Celeriac: A Daikon .NET Front End

June 3, 2013

1 Introduction

Daikon is a tool that infers likely program invariants from program traces [2]. The Daikon tool relies on front ends, such as the Chicory front end for Java, to generate program traces to use for input. There are currently no available front ends for the .NET languages – C#, VB.NET, and F#.

The purpose of this document is to describe the architecture of a Daikon front end for .NET. It will be updated throughout the development process as to accurately reflect the design and architecture of the system.

2 Design Goals

This section describes the design goals guiding the development of the .NET Daikon front end.

Work with any .NET program. While Celeriac will likely primarily be used with programs written in C#, we leverage the Common Language Runtime architecture to make the front end work with many .NET languages. Celeriac has been successfully used on F# and VB.NET in addition to C#. This compatibility is achieved by targeting the tool at binaries that have already been compiled to CIL, as opposed to targeting source code.

However, Celeriac must specially detect the presence of some language-specific constructs, to enable detection of meaningful properties. As an example of this F# uses its own list type instead of the .NET `System.Collections.List` type. To capture meaningful invariants over lists Celeriac treats a variable as a List if it has the `System.Collections.List` or the `FSharp.List` type.

Don't require modification of the developer's build process. Commercial build processes can become very complex over time. As a result, it may be unwieldy or even impossible to integrate new tools into the build chain. In order to promote tool use, Celeriac operates on the output of the build process: a managed binary executable or library. An added benefit of meeting this design goal is that Celeriac can run on programs even when the source is not available.

Don't modify files on disk. Modifying the contents of the disk to instrument the program, e.g., creating a patched version of the executable, is similarly undesirable. The creation or replacement of binary files opens the door for versioning issues such as accidentally running an instrumented binary with incorrect options.

There are some situations meeting where this goal isn't possible. For example, when create an instrumented library that will be used by other programs the library must be saved to disk. Also, Celeriac is running on GUI programs using Windows Presentation Foundation (WPF), Celeriac must be run in off-line mode. This is because WPF programs load resources in a way that cannot be done in online mode, since the subject program is loaded into the custom launcher's `AppDomain`.

3 Tool Architecture

The front end has two major components: an IL rewriter that inserts instrumentation calls into procedures and instrumentation code (a run-time system) that traverses data structures and outputs information in the Daikon format.

3.1 Profiler

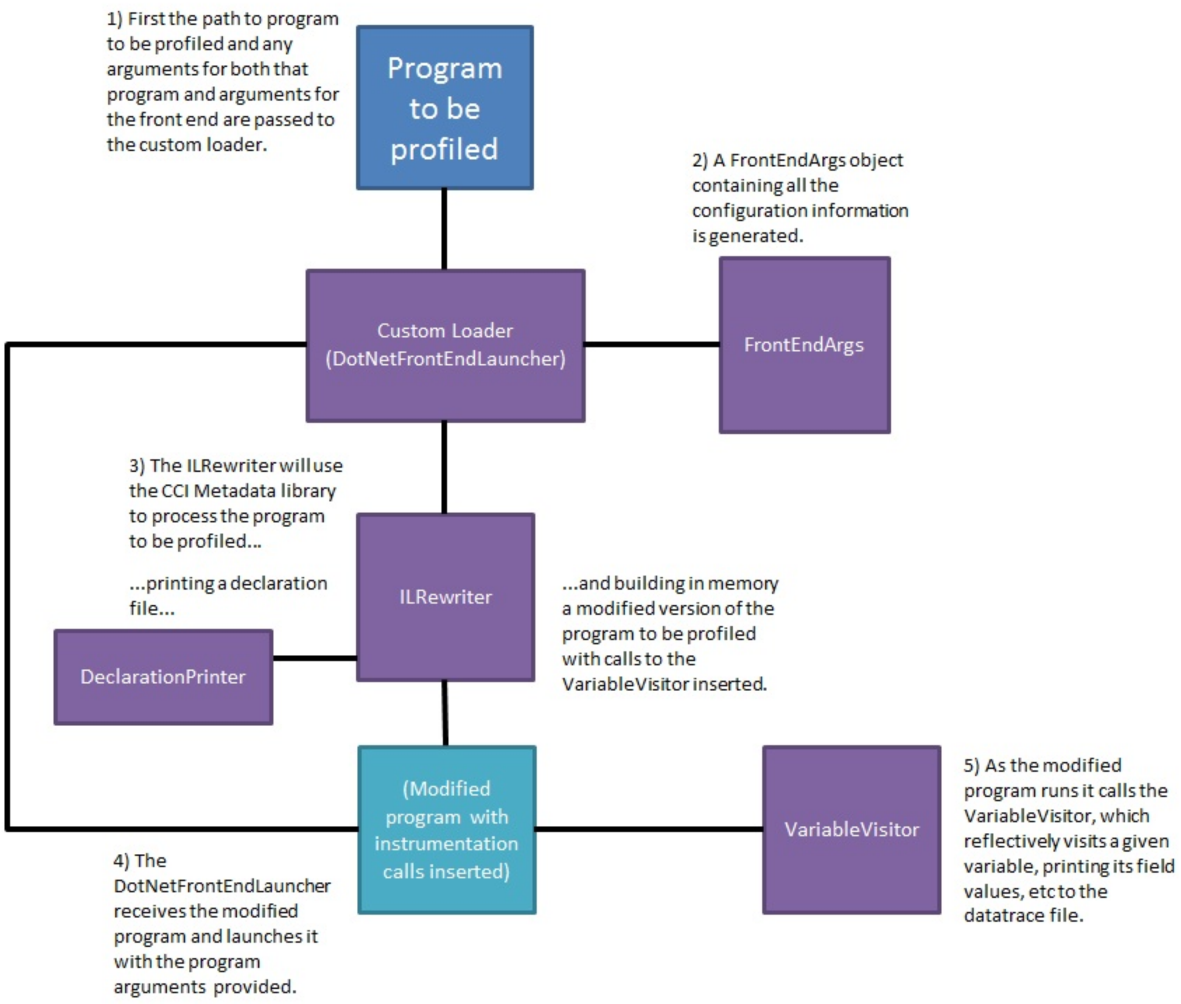
The profiler, written in C#, utilizes the CCIMetadata IL Rewriting library. The CCIMetadata library provides an API for reading and writing MSIL at a higher level than raw bytecode. The profiler component of the application, named ILRewriter, was adapted from a CCIMetadata example program that printed the name of local variables. The profiler loads a .NET Module in memory, walks over the IL and inserts instrumentation calls at the entrance and exits of methods. While it is inserting instrumentation calls the ILRewriter also makes calls to the DeclarationPrinter class which writes the declaration portion of the datatrace file. When finished, the profiler returns a memory stream which the front-end loads and executes.

3.2 Instrumentation Code

The instrumentation code, written in C#, (or another .NET language) and compiled to MSIL, uses reflection to log variable information. The code, written in the VariableVisitor class, is called with the name of the variable, value of the variable, and the assembly-qualified name of the declaring class. The value of the variable itself is printed, along with any static or instance fields of that variable, which are visited reflectively based on the declared type of the variable. The instrumentation code traverses data structures and enumerates variables that implement the IList, ISet, or Dictionary interfaces.

3.3 Use case illustrated

The following figure shows the steps executed during a run of the front-end. The names in parenthesis indicate the class represented by the block.


1) First the path to program to be profiled and any arguments for both that program and arguments for the front end are passed to the custom loader.
Program to be profiled
2) A FrontEndArgs object containing all the configuration information is generated.
Custom Loader (DotNetFrontEndLauncher)
FrontEndArgs
3) The ILRewriter will use the CCI Metadata library to process the program to be profiled...
...printing a declaration file...
ILRewriter
...and building in memory a modified version of the program to be profiled with calls to the VariableVisitor inserted.
DeclarationPrinter
(Modified program with instrumentation calls inserted)
VariableVisitor
5) As the modified program runs it calls the VariableVisitor, which reflectively visits a given variable, printing its field values, etc to the datatrace file.
4) The DotNetFrontEndLauncher receives the modified program and launches it with the program arguments provided.

4 Risks

This section describes the aspects of the project that are the riskiest, that is parts of the project may be difficult to complete during the fall (or at all).

Instrumenting Exception Return Points The instrumentation calls that the unmanaged profiler adds must maintain the validity of the MSIL code. For function prologues, this maintaining validity is apparently easy. For intermediary return points and exceptional return points, this is not trivial [3].

Detecting / Instrumenting Complex Language Features Certain language features, such a closures, may be compiled into weird constructs in the MSIL. It may be hard or impossible to detect all of these features in the MSIL code an instrument them accordingly.

A Appendix: Project Schedule and Milestones

This section describes the project milestones and history.

A.1 10/2010 - 12/2010

Work began on the instrumentor this quarter. A first attempt was to use the unmanaged Profiler API. First C# reflection was used to analyze the program, and then the Profiler API's function enter/leave hooks were used to output the appropriate variable information. Manually editing the MSIL was extremely difficult and error-prone, especially for methods with multiple exit points and exception handling, so this approach was abandoned.

A pure IL rewriting approach using the CCIMetadata Library was attempted, where the output program was saved to disk as a new executable, however this violated the Disk Modifications Avoidance design goal. A hybrid of the previous two methods was considered where the CCIMetadata Library would create the new MSIL, and this would be handed to the unmanaged Profiler, to be substituted for the original method IL.

A presentation of the work to this point was presented to the UW CSE PL&SE group on December 13th.

A.2 1/2011 - 3/2011

The previously described method was almost immediately shown to be infeasible. The CCIMetadata rewritten code produced different method tokens than were present in the original program, and thus could not be used. The main developer on the CCIMetadata project confirmed this issue could not be resolved in context. Analysis of alternate methods for IL rewriting was completed. It was determined that using the program with instrumentation inserted by CCIMetadata, but loaded into memory and ran instead of written to disk was the best approach.

The remainder of the quarter was spend implementing the profiler and visitor, and getting them to conform to Daikon specification. At the conclusion of the quarter an alpha version of the front end was functional, and could generate simple datatrace files that Daikon could process.

A.3 4/2011 - 6/2011

During this quarter reflective visiting and array-style visiting of IEnumerable types were implemented. Most command-line arguments available to Chicory were documented. A test suite was added, modeled off the Kavasir (C++) front-end test suite. Chicory test sources were transliterated from Java to C# and ran through the front-end. Output invariants were compared to the invariants produced by Chicory, and bugs in the front-end were fixed to rectify the differences. Exception handling was also added, a feature new to Daikon front-ends.

A.4 4/2012 - 6/2012

A.5 Introductory Milestones - From the 10/2010 - 12/2010 section

A.5.1 Read the First Three Sections of the Daikon User Manual

Read the first three sections of the Daikon Developer Manual [2]. Perform the Java `StackAr` example to get an idea of what Daikon can do and what it is like to use Daikon.

Estimated Completion Date:	10/4/2010	Estimated Completion Time:	2 hours
Actual Completion Date:	10/5/2010	Actual Completion Time:	2 hours

A.5.2 Read the "New Front Ends" Section of the Daikon Developer Manual

Read the "New Front Ends" section of the Daikon Developer Manual [1]

Estimated Completion Date:	10/6/2010	Estimated Completion Time:	1 hour
Actual Completion Date:	10/6/2010	Actual Completion Time:	1 hour

A.5.3 Read the File Formats Appendix

Read the "File Formats" appendix of the Daikon Developer Manual to get a clearer idea of the information that the front end will have to output.

Estimated Completion Date:		Estimated Completion Time:	
Actual Completion Date:		Actual Completion Time:	

A.5.4 Install Visual Studio 2010 Ultimate Edition from MSDNAA

Download and install a copy of Visual Studio 2010 Ultimate Edition form the MSDNAA website. It is available for free. We have to use Visual Studio 2010 Ultimate Edition in order to use Microsoft Research's tools (e.g., the code contract framework).

Estimated Completion Date:	10/6/2010	Estimated Completion Time:	1.5 hours
Actual Completion Date:	10/6/2010	Actual Completion Time:	1.5 hours

A.6 Profiler Milestones

A.6.1 Hello World

Build an unmanaged profiler that logs "Hello World" to a file specified by the user via the command line whenever a program is run using the profiler.

Estimated Completion Date:	10/17/2010	Estimated Completion Time:	7 hours
Actual Completion Date:	10/17/2010	Actual Completion Time:	8 hours

A.6.2 Function Names

Modify the profiler to log the name of each function that is compiled by the JIT right before they are compiled.

Estimated Completion Date:	10/25/2010	Estimated Completion Time:	15 hours
Actual Completion Date:		Actual Completion Time:	

A.6.3 Function Names 2.0

Modify the profiler to insert a prologue in each function that outputs the name of the function to the log whenever it is called. We decided we didn't need to do a prologue after all.

Estimated Completion Date:	Estimated Completion Time:
Actual Completion Date:	Actual Completion Time:

A.6.4 Parameter Names

Modify the profiler to insert a prologue in each function that outputs the names of the function parameters to the log whenever it is called. Didn't end up using a prologue, instead use reflection to output function parameter names to a file, then parse those in the profiler.

Estimated Completion Date:		Estimated Completion Time:	
Actual Completion Date:	11/11/2010	Actual Completion Time:	22 hours

A.7 Instrumentation Milestones

This section was left intentionally blank

References

[1] *The Daikon Invariant Detector Developer Manual*, September 2010. `http://groups.csail.mit.edu/pag/daikon/download/doc/developer.html`.

[2] *The Daikon Invariant Detector User Manual*, September 2010. `http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html`.

[3] Aleksandr Mikunov. Rewrite msil code on the fly with the .net framework profiling api, September 2003. `http://msdn.microsoft.com/en-us/magazine/cc188743.aspx`.