Celeriac: A Daikon .NET Front-End

Developer Manual

June 14, 2013

1 Introduction

Daikon is a tool that infers likely program invariants from program traces [1]. The Daikon tool relies on front-ends, such as the Chicory front-end for Java, to generate program traces for input. Celeriac is the first Daikon front-end for the .NET languages — C#, VB.NET, and F#. This document describes the architecture of Celeriac, a Daikon Front-End for .NET.

1.1 Instrumented Program

Celeriac creates a binary with instrumentation calls added, which will produce a Daikon trace file (described below) when executed. Celeriac can be configured to execute the instrumented program directly from memory (*online-mode*), or to save the instrumented program to disk (*offline-mode*).

1.2 Daikon Declaration File

The first part of a Daikon trace file is called the declaration component, which is metadata describing the schema of each program point (method entrance or exit). While Celeriac is inserting instrumentation calls the `ILRewriter` (described in section 3.1) also makes calls to the `DeclarationPrinter` class which writes the declaration portion of the trace file, as described in Section 3.2.

1.3 Daikon Trace File

The second component, the trace component, is not output by Celeriac directly. Instead it will be written by the program as it executes. When the IL rewriter is finished, it returns a memory stream which the front-end loads and executes, or saves to disk if desired.

2 Design Goals

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

2.1 Work with any .NET program.

While Celeriac will likely be used mainly with C# programs, 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 programs. This compatibility is achieved by targeting the tool at binary programs (which are in 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. For example, F# uses its own list type instead of the .NET `System.Collections.List` type. To capture meaningful invariants over lists, such as the length of a list, or whether its elements are sorted, Celeriac treats a variable as a List if it has the `System.Collections.List` or the `FSharp.List` type.

2.2 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. An added benefit of meeting this design goal is that Celeriac can run on programs even when the source is not available, although the debugging symbols are necessary.

2.3 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. Celeriac solves this problem by modifying the assembly, specifically adding the instrumentation calls, in memory and then executing the modified assembly directly from memory.

There are some situations where meeting this goal isn't possible. For example, when Celeriac creates an instrumented library that will be used by other programs, the instrumented library must be saved to disk. Also, when Celeriac is run on GUI programs using Windows Presentation Foundation (WPF), Celeriac must be run in off-line mode. This is because WPF programs resource loading process cannot complete when the program is in online mode, since the subject program is loaded into the Celeriac launcher's `AppDomain`. The program will then attempt to resolve resources from the Celeriac launcher's resources, but the resources exist only in the WPF application.

3 Tool Architecture

The front-end has two major components: an IL rewriter that inserts instrumentation calls into procedures and a run-time system that, when called from the subject program, outputs variable values to construct a Daikon trace file in the Daikon format.

3.1 IL Rewriter

The IL rewriter is implemented in `ILRewriter.cs` and rewrites .NET binaries by inserting calls to the Celeriac runtime library. It utilizes the CCIMetadata IL Rewriting library[1]. The CCIMetadata library provides an API for reading and writing CIL code using C#. CCI also handles resolution of class and method references. The rewriter loads a .NET assembly in memory, walks over the IL and inserts instrumentation calls at the entrance and every exit of methods. The following figure illustrates the IL rewriter component.

[1]https://ccimetadata.codeplex.com/

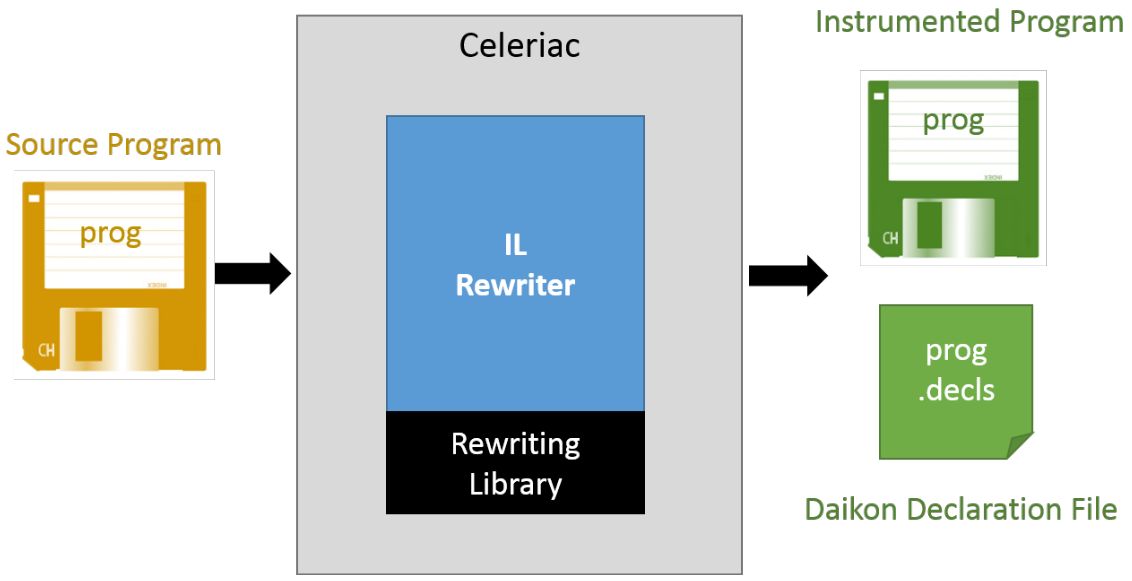

Figure 1: The IL Rewriter adds instrumentation calls to the source program and outputs expression metadata to be consumed by Daikon

3.2 Declaration Printer

While performing IL rewriting to create an instrumented program Celeriac simultaneously builds the declaration component of the Daikon trace file. This is handled in the `DeclarationPrinter` class. Whenever Celeriac inserts an instrumentation call for a method, say `foo()`, it also writes to the declaration file a program point for the entry `foo()` and a listing of its parameters if any, and the receiver's fields and pure methods.

3.3 Runtime Library

The runtime library creates the actual trace file that will be input to Daikon. The main method exposed to source programs prints the name of a variable, and its value, then prints the name and value of each of that variables fields, pure methods, and elements if any. The following figure illustrates the functionality of the runtime library component.

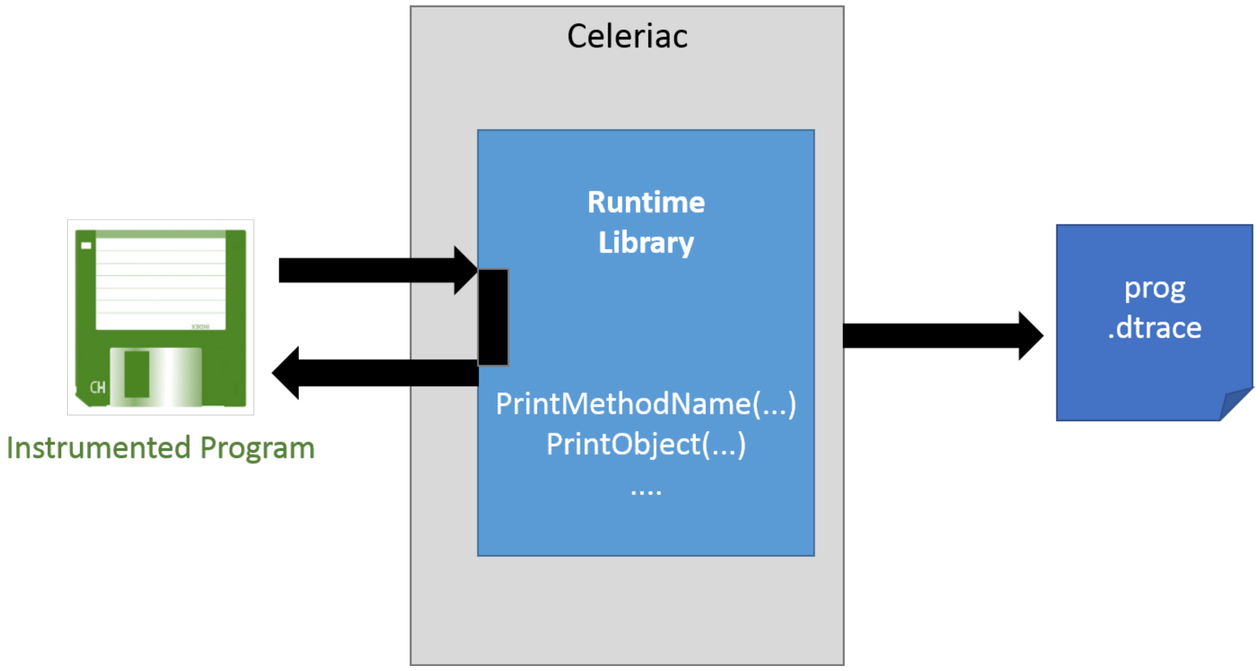

Figure 2: The runtime library contains instrumentation methods that are called by the modified source program. When called the methods write entries to a trace file, to be consumed by Daikon. ~

The runtime library is implemented in the `VariableVisitor` class. When the `VisitVariable()` method is called from the instrumented program the value of the variable itself is printed. Then any static or instance fields of that variable, which are visited reflectively based on the declared type of the variable, are also printed. The instrumentation code traverses data structures and enumerates variables that implement the IList, ISet, or Dictionary interfaces, or their F# equivalents.

4 Design Decisions

This section describes important design decisions, the alternatives, and the justification for the implemented choice.

4.1 Type Resolution

A Daikon front-end needs to print out each the same list of fields and pure methods for a variable fields at every instance of a program point. This is a requirement of Daikon, and exists because Daikon outputs invariants that are always true. If a variable were present at some instances of a program points but not others then Daikon would not be able to reason about that variable.

This means a front-end needs to know the declared type of each variable, rather than the run-time type, which may be different each time a method is called, and not available at instrumentation time, when the declaration file is printed.

There are several possible approaches for identifying the declared type of each variable:

- Inspect the source code as text, and manually parse the program to learn each expression's type.
- Mimic or operate with the code compiler, since it must already determine each expression's type, and to use this result.
- Extract declared types from a binary program using a bytecode or metadata library.
- Perform reflection on the binary program, in a structured search to match field names to a list of fields extracted via reflection, which would contain their declared types.

Celeriac uses the metadata library approach. Celeriac uses the CCIMetadata library [3] to determine the types of each variable, albeit in a type system constructed by CCI, that does not have information, such as the fields of the type. The fields of the type are available in the .NET Type, so a bridge between these two type systems must be created. This is accomplished through the use of assembly-qualified name [2]. An assembly-qualified name uniquely identifies a type, and allows the Type to be retrieved at run-time.

The complication with this approach is to build the appropriate assembly-qualified name given a CCIType. This especially problematic for generic types that may have multiple bounds, e.g. `public addEdge<TEdge>(TEdge edge) where TEdge : IEdge<TVertex>, IComparable<TEdge>, IPrettyPrintable`. Generics are reified at runtime in .NET, so at runtime a type `TEdge` that implements all necessary interfaces would be created, but this type isn't available at instrumentation time, when it is needed by Celeriac. The class `CeleriacTypeDeclaration` compensates for this, and can be constructed so that a declared type is actually of a list of types, which are a list of constraints on a generic expression. In the code presented above the declared type of the parameter `edge` would be a list of its generic constraints: `IEdge<TVertex>, IComparable<TEdge>, IPrettyPrintable`.
The implementation of the type resolution, which is in `TypeManager.cs`, was difficult to complete, with bugs appearing over many months of the development. The difficulty resulted from the complex structure of CCI's representation of the .NET type system. The CCI type system does not correspond closely to the .NET one for the purposes of creating assembly qualified names. For example, types of class generics and method generics appear the same in assembly qualified names, but are very different C# types. Running Celeriac on the test suite for the Mono C# compiler[2] provided high confidence that the resulting type resolution system is correct.

We chose the metadata approach over alternatives because it fit best with the design goals (discussed earlier in Section 2. Source code parsing alternatives would not be feasible because they fail the design goals of allowing Celeriac to run on multiple languages and being build processes independent (since they require source code). Compiler interop would also fail the build process independence goal, and would be infeasible as well because most .NET code uses Microsoft's proprietary compiler.

[2] `http://www.mono-project.com/CSharp_Compiler`

4.2 Instrumentation Granularity

Celeriac inserts instrumentation calls to print out all appropriate fields for a given object. This means Celeriac inspects each object's type twice, once at instrumentation time to print the declaration file, and again at runtime. This process involves examining an object's type, determining which fields to print, and then printing either a declaration (when the program is instrumented) or printing the value (when the program is run).

An alternative would insert calls to print the field values directly, and perform no inspection at runtime. The two figures below illustrate the practical difference.

```
public void compare(Point p1,
                    Point p2) {
        PrintObject(this);
        PrintObject(p1);
        PrintObject(p2);
        ... Client Code ...
}
```

Figure 3: Instrumentation pseudo-code: the variable visitor is called for each parameter, including the receiver, but not directly on a variables's fields.

```
public void compare(Point p1,
                    Point p2) {
        PrintField(this, foo);
        PrintField(this, bar);
        ... More Visitor Code ...
        ... Client Code ...
}
```

Figure 4: Alternative Instrumentation pseudo-code: the variable visitor is called for each parameter and variable's fields directly.

However, this choice was not implemented for three reasons. First, it would lead to a much greater increase in code size. Whereas Celeriac inserts 1 new method for each relevant variable (parameter, field, etc.), the proposed alternative would insert 1 new method for each new variable's fields, resulting in much larger instrumented programs. It would require many more context-switches between the client's program and Celeriac's runtime library. Finally, much more work being would be done in IL rather than managed C#, and C# is much easier to develop in.

References

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

[2] Microsoft. Type.assembly qualified name property, .NET Framework 4.5. `http://msdn.microsoft.com/en-us/library/system.type.assemblyqualifiedname.aspx`.

[3] Microsoft Research. `https://ccimetadata.codeplex.com/`.