Ada 2005 on .NET and Mobile and Embedded Devices

Dr. Martin C. Carlisle
U.S. Air Force Academy

Ada is well known for supporting good software engineering practices and for interfacing cleanly with other languages; these features have only gotten better with Ada 2005. The A# project is an open-source implementation of Ada 2005 for Microsoft's .NET Framework. Using A#, programmers can combine Ada code with reusable .NET components, including modules written in C#, as well as legacy component object model components and Win32 Dynamically Linked Libraries. This allows leveraging both the software engineering advantages of Ada and the large amount of reusable libraries written for .NET. Additionally, A# targets portable digital assistants and other mobile and embedded devices.

Similar to the Java Runtime Environment, Microsoft's .NET Framework [1, 2] is attractive to software developers as it provides a large collection of precompiled classes, including security classes that allow dynamic loading of modules. Just as the Java 2 Micro Edition (J2ME) allows Java programs to be run on embedded and mobile devices, the .NET Compact Framework enables .NET applications to be run on these devices. Unlike Java, .NET had a goal of interfacing with legacy Windows code. Therefore, Microsoft provided mechanisms in .NET for easily combining legacy component object model (COM) objects and Win32 Stdcall dynamically linked libraries (DLLs) with new .NET code.

Microsoft developed a flagship language for the .NET Framework: C#. C# is an object-oriented language with a syntax and semantics that are very similar to Java. While C# and Java resolve many of the really bad problems with C and C++ (in particular, buffer overflow vulnerabilities, **=** versus **==**, single character errors, etc.), they still fail to meet many of the almost 30 year old Department of Defense (DoD) Steelman requirements [3] for programming languages that have always been met in Ada. For example, while the latest versions of C# and Java have finally added generics, both languages still fail to provide subtypes to properly model scalar values, and proper enumeration types. C# does have an *enum* type, which at first glance appears to be a proper enumeration type, but does not provide successor or predecessor functions. Additionally, C# and Java still require arrays to be indexed with integers starting at zero (Ada allows the starting index to be specified or allows indexing an array with an enumeration type, such as colors).

Ada, on the other hand, has suffered from a lack of available components (although Ada 2005 does add a new container library). Only a few of the widely used libraries support Ada. Additionally, compiler vendors have been slow to provide compilers for new platforms such as embedded and mobile devices. As a result, despite its engineering superiority, Ada is often not the best tool to get the job done.

The A# project[1] seeks to have the best of both worlds. By providing an open-source compilation environment for Ada on the .NET Framework, A# gives software developers the opportunity to leverage the large amount of reusable .NET classes while also being able to write code in a language that strongly supports good software engineering practices.

Compiling for .NET

One of the key design goals for .NET was supporting multiple different programming languages. Microsoft provides technical support to language developers and has published a list of 27 languages that compile to .NET (not including the four distributed with Visual Studio) [4]. To make it easier for compiler developers to create compilers for the .NET Framework, Microsoft provides the .NET Common Intermediate Language (CIL). The .NET CIL can be viewed as a high-level assembly language, which directly supports object-oriented features such as inheritance, dispatching, and interfaces. Since this intermediate language is object-oriented, compiling an object-oriented language to the .NET CIL is much simpler than compiling to Intel assembly language.

The .NET CIL bears a strong resemblance to Java bytecode. Therefore, JGNAT [5] (Gnu Ada Translator [GNAT] for the Java Virtual Machine) – an open source Ada compiler that compiled from Ada to Java bytecode – was used as a starting point for the compiler. We modified JGNAT to emit CIL instead of Java bytecode. The resulting compiler is called MGNAT (GNAT for Microsoft .NET). Also, we rewrote the Ada standard library packages to call .NET routines instead of those from the Java platform. JGNAT is no longer maintained by Ada Core Technologies, so it does not contain any Ada 2005 features. To support Ada 2005, MGNAT reuses code from the latest GNAT compiler (with some modifications) [6]. GNAT is an open-source Ada 2005 compiler distributed under the Free Software Foundation General Public License. GNAT runs on many different platforms, but not .NET.

To make using the compiler easier, we have modified Ada Graphical Integrated Development Environment (AdaGIDE) [7], a freely available development environment for Ada, so that the A# compiler, MGNAT, can be selected simply by pushing the target button and then selecting the .NET radio button.

Using .NET Classes in Ada

Whenever you mix programming languages, it is necessary to have a language binding that enables communication among components written in different languages. These bindings may be either written by hand or automatically generated. Win32Ada [8] is an example of a handwritten binding to the Microsoft Win32 Application Program Interface (API). The problem with manually writing such a binding is that it is incredibly tedious and quickly becomes stale. As the Microsoft Win32 API developed, Win32Ada was not kept up to date.

A# provides the MSIL2Ada (Microsoft .NET Common Intermediate Language to Ada) tool, which automatically generates bindings. MSIL2Ada is written in a combination of Ada and C# and uses the .NET reflection classes to enumerate all of the classes, methods, and attributes of a .NET DLL, then generates corresponding Ada specifications.

Consider the following C# class:

```
namespace crosstalk {
    public class Class1:Superclass,
    MyInterface {
            public color my_color;
            private string s;
            public Class1(string x) {…}
            public void package() {…}
            public static color get_color() {…}
            private int get_value() {…}
    }
}
```

MSIL2Ada parses the compiled DLL containing this class and generates the following Ada specification (corresponding to only public attributes and methods):

```
with crosstalk.Superclass;
with crosstalk.MyInterface;
with crosstalk.color;
limited with MSSyst.String;
package crosstalk.Class1 is
    type Typ;
    type Ref is access all Typ'Class;
    type Typ is new crosstalk.Superclass.Typ
    and crosstalk.MyInterface.Typ
    with record
            my_color : crosstalk.color.Valuetype;
            pragma Import(MSIL,my_color,
            "my_color");
    end record;
    function new_Class1(This : Ref := null;
            x : access MSSyst.String.Typ)
            return Ref;
    function get_color returncrosstalk.
    color.Valuetype;
    procedure package_k(This : access Typ);
private
    pragma Convention(MSIL,Typ);
    pragma MSIL_Constructor(new_Class1);
    pragma Import(MSIL,get_color,"get_color");
    pragma Import(MSIL,package_k,
    "package");
end crosstalk.Class1;
pragma Import(MSIL,crosstalk.Class1,".ver
1:0:2161:15913","[crosstalk]crosstalk.Class1");
```

In the Ada code above, the compiler directive pragma *Import* specifies that an item is being imported from a different programming language. Several new Ada 2005 features are used in this binding, making it easier to read than similar bindings written in Ada 95. First, Ada 2005 adds interfaces, which are styled after those in Java and C#. In the definition of Class1, the *and* shows how to specify that a tagged type implements an interface. Oddly, in Ada 2005, only child classes are allowed to implement interfaces. This poses no problems in mapping the C# types as all types in C# are derived from System.Object (which does not implement any interfaces).

Second, the new *limited with* clause in combination with new anonymous access types makes it easy to map mutually dependent C# classes. A *limited with* clause is added for each dependent class (as is seen for MSSyst.String).

Reserved words in C# and Ada differ, and C# is case-sensitive while Ada is not, so the *pragma Import* is used to map C# names to Ada names. Note that since *package* is a keyword in Ada, *_k* is added to its Ada identifier. The namespace *System* from C# is renamed to *MSSyst* for similar reasons.

A# was the first Ada compiler to introduce the object.method syntax, which has now become part of the Ada 2005 standard. This means that programmers using both C# and A# can use the same object-oriented syntax for method calls in both languages. In Ada 95, a call to the *package_k* method would have appeared as:

```
crosstalk.Class1.package_k(This=>
Class1_ Ptr);
```

In A# and Ada 2005, this can now be written as:

```
Class1_Ptr.package_k;
```

This is not only shorter, but simpler, as the programmer does not need to worry about what package a class was declared in to call a method on it. This is particularly helpful for non-dispatching methods (those declared with 'Class), as they may be in packages where superclasses were declared.

On the Java Virtual Machine, there were only base types (such as integer and float) and class references. The Microsoft .NET platform allows for the creation of types that stored on the stack are passed by value (hence the name *Valuetype*) instead of by heap reference. To resolve this, we have added the reserved word *Valuetype*. The *get_color* method returns something of type *crosstalk.color.Value-type*. If a type is so named, then the compiler will generate code for it according to the .NET calling conventions for Valuetypes. Since there are no pointers for these types, a full *with* is needed for the *crosstalk.color* package instead of the limited *with* used for other dependencies.

The C# *enum* is another example of a Valuetype. Although, as previously mentioned, it differs from an Ada enumeration type. A# currently maps it to an Ada enumeration. However, since C# *enum* types do not support determining a successor or predecessor, these mapped types cannot use Ada enumeration attributes (such as 'Pos or 'Succ). Unlike Ada enumeration types, .NET enumerations can have multiple names corresponding to the same value. In these cases, a named constant is declared for each additional name. In certain cases, enum values can be combined to create values that have no name (e.g., in the *FontStyle* enumeration, *Bold* and *Italic* are listed – they can be added together to create a Bold Italic style, although this is not listed in the enumeration). When the type allows this, we provide a function (+) for performing such combinations. The C# enum differs so significantly from an Ada enumeration that it would be more accurately mapped to a named integer type (e.g. *type FontStyle is new Integer*). This more precise mapping may be accomplished in a later version.

Another key difference between C# and Ada data types is the use of strings. In C#, strings are stored in 16-bit unicode, while the Ada basic string is eight-bit International Standardization for Organization (ISO) Latin-1. Since it is expected that strings will be commonly passed between the languages, A# provides a unary (+) operator to perform the following conversion.

```
Csharp_String : MSSyst.String.Ref := +
  "hello world";
```

When calling a C# method that takes a string as a parameter, the compiler will automatically insert the conversion:

```
C1 : crosstalk.Class1.Ref :=new_Class1
  (x => "hello world");
```

Extending a .NET class in Ada

A peculiarity that is exposed by the mapping to Ada is the appearance of the *this* parameter in the constructor for Class1. All C# constructors are required, as their first act, to call a parent constructor. Generally, the no-argument constructor of the parent is used; however, this can be changed in C# by using base, as the following:

```
public Class1(string x) : base(x)
```

This indicates that parameter *x* should be passed to the superclass constructor that takes a string as a parameter. When extending a .NET class in Ada, the call to the constructor method is done explicitly in the variable declaration:

```
function New_MenuItem(This : Ref := null)
        return Ref is
    Super : MenuItem.Ref :=
        MenuItem.New_MenuItem
                (MenuItem.Ref(This));
begin
```

```
    return This;
end New_MenuItem;
```

This code seems a bit peculiar, as *Super* appears to be an unused local variable, and the function looks like it will return null; however, the compiler generates the correct code that allocates a new object and calls the parent constructor. The Ada child class also needs to be marked with the convention MSIL (Microsoft .NET CIL) and have its constructor marked as an MSIL constructor (as shown in the package *crosstalk.Class1*).

Calling A# Code From C#

Although a free graphical user interface (GUI) builder tool, Rapid [9], can be used to develop user interfaces for the .NET framework in Ada, Visual Studio [10] provides a much more extensive GUI builder. Consequently, it can be advantageous to use Visual Studio to develop the user interface and then write the rest of the code in Ada.

In this case, you create a DLL from the Ada code instead of an executable. A# comes with mgnatmake, a tool which automatically detects dependencies between Ada source files, performs the required compilations, and combines the results. By default, mgnatmake generates executables, but it can be instructed to generate DLLs instead:

```
mgnatmake msil2ada -o msil2ada_output.
  dll- z -largs /DLL
```

These arguments tell mgnatmake to combine all of the dependencies of the project MSIL2Ada into an output file named msil2ada_output.dll (-o), with no main program (-z) and to create it as a DLL instead of an executable (-largs/DLL).

In Visual Studio, you can simply add a reference to this DLL, and the Intellisense will automatically suggest the Ada methods (Visual Studio automatically creates the appropriate language binding). Because .NET does not allow a namespace and a class to have the same name, it was necessary to add *_pkg* to the end of the final Ada package name, so *P1.P2.P3.Put* would be referenced as *p1.p2.p3_pkg.put*. Also, it is necessary to call the initialization routine generated by the binder explicitly from the C# code as:

```
ada_msil2ada_output_pkg.adainit();
```

The binder output has an additional *ada_* prefix on the package name (which has been given the *_pkg* suffix as previously described).

Uses of A#, Embedded and Mobile Devices, and More Interoperability

The most widely used program implemented using A# is RAPTOR [11] (Rapid Algorithmic Prototyping Tool for Ordered Reasoning). RAPTOR is an open-source visual programming environment designed for use in an introductory computer science class. RAPTOR's visual programming model is based on flowcharting. RAPTOR is being used in an increasing number of universities across the United States and Canada with inquiries from as far away as Japan.

RAPTOR is an interesting use of the A# technology, as it incorporates C# and A# code, as well as a legacy C++ graphics library. Figure 1 shows the interrelation between the various software components in RAPTOR.

The C#, C++, and Ada code are written by hand; the interoperability DLL is generated automatically by Visual Studio when a reference to the COM object is added to the project.

A# is also being used on some defense projects, in particular to port Ada applications to embedded and mobile devices using the .NET Compact Framework. It is a trivial matter to target an embedded or mobile device using A#. Simply adding the-*compact* flag to both MSIL2Ada and MGNAT instructs these tools to generate code suitable for use with the .NET Compact Framework. Two of the platforms available with the compact framework are the Pocket PC and the Smartphone 2003.

A final, recently added piece of interoperability is the ability to interface with legacy Win32 DLLs. In C#, you would add the following code to import from a Win32 DLL:

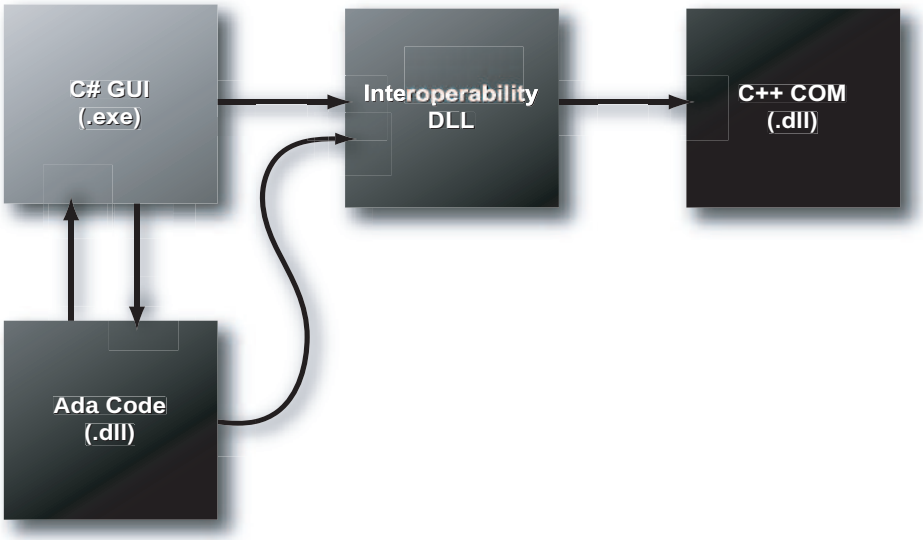
```
[DllImport("adagraph2001.dll")]
public static extern int
    CreateGraphWindow(int size);
```

In A#, you instead do the following:

```
function Open_Graph_Window(Size : in
    Integer)
    return Integer;
pragma Export(Stdcall,Open_Graph_Window,
    "[adagraph2001.dll]CreateGraph
    Window");
```

This also provides a function body (which will be ignored). While it is irregular to use a *pragma Export* to import from another file, this is done because it is necessary to generate code in addition to merely a call, and it was simpler to generate a body on a function marked for export.

Figure 1: *Interoperability Demonstrated in RAPTOR*



Conclusions and Future Work

A# has demonstrated the viability and usefulness of combining Ada with other .NET languages, as well as providing interfacing to legacy Win32 and COM libraries. This allows developers to gain the advantages of Ada's strong typing while also leveraging the vast number of libraries available with .NET and the GUI builder from Visual Studio. Furthermore, A# provides an easy mechanism for getting Ada code running on embedded and mobile devices via the .NET compact framework (e.g. Windows CE, Pocket PC, Smartphone 2003).

There are significant areas for future work. First, we are currently working to fully integrate Ada into Visual Studio 2005. Visual Studio 2005 now allows extensions to be written in .NET languages (Visual

Studio 2003 required the extension to be written using C++ COM objects), so the integration code is also being written in a combination of A# and C#.

Second, version 2 of the .NET framework adds generics. Using the generics built into the framework to implement Ada generics might reduce code size and make the genericity of Ada constructs visible to other .NET languages. This will be a non-trivial effort as the generic model in .NET is not as fully featured as that in Ada (it allows only types as generic parameters). Also, MSIL2Ada and MGNAT need to be updated to allow Ada programmers to use generic classes written in C#.

Finally, while A# has been maintained as an academic project (even though it is in use by defense contractors), it would be preferable to perform technology transfer to the private sector, which has greater resources to develop and maintain this product.◆

References

1. "Introduction to the .NET Framework." DevHood. 7 Mar. 2006. <www.devhood.com/training_modules/dist-a/Intro.NET/?module_id=1>.
2. "Technology Overview: What Is .NET?" Microsoft. 7 Mar. 2006 <http://msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx>.
3. U.S. Department of Defense. Requirements for High Order Computer Programming Languages. "STEELMAN." 1978. 29 Nov. 2005 <www.adahome.com/History/Steel man/intro.htm>.
4. "About Languages: Welcome to the .NET Language Developers Group." Gotdotnet.com. Oct. 2004 <www.gotdotnet.com/team/lang/> Microsoft. (29 Nov. 2005).
5. Comar, Cyrille, Gary Dismukes, and Franco Gasperoni. "Targeting GNAT to the Java Virtual Machine." Proc. of the Tri-Ada 97 Conference. St Louis, Mo., 9 Nov. 1997.
6. "The Libre Site for Free Software Developers." Ada Core Technologies. 29 Nov. 2005 <http://libre.adacore. com>.
7. Carlisle, Martin. "AdaGIDE Home Page." 29 Nov. 2005 <http://adagide.martincarlisle.com>.
8. Taft, S. Tucker. "Win32Ada." 22 June 1999 Averstar and Labtek (29 Nov. 2005) <http://archive.adaic.com/tools/bindings/win32ada/win32 ada.html>.
9. Carlisle, Martin. "RAPID Home Page." 29 Nov. 2005 <http://rapid.martincarlisle.com>.
10. "Microsoft Visual Studio Developer Center." Microsoft. 29 Nov. 2005 <http://msdn.microsoft.com/vstudio/>.
11. Carlisle, Martin. "RAPTOR Home Page." 1 Dec. 2005 <http://raptor.martincarlisle.com>.

Note

1. See <http://asharp.martincarlisle.com> for more information on the A# project. Developers can download A# for free.

About the Author

Martin C. Carlisle, Ph.D., is a professor of computer science at the U.S. Air Force Academy in Colorado Springs, Co. His research interests include programming languages, computer security, and computer science education. Carlisle is the primary author of several open-source software products used worldwide, including AdaGIDE, RAPTOR, and A#. He has a Bachelor of Science in mathematics and computer science from the University of Delaware and a Master of Arts and Doctorate in computer science from Princeton University.

Department of Computer Science
2354 Fairchild DR
STE 6G101
U.S. Air Force Academy
Colorado Springs, CO 80840-6234
Phone: (719) 333-3590
Fax: (719) 333-3338
E-mail: carlislem@acm.org

Letter to the Editor

Dear CrossTalk Editor,

Kevin Stamey's sponsor note, "Why Do Projects Fail?" in CrossTalk's June 2006 issue was encouraging. Software people are finally starting to realize that systems engineering is necessary to their success. What Stamey observes is mostly correct. But he does omit several items, some of which were touched on by the articles in the June issue.

He omits Configuration Management (CM). Without it you are doomed to fail. Who hasn't been burned by some cowboy coder who decided to make an *improvement* without telling anyone, let alone obtaining authorization, delaying testing and causing previously working code modules to fail unexpectedly. Even finding the latest version of a document challenges most organizations.

But CM is really a subset of communication and coordination. When I worked in acquisition, I included a glossary of every term used so there would be no mix-ups, as in Alan Jost's article. Anyone who does not define their terminology is asking for protests, screw-ups, and lawsuits. Why including a glossary isn't standard practice is a mystery. It should continue into the development work by instantiating a project glossary that goes to the level of detail of the units used in calculations.

As Capers Jones alludes to in his article, lack of adequate resources is a root cause of failure. Lack of ethics and moral courage on the part of management and engineering exacerbates the problem, as does outside influences such as political pressure and executives who want to *make the numbers* to get their bonus; congress may cancel funding if progress is not shown. With such a situation, misleading status reports are sure to result, making the situation even more critical later on.

Tim Perkins has the best high-level diagram that I have seen. I infer that it puts too much faith in CMMI-type answers, but it captures the paths to the real root causes. However, Item 150 is a constraint that must be considered in the Systems Architecture; it is not a valid cause of project failure.

Between large, complex, unprecedented systems and small routine, incremental improvements to COTS, there is a wide range of processes that should be used. Processes must be tailored to fit the situation. This requires that competent people be used. Ones who understand, not merely check off boxes on some list. They must truly understand the essence of what they are doing and not just chant the black magic incantations they were promulgated by some professor.

William Adams, PE, Ph.D.
<williamadams@ieee.org>