TPS development requires the same infrastructure definition that would be required for any development environment:

- Select a Test Executive
- Define the structure of the TPS
- Define and implement instrument interfaces

1) *Select a Test Executive*

Most commercial test executives already provide support for the .NET Framework. For test executives that do not support the .NET Framework, an interoperability layer is needed.

2) *Define the Structure of TPS*

This step is required regardless of the development environment and technology used. Decisions need to be made about packing:

- Is TPS code packaged as a DLL or an executable?
- How many DLLs or executables?
- How are pre-conditions and diagnostics handled?

3) *Define and Implement Instrument Interfaces*

Finally, a decision needs to be made about how instruments and other resources are accessed. Are instrument drivers called directly or are wrappers needed? This is often the most time consuming and crucial part of the infrastructure. This paper will explore three approaches and show how these approaches can all be used together so that the best interface is used for the task at hand.

- IVI.NET
- An ATLAS-like Wrapper
- An instrument-specific Wrapper

B. *Examples*

1) *IVI.NET*

The IVI Foundation has updated all of the IVI architecture and instrument class specifications to add support for native .NET drivers.

What follows is an example which shows how to make a simple DMM DC Voltage measurement using IVI.NET.

```
IIviDmm dmm = null;
try
{
   // Create an instance of the DMM using
   // the IVI Session Factory
   dmm = factory.CreateDriver("MyDmm");

   // Configure the DMM to make a DC voltage
   // measurement using auto-ranging
   dmm.Configure(MeasurementFunction.DCVolts,
       Auto.On, 0);

   // Make the measurement
   double voltageMeasurement =
      dmm.Measurement.Read(PrecisionTimeSpan.MaxValue);

   // Test value and determine pass/fail
}
catch (Exception exception)
{
   // Report the error
}
finally
{
   // Clean up if we successfully initialized
   if (dmm != null)
   {
       dmm.Utility.Reset();
       dmm.Dispose();
   }
}
```

Figure 4. IVI.NET Example

2) *ATLAS-like Wrapper*

The following is an ATLAS-like Wrapper example, where ATLAS instructions, i.e. verbs and nouns, are implemented as classes and interfaces. Member methods of these classes support variable argument lists, where arguments are keyword/value pairs. These similarities to ATLAS provide an easier transition to .NET programming for the experienced ATLAS programmer. A common class called Atlas provides central resource management functionality and manages globals, logging, error handling, and reporting test results.

This example, and the one that follows, uses C# named arguments. *Named arguments enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than the parameter's position in the parameter list.* [10] In addition to providing flexibility for the calling in terms of the order in which the arguments are specified, named arguments also enhance the readability of the code.

```
Atlas.Initialize();
Impedance impedance = null;
double measurement = 0.0;

try
{
   // Make a resistor measurement.
   impedance = Atlas.CreateImpedance("DMM-RES");

   impedance.Res.Measure(ref measurement,
                     testID: "100000",
                     cnxHi: "J7-8", cnxLo: "GND");

   // Compare the measurement value against limits.
   // Terminate the program upon a test failure.

   Atlas.TerminateOnFailure = true;
   Atlas.Compare(measurement,
                     "100000",
                     ul: 263, ll: 237);
}
catch (AtlasException aEx)
{
   aEx.Report();
}
finally
{
   Atlas.Close();
}
```

Figure 5 ATLAS-like Wrapper Example

3) *Instrument Specific Wrapper*

Occasionally you run into an instrument whose programming interface does not map to an IVI defined class or to a construct defined in ATLAS or some other standard, in this case an instrument specific wrapper is needed.

The following example shows a wrapper that mimic's Teradyne L-Series Programming Language for controlling a digital instrument.

```
CShell.Initialize();
try
{
   // Create two pins
   Pin p1 = CShell.CreatePin("P1", 6, 1);
   Pin p2 = CShell.CreatePin("P2", 6, 2);

   // Configure the pins and specify the
   // voltage levels
   CShell.SetDigital(new PinList(p1, p2),
       mode: ChannelMode.Static,
       load: Load.Off,
       level: LevelSet.A);
   CShell.SetLevels(level: LevelSet.A,
       vih: 5.0, vil: 0.0,
       voh: 2.5, vol: 2.5);

   // Execute 3 static patterns that drive
   // the two pins low, high, low
   CShell.IL(p1, p2); CShell.EOP();
   CShell.IH(p1, p2); CShell.EOP();
   CShell.IL(p1, p2); CShell.EOP();
}
catch (CShellException exception)
{
   // Catch CShell specific exceptions
}
catch (Exception exception)
{
   // Generic catch for any .NET exception
}
finally
{
   // Cleanup
   CShell.Close();
}
```

The code is self-documenting and easy to follow.

When you look at the three examples, one thing that quickly jumps out is that regardless of the interface or wrapper we are using, the structure of the code is always the same:

- The wrapper or instrument interface is initialized
- The code executed in a try/catch/finally block so that exceptions are caught
- The finally clause is used to perform cleanup

These similarities mean that the code for initialization, error handling and cleanup could be moved into the Test Executive or other TPS infrastructure component, leaving the TPS developer to focus just on the task at hand – testing the UUT.

VIII. CONCLUSIONS

.NET languages, specifically C#, should be added to the list of general purpose programming languages used for TPS development. The combination of the languages, tools and

framework address many of the problems encountered with other languages. The examples described are meant to provide inspiration for exploring a managed TPS development environment. The next step is to build such an environment and use it to develop real world test programs.

REFERENCES

[1] ".NET Framework Conceptual Overview", Internet: http://msdn.microsoft.com/en-us/library/zw4w595w(v=VS.100).aspx

[2] "Introducing C# and the .NET Framework – ECMA Standardization", Internet: http://en.csharp-online.net/Introducing_CSharp_and_the_.NET_Framework%E2%80%94ECMA_Standardization

[3] *Standard ECMA-334: C# Language Specification 4th edition*, June 2006, Internet:http://www.ecma-international.org/publications/standards/Ecma-334.htm

[4] *Standard ECMA-335: Common Language Infrastructure (CLI) 5th edition*, December 2010, Internet:http://www.ecma-international.org/publications/standards/Ecma-335.htm

[5] *Standard ECMA-372: C++/CLI Language Specification*, December 2005, Internet:http://www.ecma-international.org/publications/standards/Ecma-372.htm

[6] *Cross Platform, Open Source .NET Development Framework*, internet: http://www.mono-project.com

[7] *The Open Source Development Environment for .NET*, internet: http://www.icsharpcode.net/OpenSource/SD/

[8] *Design Guidelines for Developing Class Libraries*, internet: http://msdn.microsoft.com/en-us/library/ms229042.aspx

[9] *IVI-3.1: Driver Architecture Specification*, IVI Foundation, Revision 3.2, May 26, 2011.

[10] "C# Programming Guide", Internet: http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx