ROOT.NET: Using ROOT from .NET languages like C# and F#

G. Watts
Box 351560
Department of Physics, University of Washington
Seattle, WA, 98195
USA
gwatts@uw.edu

Abstract. ROOT.NET provides an interface between Microsoft’s Common Language Runtime (CLR) and .NET technology and the ubiquitous particle physics analysis tool, ROOT. ROOT.NET automatically generates a series of efficient wrappers around the ROOT API. Unlike pyROOT, these wrappers are statically typed and so are highly efficient as compared to the Python wrappers. The connection to .NET means that one gains access to the full series of languages developed for the CLR including functional languages like F# (based on OCaml). Many features that make ROOT objects work well in the .NET world are added (properties, IEnumerable interface, LINQ compatibility, etc.). Dynamic languages based on the CLR can be used as well, of course (Python, for example). Additionally it is now possible to access ROOT objects that are unknown to the translation tool. This poster will describe the techniques used to effect this translation, along with performance comparisons, and examples. All described source code is posted on the open source site CodePlex.

1. Introduction
For a programming language to be useful in High Energy Physics (HEP) it must work well with the ROOT toolkit [1]. ROOT is a de facto standard in HEP. ROOT is a kitchen-sink toolkit—it defines, reads and writes the data format used by many experiments in HEP, supports histogram generation, fitting, multivariate analysis, and statistical tools. In short, access to the ROOT toolkit is a must-have for any programming language to be used in HEP.

The .NET languages on the Microsoft platform (and Mono) are, in many ways more advanced than C++, incorporating recent research into programming languages. Functional features, like lambda functions and Language Integrated Query (LINQ), make languages like C# more expressive. Garbage collection aids with lifetime management of objects. In addition recent activity around functional languages (OCaml [2], etc., F# on the .NET platform [3]) are bringing a new way of expressing problems in very concise declarative forms rather than our usual imperative forms.

.NET (and the Java platform as well) use a runtime and allow trivial sharing of libraries between languages: making the library available for one language (C#) generally means it is available for all others (F#, .NET Python, etc.). This work need be done only once and it is available in many languages.

Initial motivation was driven by a number of small projects written by the author at the DØ experiment, located at the Tevatron accelerator at Fermilab. Initially custom code was written to wrap
a few ROOT objects. These wrappers were efficient, but tedious and limited. Passing ROOT objects to ROOT methods of other ROOT objects was painful. The result was minimal use of ROOT’s functionality due to the overhead.

The lack of speed of the pyROOT interface was also a driving force. pyROOT is a project similar to ROOT.NET: it makes the complete ROOT object structure available in Python. However, access is too slow to code a large project in Python due to its late binding nature. However, the techniques used in pyROOT could be extended to automatically write efficient static wrappers in the .NET system.

The ROOT.NET project is actually a wrapper generator. It uses ROOT’s meta-data information to enumerate all the objects found in ROOT, their methods, their arguments, global variables, enums, etc. The generator then writes wrapper C++/CLI source code and instructions to build the source into redistributable libraries. C++/CLI is the implementation of the C++ language in .NET – instead of compiling down to native code, it compiles to the .NET’s assembly language. The most important feature of C++/CLI is that it supports both the worlds of native code and .NET code in a single executable. It automatically handles the transition between the two worlds at the function level.

This paper describes the usage of ROOT.NET in the next section. Section 3. describes the implementation of the wrapper generator, including marrying the object model in the .NET world and the C++ world. Section 4. discusses dynamic access to ROOT. This is a fairly new feature which allows access to the ROOT object model w/out the full wrappers present. It is slower; however the libraries required are very small. Finally, Section 5. discusses the speed and performance of the wrappers as compared to native code and pyROOT.

2. Usage

All source code is open source. The code and documentation are all available at the CodePlex website for this project: [http://rootdotnet.codeplex.com/](http://rootdotnet.codeplex.com/). Wrappers must be built for each version of ROOT due to the way a DLL is laid out in Windows. The wrapper libraries are available using Nuget [4], the open source library/package distribution mechanism for Windows libraries. The Nuget feed is a custom one: [http://deeptalk.phys.washington.edu/rootNuGet/nuget](http://deeptalk.phys.washington.edu/rootNuGet/nuget), and must be added to the Nuget package sources (found in Visual Studio 2010 Package Manager, preferences).

Once that onetime setup is done the ROOT wrappers may be added to any .NET library or application by using the standard Nuget management dialog (right click on project References folder). Figure 1 shows an example. When choosing the packages to install make sure that the ROOT version is the same one or unpredictable behaviour may occur.

![Figure 1: The Nuget package dialog, showing a list of wrapper packages available for v5.30.01 of ROOT.](image-url)
For each version of root there are 6 different packages available, depending on which ROOT libraries are needed by the user’s project: Core, Dynamic, Graphing, PROOF, Tree, and All. Core contains the main resources for doing basic file I/O, creating histograms, threading, etc., and Graphing includes access to TPad objects and the like. The All wrapper library contains all libraries not included by the other libraries. The decision of what libraries go where was based a bit on usage statistics from the ROOT team and inter-library linkages. The Description associated with each library lists the ROOT libraries that are wrapped for easy reference. The All library will bring down about 20 MB of libraries. Core is all you need for most simple projects. Dynamic is less than 2 MB and allows you to access (inefficiently) all of ROOT.

**Figure 2:** Simple program showing how to create a ROOT object and access its methods.

Figure 2 shows a simple example. Line 8 creates a new TH1F object – all wrapped objects have an \textit{N} in front of them. The arguments are as expected, as are the method names and calls. This similarity is one of the goals of this project. There are differences, however. For example, .NET languages are garbage collected, and this can conflict with ROOT’s scoping and life-time rules. In this short program the histogram will be automatically deleted when the \textit{h} variable is garbage collected at the end of the program.

There are ways to manage the lifetime. However, this is rarely required. Figure 3 shows an example of this. Line 12 informs the ROOT.NET wrapper that it should not delete the object when it is garbage collected. This is required here because the call to \textit{SetDirectory} in line 11 causes the TFile object to take over ownership of the histogram, and it will delete the histogram when the file is closed in line 14.

**Figure 3:** The ability to manage how .NET’s garbage collector treats ROOT objects is controlled with the \textit{SetNativePointerOwner} method.

In this particular case the call to \textit{SetNativePointerOwner} is not required. This is because the ROOT.NET wrappers monitor object deletes by the ROOT system. If an object delete occurs, ROOT.NET updates the wrapper so that it won’t access a dangling pointer. Unfortunately, ROOT does not always inform the ROOT.NET wrappers this has happened, hence the need for \textit{SetNativePointerOwner} in rare circumstances.

### 3. Generating the Wrappers

Generating the wrappers is quite simple. The devil, as always, is in the details. CINT, the C++ interpreter built into ROOT, maintains complete meta-data on all objects and functions known to ROOT, including their super-classes, methods, global instance variables, and fields. The wrapper generator uses this information to produce the wrapper source code. The Visual Studio C++/CLI compiler is then invoked to compile the source files into libraries, one .NET library per ROOT library.
These libraries are wrapped into Nuget packages and uploaded to University of Washington Nuget server.

In most circumstances wrapper generation is straightforward. The translation between .NET arguments and C++ arguments must occur for all function arguments, and for the return type. Some translations are trivial – for example, int, double, float, etc. Others are more difficult – strings in particular. C++ strings are either the STL `std::string` object, or `char *`. In the .NET world, however, strings are all Unicode garbage collected objects – so there is a real cost to the translation.

Arrays cause particular problems as well, especially non-const arrays. In the .NET world one knows, implicitly, the length of any array argument: it is part of the array object’s instance. In the C++ world this is not always the case. It is possible to pass in a `float *` as an argument and from the argument type one can’t tell if that is a pointer to a single floating point number, or the start of a list of floating points whose length is known by convention (see arguments to TLorentzVector’s constructor for examples of this). The wrapper generator assumes that pointers like this are arrays – though they may be an array of length one.

Figure 4: Example generated wrapper method. See the text for a detailed explanation.

```csharp
    188     double TH1F::Chi2Test (ROOT::ITh1 * h1, ROOT::ITh1 * h2, System::String * option, array<double> res);
    197     {
    198         ::TH1 trans_h1 = (h1 == nullptr) ? 0 : (h1->CloneInstance_TH1());
    199         ROOT::ITh1::Utility::NetStringToConstCHAR trans_option(option);
    200         ROOT::ITh1::Utility::NetArrayToTranslator<double> trans_resN(res);
    201         double *trans_res = (double *) trans_resN;
    202         double f_DomResult = ((const TH1 *)h1)->Chi2Test(trans_h1, trans_option, trans_res);
    203         trans_res->updateFres(res);
    204         return f_DomResult;
    
```

Figure 4 shows an example of a generated wrapper method for the TH1F::Chi2Test method. Line 198 shows how the pointer to another TH1 object is fetched from the wrapper. Note protection against a null pointer, just in case that is an allowed as an argument. Line 199 uses a utility object to translate the Unicode string to a standard `char *` string. Some space must be allocated to do this, and it will be cleaned up when the `trans_option` variable goes out of scope. Line 200 translates an incoming array of doubles to a C++ array. Note the pointer is passed in directly in line 202: if the user called with the incorrect number of array elements ROOT will crash or behave unexpectedly, just as if this was done in a normal C++ program. Line 203 then scans the C++ array for changes and copies them back into the .NET world. Thus, if the Chi2 test wrote into that pre-allocated array, the changes will be reflected back to the .NET world as one would expect.

There are a number of differences in the programming model of C++ and .NET. Global variables can exist in the root namespace in C++. This is not possible in .NET, however. So ROOT makes a guess as to where they should be placed – for example access to the `gROOT` variable is found via `NTROOT.gROOT`. Macros that are sometimes used to make access to ROOT global variables are not detected, and, thus, not translated (for example `gDebug`).

The inheritance model in .NET languages is different than in C++. Method hiding works differently in C++ than in .NET. Once a method is visible in an object, it is not possible to hide it – unlike C++ where a subclass can hide a super-classes method, and make it inaccessible for calling, even! This is accommodated by throwing an exception when the user attempts to access one of these illegal methods. Issues like these can’t be simply ignored because they will generate invalid C++/CLI code if not watched for.

Worse is the fact that .NET does not allow multiple inheritance. Though ROOT doesn’t make heavy use of this feature, it does enough that some major design work was done around it (think of TAttLine and TH1). .NET (along with other languages that support only single inheritance) use the concept of interfaces to get around this problem. An interface is just a specification of methods that will be implemented by the object. An object can support an unlimited number of interfaces. A consequence of this is that only interfaces can be passed in as arguments to methods, as can be seen in line 196 of Figure 4. The current version of the wrappers never has an object inheriting from a super-class. The addition of this would likely dramatically reduce the size of the wrappers.
The code that deals with the inheritance model and public/private methods is the most complex in the wrapper generator. But it isn’t always sufficient. For example, there are times that CINT lies – like the arguments from TTree::Process. These are cases where the CINT signature does not match the C++ signature and as a result the generated code is incorrect and will not compile. For this very small number of cases there is a mechanism to hardwire a different behavior in a fairly modular and general way.

Only objects that derive from TObject are translated. This is because the base wrapper object needs some way to do proper pointer referencing. Further, objects in nested namespaces, or defined in other classes are not currently supported. There is no .NET reason for either of these restrictions.

As noted in Section 2., the lifetime rules for objects in .NET are different than C++. Each wrapper maintains a pointer to a ROOT object and it knows if it owns the object. If it does own the object, then it will delete the object when the wrapper is garbage collected. Ownership rules in ROOT are a complex. There is a central clearing house for all wrapped objects. Anytime ROOT returns an object pointer as a method result this clearing house is queried. If the object has never been seen before, then the wrappers assume the object’s method is managed by ROOT and the wrapper does not own the object. If the object is created using the new operator, then the wrapper owns the object. The clearing house has the extra benefit of returning a pre-existing wrapper if the object pointer returned is already known. The wrappers also hook into ROOT’s notification system and should be informed if ROOT deletes a C++ object (this is not, as noted previously, 100% fool-proof). A wrapper that has a deleted object will set its instance pointer to zero – and will generate a null exception the next time the user attempts to use the wrapper.

Finally, there are some things that just can’t be translated. If a C++ global instance (static) variable is accessed only in inline functions the resulting libraries can’t be properly linked and the build will fail. Further, a wrapper library is generated for each ROOT library, and if there is a circular reference between libraries then the linkage rules for .NET will be broken. In all of these cases a configuration file provides a way to tell the wrapper generator to ignore the methods. There is also a build server which quickly builds and reports errors that can be quickly fixed by the ROOT team.

.NET provides a number of convenience programming features. Indeed, these are some of the reasons that code is more concise in C# and other languages than it might be in C++. The wrapper generators have some rudimentary support for these features built in.

.NET properties are like class instance variables, but allow for one to have code behind their access. For example, GetMaximum and SetMaximum in TH1F are converted to a Maximum property (the GetMaximum and SetMaximum methods remain available). Property assessors are created only for GetXXX methods with zero arguments and SetXXX methods with a single argument.

Any object that derives from TCollection has IEnumerable implemented. IEnumerable is the .NET equivalent of a C++ iterator. This allows simple iteration over the list’s contents using the foreach construct in C# or even LINQ. Figure 5 shows an example of both the properties and the IEnumerable extensions.

```
static void Main(string[] args)
{
    var h = new ROOTNET.NTH1F("histo", "Sample Histogram", 20, 0.0, 20.0);
    h.Fill(5);
    Console.WriteLine("Max value of histogram is \{0\}", h.Maximum);
    h.Maximum = 2.0;
    Console.WriteLine("Max value of histogram is \{0\}", h.Maximum);

    foreach (var v in ROOTNET.NTOCLASS.getROOT().ListOfClass().Cast<ROOTNET.Interface.NTCLASS>())
    {
        Console.WriteLine("Class name \{0\}", v.Name);
    }
}
```

Figure 5: .NET features like properties and IEnumerable are implemented by the wrapper generator. The NTH1F::Maximum property is translated into calls to the TH1F::GetMaximum and TH1F::SetMaximum methods. The iteration happens over the list of classes using the automatic IEnumerable extension in Line 15. Note the use of the LINQ Cast operator to convert each object from an NTOClass into a NTClass object.
4. Dynamic access to ROOT objects and methods

In C# version 4.0, late-binding dynamic access to object methods, properties, and iterators was added via the dynamic keyword. This allows for late-binding: the compiler assumes the method call or property access is correctly formed and allows the runtime to determine exactly how to make the access or method call. This is exactly the technique used by the pyROOT library. The ROOT.NET implementation of this feature was, in many ways, patterned after the pyROOT library’s source code.

Though strongly typed wrappers are fast and provided for almost everything in ROOT, there are circumstances where they prove to be convenient. First, they require only a minimal library – less than 2 MB. Second, if the object is a custom object that doesn’t have the wrappers – a user object, for example – then access will not be possible without this feature. The size of the library could be shrunk further in the future.

Figure 6: Dynamic creation of a TH1F object, and then filling. The ROOT object there can be called with an arbitrary object name to create it, with the arguments after that being passed to the constructor.

```csharp
static void Main(string[] args)
{
    var h = ((dynamic)ROOTNET.Utility.ROOTCreator.ROOT).TH1F("histo", "Sample Histogram", 20, 0.0, 20.0);
    h.Fill(10);
    h.Print();
}
```

Figure 6 shows a simple example. Line 12 uses the ROOT dynamic object to allow one to create any ROOT object that derives from TObject (the line can be made more compact with namespace declarations). Line 13 and 14 are actually dynamic calls – at compile time the compiler has no idea what type h is, as opposed to previous examples.

Microsoft’s DLR (Dynamic Language Runtime) is at the heart of the ROOT.NET dynamic implementation. When a dynamic call site is encountered on a dynamic object, a call will be made to the object to allow it to determine how to actually make the method call or property access. The ROOT.NET dynamic infrastructure uses CINT to determine which method to call, and then codes up a CINT driven call to the method, and then invokes CINT to do the call. Hence the call goes from the .NET world, to the C++/CLI world, then to the CINT world, and then all the way back again. This is not efficient.

The DLR is capable of many optimizations. The current implementation of the dynamic interface used by ROOT.NET is as inefficient as it can get. For example, many object allocations are made on every dynamic call—even the same one repeatedly. On each call CINT is queried for methods that match the incoming parameters. Then the incoming parameters are correctly type-cast into a form CINT can use, and finally CINT is used to dispatch the call. Future releases will more closely integrate ROOT with the DLR and will remove many of these steps for the same call-site. However, it can never go as fast as the static wrappers: all calls are forced to go through CINT—and that isn’t fast. However, performance similar to pyROOT should be possible eventually.

The convenience features – like properties or iterators, are not implemented in the dynamic infrastructure. There is no technical reason for this, and they are on the future upgrade list. Finally, the dynamic interface is very new and as such hasn’t had nearly the same amount of testing that the fully typed static interfaces have. It is likely there are other bugs and performance issues lurking.

5. Performance

Each time ROOT.NET implements a method call, it must move from the managed CLR world into the native C++ world. All data that moves between the worlds must be translated. Some translations are cheap (int, double, etc.) and some are expensive (strings, arrays), and some are in between.

A very simple performance test was done filling a histogram repeatedly with the same value. After running the test several times to make sure the disk cache was warm, a measurement was taken to see how many times TH1F::Fill could be called in a 5 second interval. All the test code is available in source control at the main ROOT.NET CodePlex repository.
Figure 7: Performance numbers for a simple loop that fills a histogram. The y axis is iterations in 5 seconds, and a larger result is better. Dynamic is slow – 110K iterations in the 5 seconds, and so it isn’t visible on this scale.

Figure 7 shows the results of this simple test. The Dynamic column represents a test program that fills the histogram using the dynamic call infrastructure. The actual rate is 110K iterations in 5 seconds, too small to see on that scale. Python is a small Python script which does the same thing, C# Wrapper represents the full static calling, and C++ direct is a compiled C++ code. All tests are run on the same computer (a first generation Intel Core i7 running Windows 7). This call sequence shows off the wrappers at their best: translating the double in the argument of TH1F::Fill from the .NET world to the native world is, basically, a non-operation. In this case the static wrappers perform at about 2/3’s the rate of a native program.

A second test was performed where the arguments to the method required some real translation. The bin labels in a TAxis were set using TAxis::SetBinLabel. Figure 8 shows the results. Again, the dynamic implementation is almost invisible at this scale: 125K iterations in 5 seconds. The C# wrappers aren’t much better than Python when compared to C++. This is because setting a string name in C++ is just changing a pointer. In Python and in .NET the string object must be converted to a C-style string. This involves allocation and de-allocation—which is expensive and the largest part of the conversion. The conversion in Python requires less work than in .NET because all .NET strings are in Unicode.

Figure 8: Performance results of calling TAxis::SetBinLabel repeatedly. The y axis is iterations in 5 seconds, and a larger result is better. The dynamic test was 125K iterations per second, which is barely visible on this scale.

Other operations—like passing a ROOT object as an argument to a ROOT method — fall somewhere in between these two examples. In the particular case of a ROOT object pointer, the performance looks very similar to the speed filling of the TH1F as no conversion is required.
6. Conclusions
The ROOT.NET wrappers project has evolved in the 5 years it has been around steadily gaining features and capabilities. Most recently gained are the dynamic calling and convenience features that allow for better integration with .NET. The immediate future work will be improving the dynamic performance and adding property and iterator features. There is also work underway to reduce the size of the libraries – mostly by making use of object inheritance. Some optimization work is also desirable. Work is also on-going to move all the Nuget packages into the central Nuget repository rather than the small custom one currently being used. There have been some efforts to explore efficient access to TTree data using the iterator formalism; however for performance reasons alternate methods, like the LINQtoROOT project have been perused instead.

As a result of this work .NET languages, like C# and F#, have almost full access to the ROOT toolkit. The wrappers have been used in many small projects by the author – even in GUI based Windows applications, and there has been some interest by outside people (which motivated the Nuget packaging).

References
[1] ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.
[2] The OCaml functional programming language. Documentation and downloads can be found at http://caml.inria.fr/.
[3] D. Syme, A. Granicz, A. Cisternino, Expert F# 2.0, apress, 2010, ISBN 978-1-4302-2431-0. Further documentation and downloads can be found at http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.
[4] NUGET is an open source package management tool. Libraries uploaded to repositories can be easily installed by users. http://nuget.org
[5] Watts, G. May 2012, Using Functional Languages and Declarative Programming to Analyze Large Datasets: LINQtoROOT, J./Phys: Conference Series (CHEP2012), At Press.