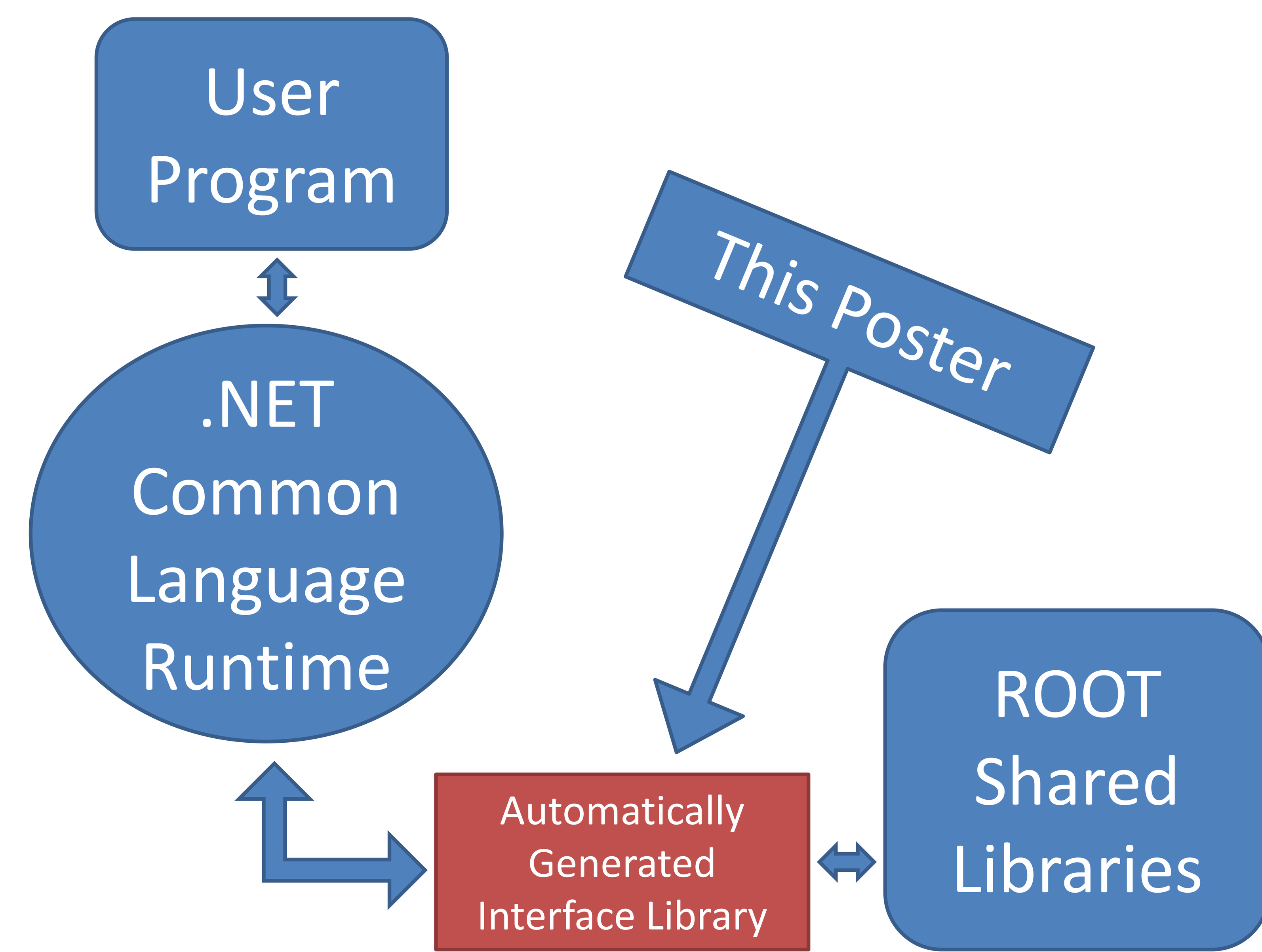
Many thanks to Axel Naumann (CERN/ROOT) & Wim Lavrijsen (LBL/pyROOT) for advice, discussion, and sugges-

ROOT.NET: Making ROOT accessible from CLR based languages

Gordon Watts

University of Washington

#470



Abstract

ROOT.NET provides an interface between Microsoft's Common Language Runtime (CLR) and .NET technology and the ubiquitous particle physics analysis tool, ROOT. This tool 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). Dynamic languages based on the CLR can be used as well, of course (Python, for example). A first attempt at integrating ROOT tuple queries with Language Integrated Query (LINQ) is also described. This poster will describe the techniques used to effect this translation, along with performance comparisons, and examples. All described source code is posted on SourceForge.

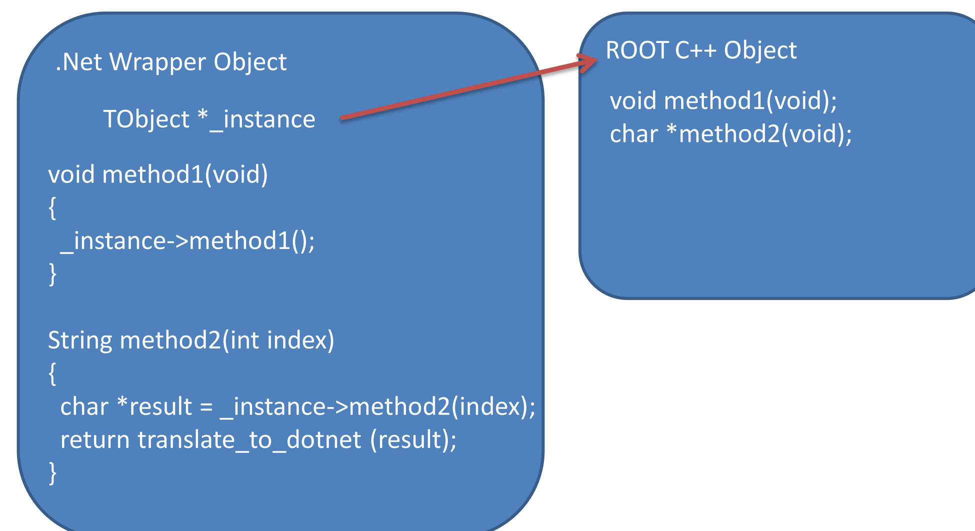
Why Marry the ROOT and .NET Worlds?

- Initial motivation was driven by a number of small projects at the DØ experiment, located at the Tevatron accelerator at Fermilab. I found myself writing custom wrappers for several of these projects to expose small amounts of ROOT functionality to the .NET language runtime. These hand-written wrappers were fragile and gave very limited access to ROOT's features. Passing ROOT objects to ROOT methods of other ROOT objects was always very difficult; I found myself making minimal use of ROOT's functionality to avoid the extra work.
- I also had a huge number of python scripts to manipulate physics plots. I find python much easier to write than C++, so I've built up quite a library. But they are slow. I wondered if I could do better with something like this project.
- I like to call C++ a dead language—not much research is going into improving it. It is, in part, a victim of its success. Duck typing, lambda expressions, functional programming are all improving how we code. Many of these languages run on top of the Java and .NET language run-times – it would be nice to have ROOT easily accessible from these languages. The proposed new C++ standard (0x) should significantly improve C++'s usefulness, however, I still see language research moving forward more quickly for these languages based on a run-time.
- 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!
- Since I've started this project I've become active on the ATLAS experiment, based at the LHC at CERN. This is a much larger experiment than DØ. And unfortunate byproduct of this is their light-weight analysis framework (Athena ROOT Access (ARA)) is large—gigabytes after compilation – and contains many external dependencies. Too many to make it platform neutral. As a result this tool has turned into something more interesting for building small GUI based tools that access ROOT.

General Procedure

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
   - 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.



Why not use the PyROOT Method?

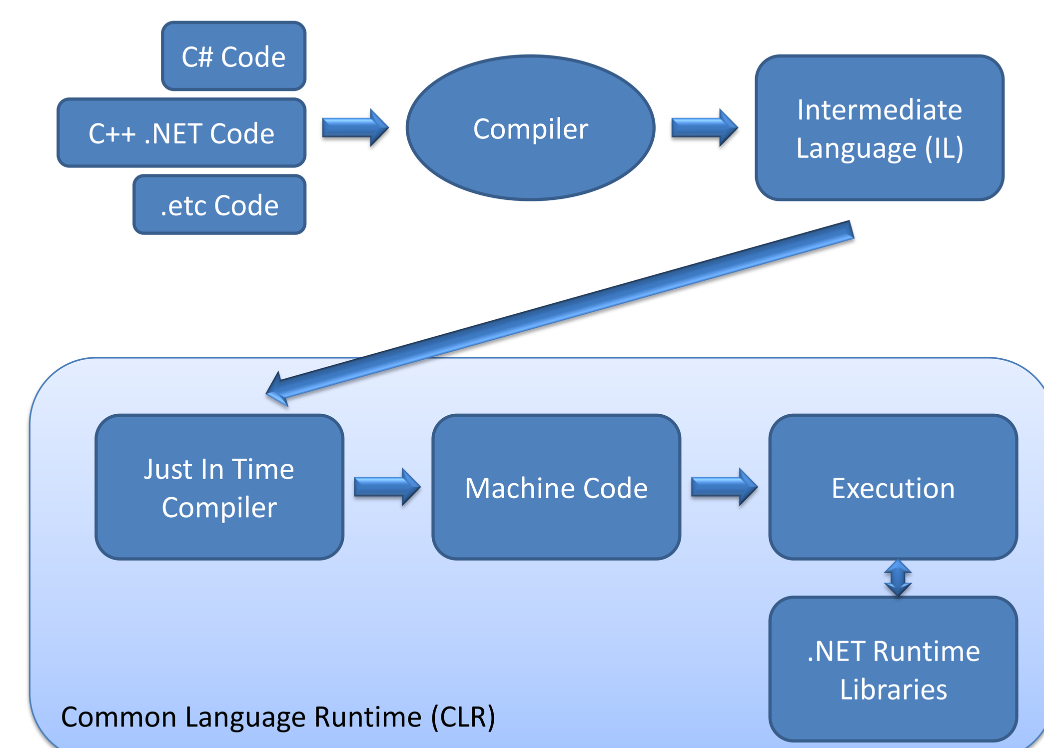
PyROOT is fantastic. It is shipped with a small set of wrapper and management classes. In python when you first access a ROOT object it, behind the scenes, builds all of the wrapper code it requires. ROOT.NET requires you to run a complete translation of the whole library before you can even start writing code. This also has the (good or bad, depending) side effect of the ROOT.NET wrappers being bound to a particular version of ROOT.

This is the difference between an early and late binding language (static vs. dynamic). C++ is the same as .NET—it is an early binding language: the compiler must know everything before it can compile your code. Python and other dynamic languages (like Ruby or Visual Basic) build the calls on-the-fly. There is a cost to this, of course: speed. Each time a call is made the Python wrappers have to lookup the method that is getting called. That lookup cost occurs every time one makes the leap from Python to ROOT.

Dynamic languages and their implementations on top of platforms like the CLR have received a lot of research in recent years. The result is there are some very interesting methods to combine the best of both worlds—the flexibility of the dynamic world with the speed of the static world – using the JIT that is part of the runtime. The next version of C# will incorporate some of these advances and this issue can be reexamined at that time.

What is .NET?

- .NET is Microsoft's answer to Java. Just like Java, languages are compiled to a byte code and then a runtime compiles them to actual machine code through a Just-In-Time (JIT) compiler. Also, just like Java, it includes a full framework library for GUI and computational needs. The byte-code and some of the languages that run on it have been standardized in internationally recognized standards bodies. This code was written by post-security conscious Microsoft and has many security constructs built directly into the runtime.
- Platforms: besides Windows, there is an open source project, Mono, that runs a complete version of the bytecode and has most of the framework implemented. This runs on Linux, Mac, and many other platforms. Microsoft has also recently released a version of the CLR that runs on the Mac as well as Windows (Silverlight)
- Which is better? I've seen many persuasive arguments going both ways. I was originally attracted to .NET because the C# language felt much more familiar to me (as a C++ programmer) than Java. I've not really looked back.
- How fast is it? It isn't as fast as raw, optimized, C++, but it isn't far behind. There are plenty of interesting benchmarks testing all sorts of different scenarios that can be found on the web.



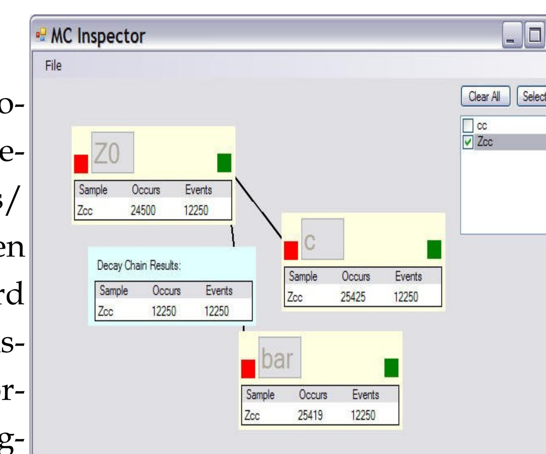
Translator Design: Some Devil-in-the-Details

The translator must smoothly integrate the .NET and C++ worlds. There are several crucial differences between the two worlds that make this more than a trivial exercise of generating a wrapper around each ROOT object.

- .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.

The pyROOT method can't be used: C#, for example, is a late binding language and needs the complete interface specified ahead of time – classes, global variables, etc. Python is late binding and so one can build the bindings as one needs them (see the box on differences between Python and C++ and C#, etc.).

Status



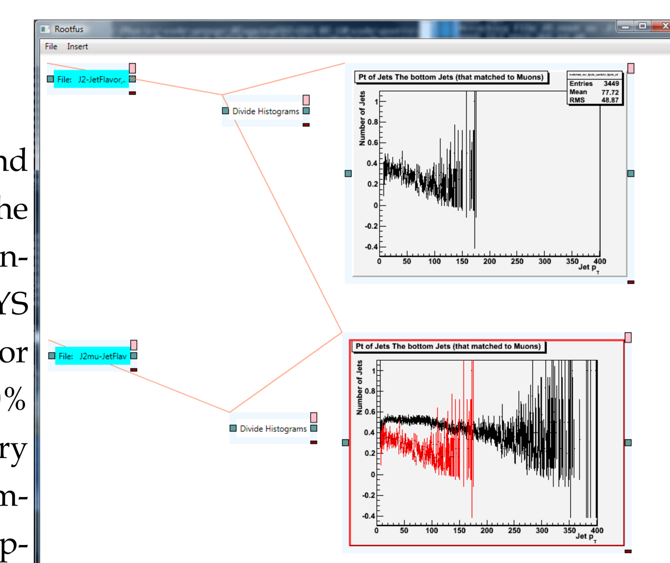
This project was first presented as a prototype at CHEP 2007 and released on sourceforge (http://sourceforge.net/projects/rootdotnet/). Since then there have been several updates but development forward has been stuck because the code that translates the C++ object and inheritance information from CINT into .NET was very fragile.

Over last summer and early fall a new version of this code, complete with about 60 unit tests, was written. This new code turns on its head how the object structure is interpreted and is, in the end, much cleaner. It also runs in about 20% of the time of the old code. These improvements are checked into CVS at sourceforge. Extra unit tests are still being written as more edge cases are found.

Code to embed graphics easily in WPF or WinForms applications has also been written. This turns out to be almost trivial (it would not have been trivial without the help of B. Bellenot of the ROOT team). WPF and WinForms are two popular application frameworks on Windows.

Examples of some small applications I've written can be found at right. The upper one is an MC explorer I wrote. By entering decay chains one can determine the fraction of time the decay chains happen in loaded MC samples. The lower one was an attempt to GUI'fy the scripts we usually write to generate plots at the end of our analysis (divide, multiply, add, scale, fit, etc.). You can see the graphics displayed on a WPF surface here. My inability to do GUI design clearly shows through here.

Future Efforts



The next project is a small stand alone application that will build the wrappers. After downloading, running it and pointing to a ROOTSYS directory it will build wrappers for everything in it. This can't be 100% automated yet, but it could be very close. This should make it a lot simpler for people to use these wrappers. I've gotten a surprising number of requests for ROOT.NET's pre-built libraries. Another possible project along these lines will be to create a Visual Studio project that will automatically prebuild just the libraries you need. This should reduce distribution requirements greatly.

The wrapper libraries can be quite large: they contain a great deal of duplicated code. For example, the wrapper code to call TH1:Draw exists in the TH1, TH1F, TH2, TH2F, etc., wrapper objects. There is no need for this other than fixing the code up. I expect this to reduce the size of the ROOT.NET libraries by over 50% (from initial tests).

The project has not been absorbing nearly as much of time free time as it used to because ATLAS does not have an analysis framework that is portable to the Windows platform.