ROOT.NET: Using ROOT from .NET languages

C# or F#

http://rootdotnet.codeplex.com

#384

Gordon Watts

University of Washington

Paper

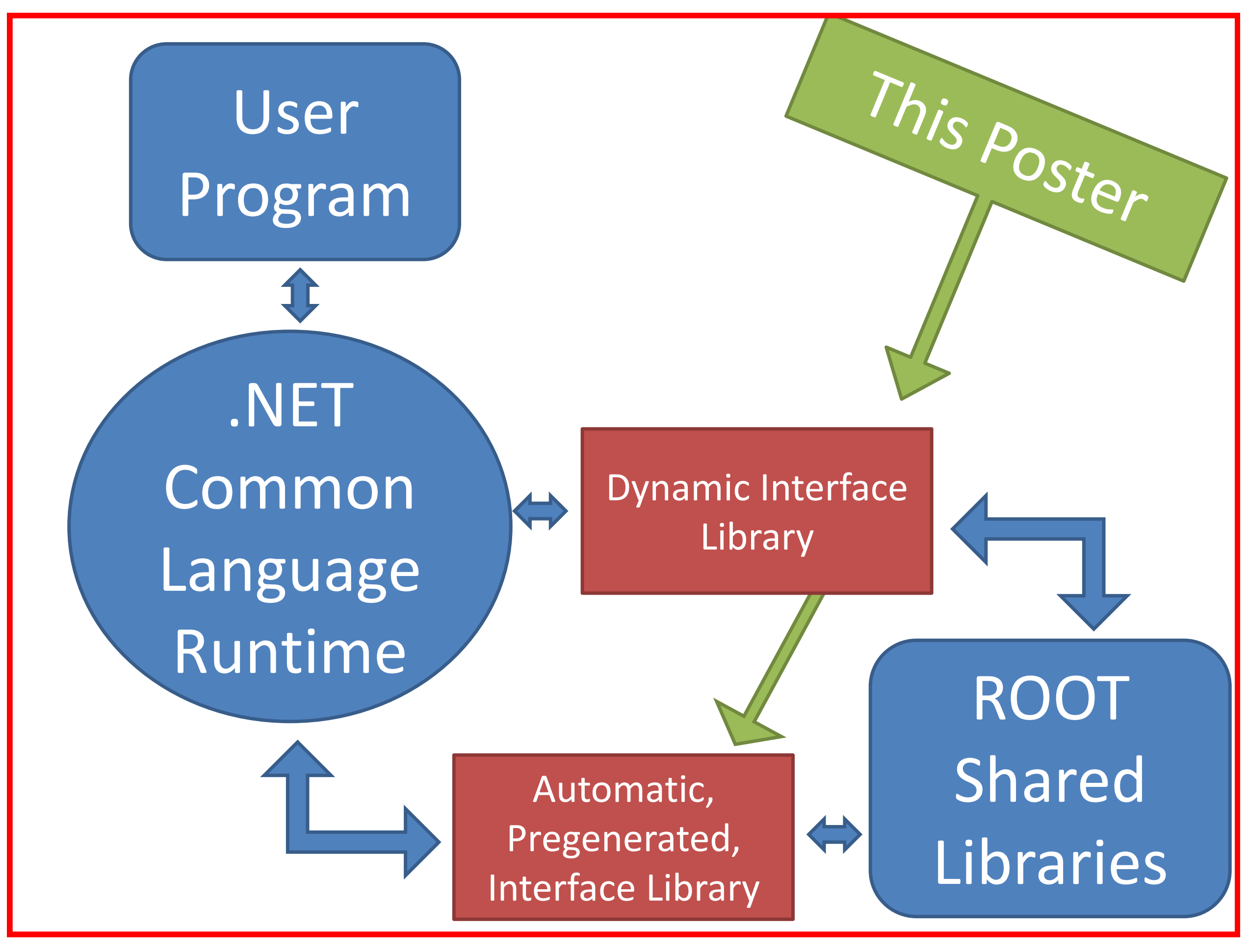
Website

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

Why Marry the ROOT and .NET Worlds?

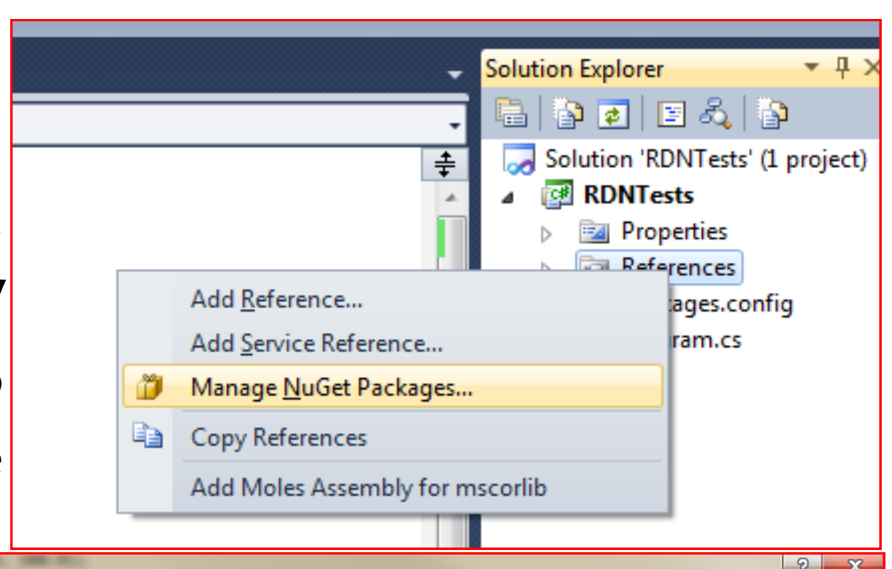
- I am much more productive in languages like python and C# than I am in C++. Many of the functional features (lambda's, Language Integrated Query (LINQ), etc.) make these languages much more concise in expression problems. In addition, the recent activity around functional languages (OCaml, etc., F# on the .NET platform) 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#, etc.). .NET was chosen over the JVM because of the features most interesting to me (lamdbas, LINQ) were not available in other tools.
- ROOT is the de facto data storage and histogramming, and fitting, etc., package for the field of HEP. This project is the infrastructure for an attempt to marry these two worlds.
- Initial motivation was driven by a number of small projects at the DØ experiment, located at the Tevatron accelerator at Fermilab. Custom wrappers that I wrote were efficient, but tedious. Passing ROOT objects to ROOT methods of other ROOT objects was difficult. I found myself making minimal use of ROOT's functionality to avoid the extra work. I also had a large number of python files used in my analysis at DØ—but they were slow.
- The pyROOT and Ruby ROOT were further inspiration and showed that this could be done. And further provided me with names of people to contact with questions and for help!


User Program
This Poster
.NET Common Language Runtime
Dynamic Interface Library
ROOT Shared Libraries
Automatic, Pregenerated, Interface Library

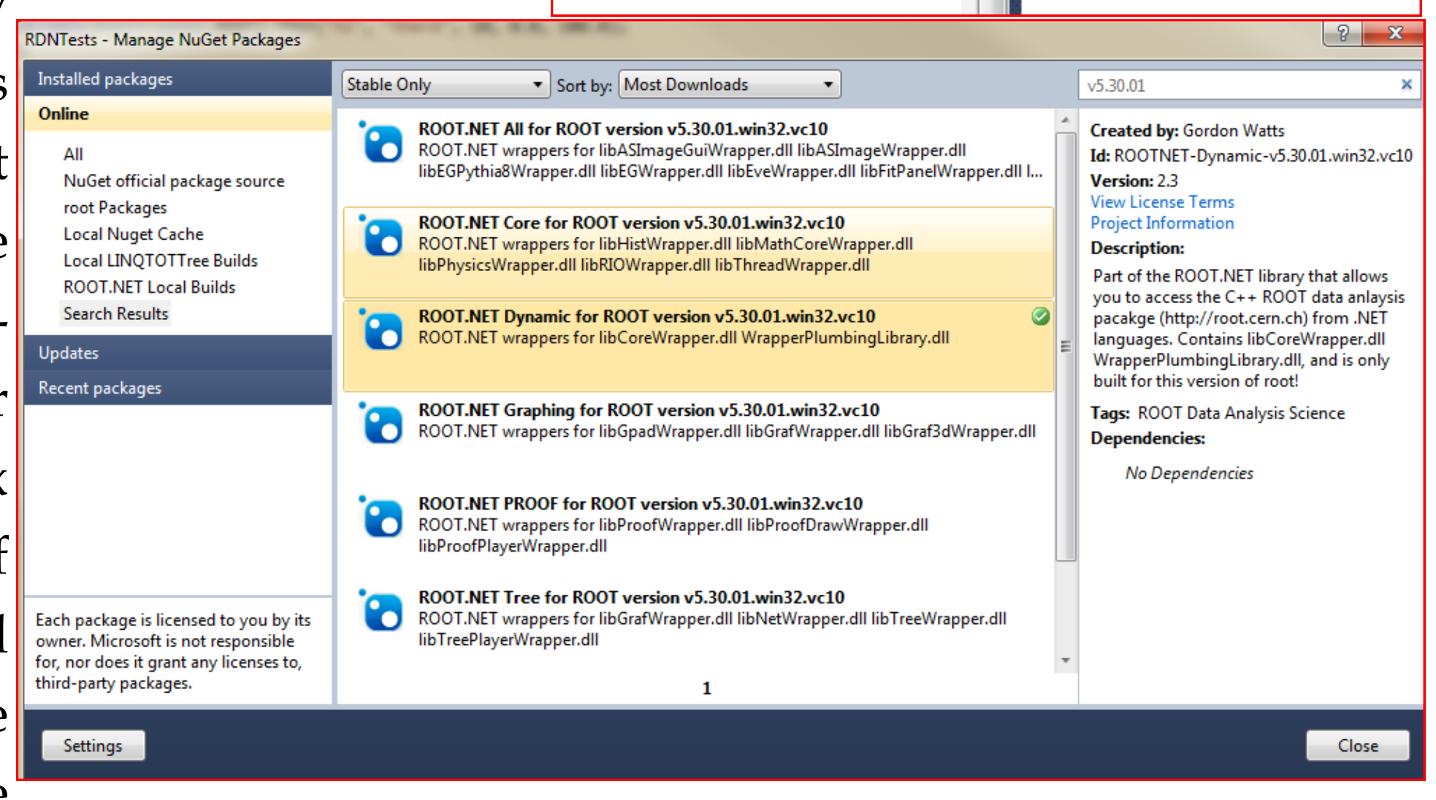
Many thanks to Axel Naumann (CERN/ROOT), Wim Lavrijsen (LBL/pyROOT), and Bertrand Bellenot (CERN/ROOT) for advice, discussion, and suggestions!

Usage

Onetime setup: in Visual Studio 2010, under options, *Package Manager*, and *Package Sources* add **http://deeptalk.phys.washington.edu/rootNuGet/nuget** to the list. That done, in any .NET solution, start the NuGet package manager by right clicking on References (as seen at right). From the list of packages presented pick the package that you are most interested in using (see lower right). Note in the search box I've entered the version of ROOT installed on my local machine. This is a must! The linkage does not translate from one version of ROOT to the next. For each version there are a set of packages that contain various libraries. Note in the description is a list of all the libraries that are wrapped in each NuGet package.



Solution 'RDNTests' (1 project)
RDNTests
Properties
References
packages.config
Program.cs



RDNTests - Manage NuGet Packages

ROOT.NET All for ROOT version v5.30.01.win32.vc10
ROOT.NET wrappers for libASImageGuiWrapper.dll libASImageWrapper.dll libEGPythia8Wrapper.dll libEGWrapper.dll libEveWrapper.dll libFitPanelWrapper.dll l...

ROOT.NET Core for ROOT version v5.30.01.win32.vc10
ROOT.NET wrappers for libHistWrapper.dll libMathCoreWrapper.dll libPhysicsWrapper.dll libRIOWrapper.dll libThreadWrapper.dll

ROOT.NET Dynamic for ROOT version v5.30.01.win32.vc10
ROOT.NET wrappers for libCoreWrapper.dll WrapperPlumbingLibrary.dll

ROOT.NET Graphing for ROOT version v5.30.01.win32.vc10
ROOT.NET wrappers for libGpadWrapper.dll libGrafWrapper.dll libGraf3dWrapper.dll

ROOT.NET PROOF for ROOT version v5.30.01.win32.vc10
ROOT.NET wrappers for libProofWrapper.dll libProofDrawWrapper.dll libProofPlayerWrapper.dll

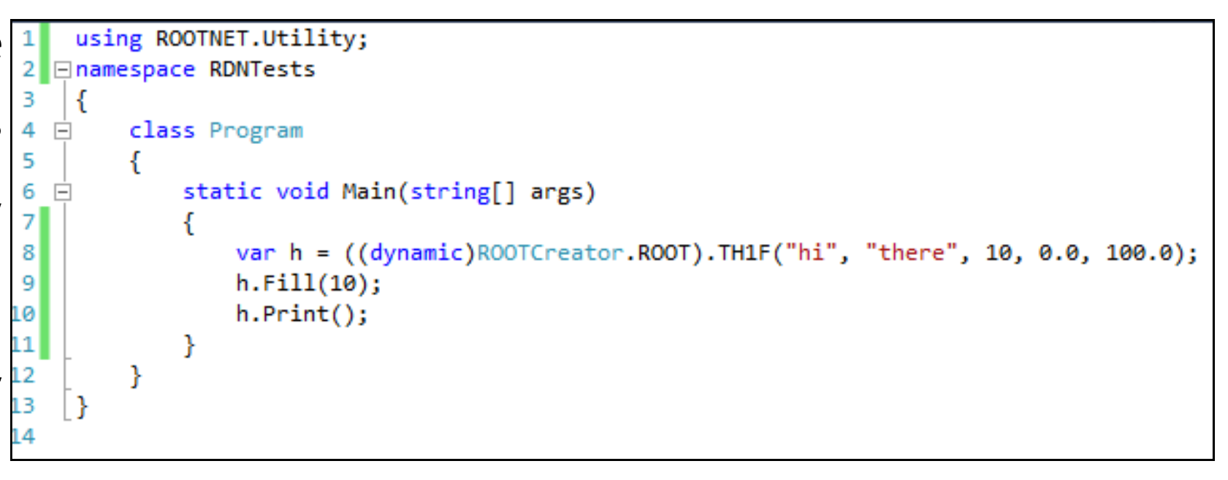
ROOT.NET Tree for ROOT version v5.30.01.win32.vc10
ROOT.NET wrappers for libGrafWrapper.dll libNetWrapper.dll libTreeWrapper.dll libTreePlayerWrapper.dll

1

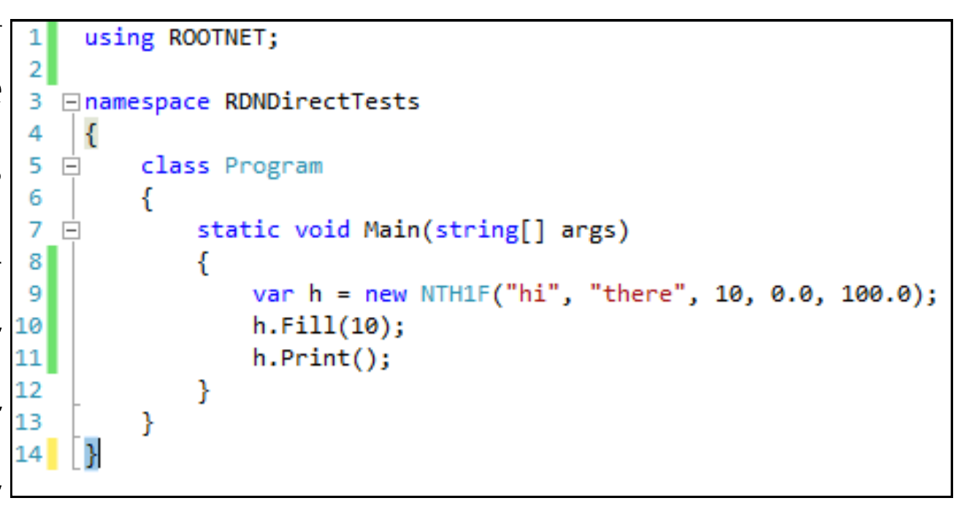
Created by: Gordon Watts
Id: ROOTNET-Dynamic-v5.30.01.win32.vc10
Version: 2.3
View License Terms
Project Information
Description:
Part of the ROOT.NET library that allows you to access the C++ ROOT data anlaysis pacakge (http://root.cern.ch) from .NET languages. Contains libCoreWrapper.dll WrapperPlumbingLibrary.dll, and is only built for this version of root!
Tags: ROOT Data Analysis Science
Dependencies:
No Dependencies

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

There are probably two or three especially important NuGet packages here. First is **Dynamic**. This includes only the classes in libCore, and is less than 2 MB. Once you've added this to your project you can write something like the code to the right. Note the use of the dynamic keyword there. This is non-optimized access to the library, but it is a comparatively small download. The second important NuGet package is **Core**. It contains classes in libCore, libHist, libMath, libPhysics, libRIO, and libThread. If you include that, the same code above can be written as shown at the right—the access will be substantially faster. Others exist for specialized uses. If you install **All** you'll get everything—about 20 MB of libraries.



```
  using ROOTNET.Utility;
namespace RDNTests
{
    class Program
    {
        static void Main(string[] args)
        {
            var h = ((dynamic)ROOTCreator.ROOT).TH1F("hi", "there", 10, 0.0, 100.0);
            h.Fill(10);
            h.Print();
        }
    }
}
```



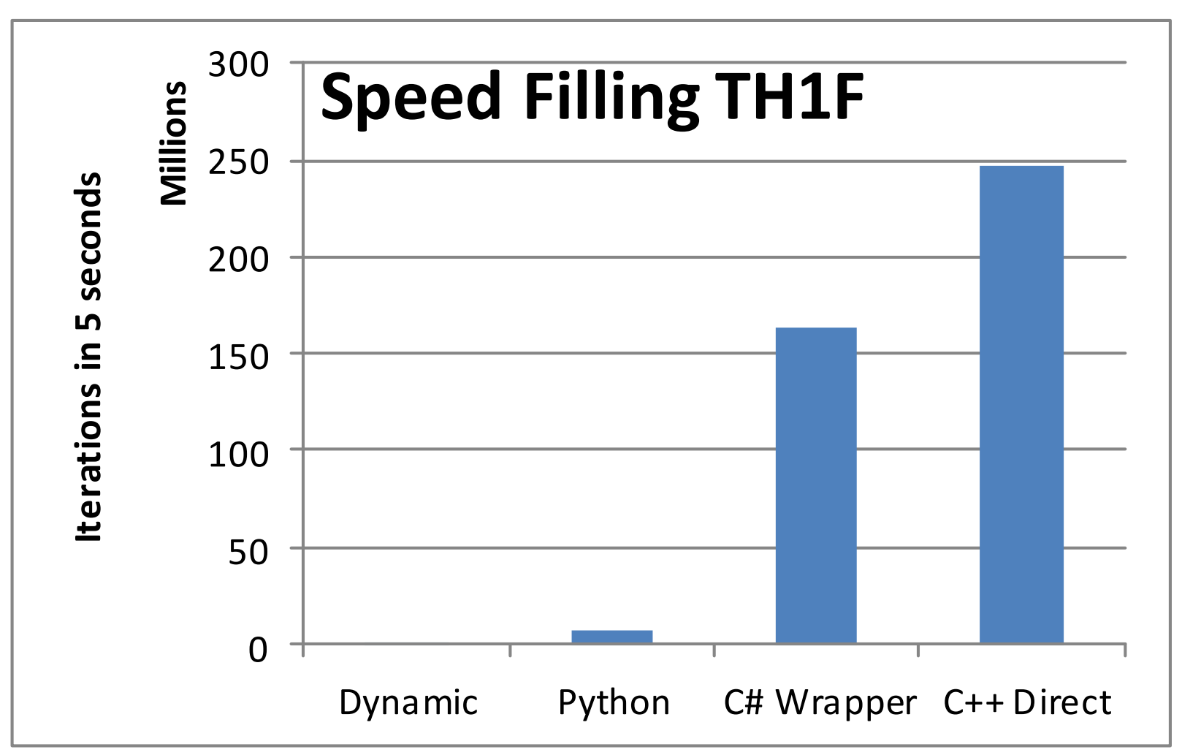
```
  using ROOTNET;

namespace RDNDirectTests
{
    class Program
    {
        static void Main(string[] args)
        {
            var h = new NTH1F("hi", "there", 10, 0.0, 100.0);
            h.Fill(10);
            h.Print();
        }
    }
}
```

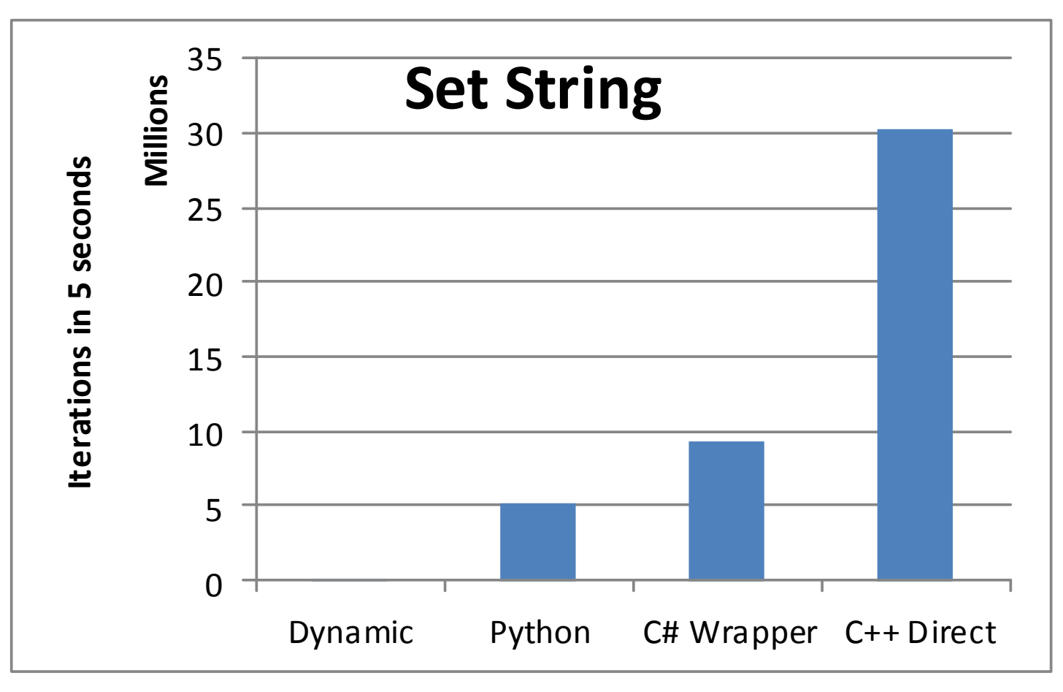
Performance

Each time ROOT.NET fields a 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), 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 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 web address. Dynamic refers to using my dynamic coding. The current implementation is horrible, and involves many object allocations on each call (the number is 110K iterations in 5 seconds). The full C# wrapper is about 2/3's the native C++ rate.



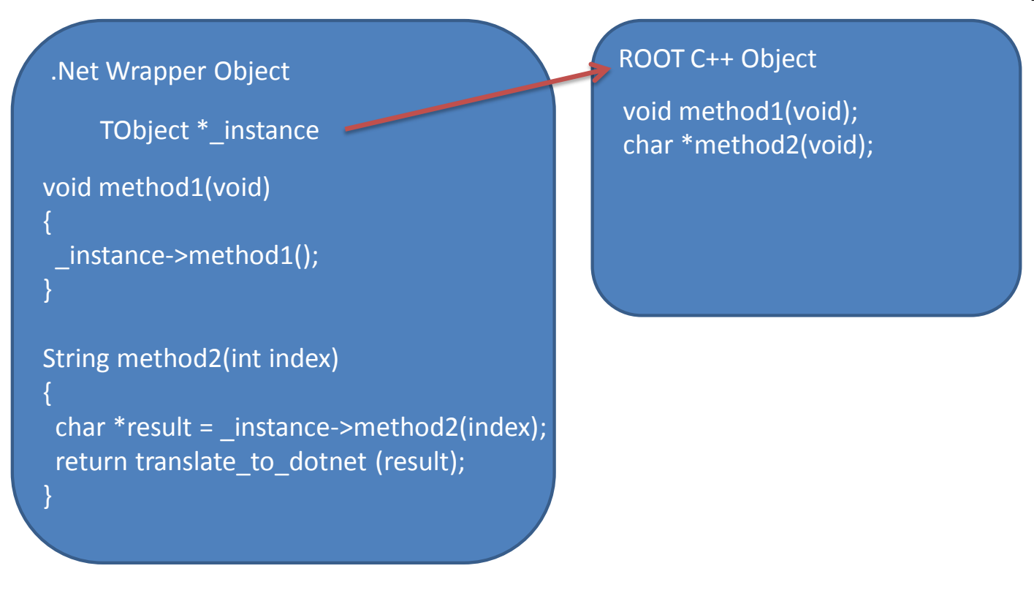
A second quick test was done setting the bin labels in TAxis. The code is very similar—a loop and calling TAxis::SetBinLabel. Again, the dynamic implementation is almost invisible at this scale: 125K iterations in 5 seconds. The C# wrappers aren't much better than python as compared to C++. This is because 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.



Other operations—like passing a ROOT object as an argument to a ROOT method — fall somewhere in between. In the particular case of a ROOT object pointer, the performance looks very similar to the speed filling of the TH1F.

Generating the Wrappers

This is actually quite simple. The devil, as always, is in the details.



1. Load up all ROOT shared libraries that contain objects we wish to convert.
2. Use CINT to scan the libraries to find all the classes, globals, enums, and typedef's
3. For classes, define a wrapper class with a pointer to the real ROOT class, and for each method translate the arguments to C++, call the method, and then possibly translate the result back.
4. For enums generate a .NET enum that directly has the values of the C++ enum.
5. For global variables define a class static variable. .NET does not allow static variables at the global scope, so to access gROOT one has to type TROOT.gROOT.

The devil, however, is in the details..NET's memory management is very different from C++. First, .NET objects can move, and, second, they are garbage collected instead of deleted. Further, it is possible for ROOT to return the same object to the Wrapper through multiple paths (think TSystem::GetObject) — but they had better translate to the same wrapper object! To accommodate the translation each ROOT object is registered and tracked and a new wrapper is created only if the ROOT object is truly new. Second, if a wrapper is garbage collected the ROOT object is deleted. And third if ROOT deletes a object (e.g. when a TFile closes and an in memory object is deleted) the ROOT.NET system is notified and the wrapper is marked invalid—causing an exception if it is accessed. There are circumstances where this isn't enough—ROOT has some very complex object ownership rules. Enough control is provided for these hopefully rare cases.

.NET only allows objects with single inheritance. ROOT makes heavy use of multiple inheritance. Further, inheritance in ROOT is important to its use (think of the TAttLine and TAttMarker, etc, that TH1 inherits from). Multiple inheritance in .NET can be simulated with a concept called an *interface*. A single .NET object can inherrit from a single other .NET object and multiple interfaces. Every ROOT object has a matching interface. The ROOT.NET wrapper inherits from all the interfaces—and implements them all. This works quite well other than having to work around some subtle differences in inheritance rules between C++ and .NET. All ROOT methods expecting other ROOT objects are implemented as interfaces. This does have the unfortunate side-effect of forcing the user to move between the interface and ROOT object. Timing tests were done to verify this added layer of complexity had no effect on the wrappers speed.

Dynamic

In C# version 4.0 the dynamic keyword was included. This allows for late-binding: not knowing what object you are operating on at the time of compilation. This is exactly what is used by python, and the pyROOT infrastructure takes advantage of this. Even in the case of the strongly typed wrappers described here, dynamic wrappers prove useful in two cases: if an object unknown by the wrappers must be operated on, or if one wishes to have only a minimal library.

Microsoft's DLR (Dynamic Language Runtime) 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.

The static wrappers have some features of convenience. For example, GetXXX/SetXXX methods are turned into .NET *properties*. Collections can be iterated over by LINQ and *foreach*. The dynamic interface doesn't implemented those features yet. There are other missing features: for example only objects that derive from TObject are supported in the implementation.

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 lurking.