daikon-dot-net-front-end

Celeriac .NET Front-End for Daikon

CeleriacOnUnitTests

Generating traces with from a unit test suite

Phase-Deploy, Users

Updated Today (9 minutes ago) by melonhead901

Introduction

A great way to produce a trace is to run Celeriac in conjunction with a unit test suite. Celeriac is compatible with all testing frameworks since Celeriac instruments the program under test and not the test suite. We've run Celeriac using the Visual Studio, MSTest, xUnit.net, and NUnit frameworks as well as numerous mocking frameworks such as RhinoMock.

The general approach is to create an instrumented copy of the assembly under test by running Celeriac with the `--save-program` flag. To run the assembly that isn't part of the test suite (i.e. your program or library), you will then either replace the original assembly with the instrumented copy, or modify the test suite setup to use the instrumented assembly. The test suite can then be run as usual, and the Celeriac output files will be generated as they normally would during offline tracing.

- Introduction
- From the Command Line or External Testing GUI
  - MSTest Configuration
- From Visual Studio
- Best Practices
  - Inferring Argument Validation

From the Command Line or External Testing GUI

Suppose that the assembly under test is `Bank.dll`, and the test suite is `BankTest.dll`. To run the test suite:

1. Create an instrumented copy of the assembly under test (see the Getting Started guide).
2. Replace the original assembly under test with the instrumented assembly
3. Run the test suite as normal. For example, when using MSTest:

```
MSTest.exe /testcontainer:BankTest.dll /testsettings:Local.testsettings
```

Some testing frameworks create a new application domain for each test by default. To prevent Celeriac from creating a new trace file for each test, configure the test runner to run the tests in a single domain.

MSTest Configuration

Purity Files

MSTest copies files to another directory in order to perform testing. Therefore, when using a purity file, you must specify that the purity file should be deployed with the tests.

For example, when running Celeriac with the QuickGraph library test suite, we add a `DeploymentItem` entry for the purity file to `LocalTestRun.testrunconfig`:

```
<Deployment>
   <DeploymentItem filename="QuickGraph.Tests\bin\Debug\QuickGraph.pure" />
   <DeploymentItem filename="QuickGraph.Tests\GraphML\" />
</Deployment>
```

From Visual Studio

Suppose that the assembly under test is `Bank`, and that the test suite is `BankTest`.

1. Create an instrumented copy of the assembly under test (see the Getting Started guide).
2. In Visual Studio, navigate in the Solution Explorer to the BankTest project and expand the `References` folder
3. Delete the reference to the original project (e.g. `Bank`)

4. Add a new reference to the instrumented assembly. This is accomplished by right clicking on the `References` folder, clicking `Add Reference`, clicking the `Browse` tab, and then navigating to find the program saved previously (`InstrumentedProgram.exe`, by default).
5. Run the test project as usual.

Best Practices

This section describes best practices for inferring contracts using a test suite.

Inferring Argument Validation

To infer argument validation, test cases should not be run that intentionally pass an invalid input to the method under test (i.e., the client should exhibit correct behavior). Consider, for example, the following method:

```
public void MethodUnderTest(object x){
  ...
  x.Method();
  ...
}
```

You should not run a test that provides a `null` value and then checks that a `NullPointerException` is thrown. If `null` is observed as a value for parameter `x`, Daikon will not infer that `x != null` is a valid invariant.

However, for non-argument validation exceptions, testing inputs that cause an exception is valuable to inferring exceptional postconditions (which can be written using the `Contract.EnsuresOnThrow<T>(...)` method). Note that in these cases, the client is behaving properly, but the method under test will still throw an exception (e.g., because a resource cannot be accessed).

CommandLineOptions

A listing of the command-line options for the Celeriac

Phase-Deploy, Users

Updated Today (8 minutes ago) by melonhead901

- Instrumentation Control Options
- Program Point Tracing Options
- Variable and Method Tracing Options
- Variable Metadata Options
- Miscellaneous Options
- Program Analysis Commands and Options

Instrumentation Control Options

--assembly-name=*assembly*

Specify the name of the assembly to be instrumented. If not specified, the assembly name will be inferred as the name of the provided assembly, preceding any extension. Thus this flag is is only necessary if the program's executable has an unusual name.

--save-program=*filename*

Instead of executing the assembly, save it as *filename*, or `InstrumentedProgram.exe` if *filename* is not supplied. This new program can then be executed, and will write its output as normal. This is necessary for class libraries and testing. It's also useful for debugging purposes, to examine the program with all instrumentation calls inserted, or to verify the bytecode.

--save-and-run

When supplied with `--save-program` or `--wpf`, will execute the program immediately after rewriting, with the user provided arguments. This is done just as it would be if the program were run in online-mode. This option has no effect if neither --save-program or --wpf are provided. It should not be used on a class library, since a class library has no EntryPoint. This flag is false by default

--wpf

Similar to the --save-program argument, except the output program is saved over the input program. This is useful for WPF programs, and any others that requires assembly resources to have specific names. The existing input program will be saved to a new location in the same folder with .bak added at the end of its filename. Will override --save-program if both are supplied. This flag is false be default.

--portable-dll

When supplied, specifies the use of a custom metadata host instead of the default one. This flag is only necessary if the source program was built with a system library other than .NET, for example Portable .NET.

Program Point Tracing Options

This section lists options that affect which program points appear in Celeriac Output.

--ppt-omit-pattern=*regexp*

Do not produce data trace output for classes/procedures/program points whose names match the given regular expression. This reduces the size of the data trace file and also may make the instrumented program run faster, since it need not output those variables.

For example the following flag:

```
--ppt-omit-pattern="MyInteger..ctor()"
```

would tell Celeriac to not output any program points (either entrances or exits) for the MyInteger? constructor.

This option works just like `--ppt-select-pattern` does, except that matching program points are excluded, not included.

Omit takes precedence over select.

--ppt-select-pattern=*regexp*

Only produce trace output for classes/procedures/program points whose names match the given regular expression. This option may be supplied multiple times, and may be used in conjunction with `--ppt-omit-pattern`.

When this switch is supplied, filtering occurs in the following way: for each program point, Celeriac checks the fully qualified class name, the method name, and the program point name against each regexp that was supplied. If any of these match, then the program point is included in the instrumentation.

Suppose that method bar is defined only in class `C`. To trace only `bar`, you could match the method name by providing the following flag:

```
--ppt-select-pattern="C.bar"
```

--sample-start=*sample-cnt*

When this option is supplied, Celeriac will record each program point until that program point has been executed sample-cnt times. Celeriac will then begin sampling. Sampling starts at 10% and decreases by a factor of *sample-cnt* each time another sample-cnt samples have been recorded. If *sample-cnt* is 0, then all calls will be recorded. The execution counts are unique to each app domain.

Variable and Method Tracing Options

This section lists options that affect which variables appear in Celeriac Output

--arrays-only=*true*|*false*

Determines whether variables that implement IList should be treated as arrays for purposes of instrumentaion. This will affect whether the variable's elements are inspected, e.g. `a[0]`, `...`, `a[n]`. `True` by default.

--linked-lists=*true*|*false*

This boolean option (default: true) causes user-defined linked lists to be output as sequences, much like C#'s List's and arrays are. A user-defined data structure is considered to be a linked list if it has one instance field that is of its own type. Supplying this option will cause Celeriac to fail (by running forever) if any linked list contains a cycle.

--nesting-depth=*n*

Depth to which to examine structure components (default 2). This parameter determines which variables Celeriac outputs at runtime. A field, parameter, or other instrumented variable (and its elements if it's a list) are considered depth 0, their identities will always be printed. Their children are depth 1, the children's children depth 2, etc. For instance, suppose that a program contained the following data structures and variables:

```
class A {
  int x;
  B b;
}
class B {
  int y;
  int z;
}

A myA;

class Link {
  int val;
  Link next;
}
Link myList;
```

If depth=0, only the identities (hashcodes) of `myA` and `myList` would be examined; those variables could be determined to be equal or not equal to other variables. If depth=1, then also `MyA.b`, `myList.next`, and the integers `myA.x` and `myList.val` would be examined. If depth=2, then also `MyA.b.y`, `MyA.b.y`, `myList.next.next`, and `myList.next.val` would be examined. Variables whose value is undefined are not examined. For instance, if `myA` is `null` on a particular execution of a program point, then `myA.b` is not accessed on that execution regardless of the depth parameter. That variable appears in the .dtrace file, but its value is marked as nonsensical.

--omit-dec-type=*regex*

Do not include variables whose dec-type matches the regular expression. Expressions will be omitted from each program point in which they appear.

--omit-parent-dec-type=*regex*

Do not include expressions whose parent's dec-type matches the regular expression. Expressions will be omitted from each program point in

which they appear.

--omit-var=*regex*

Do not include expressions whose name matches the regular expression. Expressions will be omitted from each program point in which they appear.

--purity-file=*pure-methods-file*

File pure-methods-file lists the pure methods (sometimes called observer methods) in a .NET program. Pure methods have no externally-visible side effects, such as setting variables or producing output. For example, most implementations of the `GetHashCode()`, `ToString()`, and `Equals()` methods are pure, as are automatically generated getters.

For each variable, Celeriac adds to the trace new "fields" that represent invoking each pure method on the variable. (Obviously this mechanism is only useful for methods that return a value: a pure method that returns no value does nothing!)

Here is an example:

```
class Point {
  private int x, y;
  public int RadiusSquared() {
    return x*x + y*y;
  }
}
```

If `RadiusSquared()` has been specified as pure, then for each point *p*, Celeriac will output the variables `p.x`, `p.y`, and `p.RadiusSquared()`. Use of pure methods can improve the Daikon output, since they represent information that the programmer considered important but that is not necessarily stored in a variable.

Invoking a pure method at any time in an application should not change the application's behavior. If a non-pure method is listed in a purity file, then application behavior can change. Celeriac does not verify the purity of methods listed in the purity file.

The purity file lists a set of methods, one per line. The format of each method is given as follows:

```
[AssemblyQualifiedName];[MethodName]
```

For example:

```
PureMethods.A, PureMethods, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null;PureMethod1
```

Pure methods must either take no parameters, or be static and take a single parameter of the same type as the containing class. Suppose for example the `String.IsNullOrEmpty(str)` method is marked as pure. Then, for all String objects the `IsNullOrEmpty` function will be called, with the String obejct as the parameter.

By convention, pure-methods-file has the suffix .pure.

--std-visibility

When this switch is provided, Celeriac will traverse exactly those fields that are visible from a given program point. For instance, only the public fields of class `pack1.B` will be included at a program point for class `pack2.A` whether or not `pack1.B` is instrumented. By default, Celeriac outputs all fields in instrumented classes (even those that would not be accessible in C# code at the given program point) and outputs no fields from uninstrumented classes (even those that are accessible).

Variable Metadata Options

This section describes command-line options for controlling metadata that is output for each variable.

--comparability=*filename*

When specified, Celeriac will write or use a comparability file to determine variable comparability during declaration printing. This provides more information in the trace files, which Daikon uses when determining invariants. Specifically, it will limit output invariants to those containing expressions that were actually used together. This helps reduce the number of false invariants Daikon guesses. The `--generate-comparability` flag can be used to generate a comparability file.

--enum-underlying-values

Instead of printing the string representation of an Enum's value, print the underlying integral-type value.

--is-enum-flags

When this flag is provided Celeriac will add the Daikon flag is_enum to the list of flags that are printed for all variables whose type is an enum. Use of this flag produces trace files that can only be processed with the enhanced version of Daikon.

--is-readonly-flags

When this flag is provided Celeriac will add the Daikon flag is_readonly to the list of Daikon flags that are printed for all readonly variables. Use of this flag produces trace files that can only be processed with the enhanced version of Daikon.

--is-property-flags

When this flag is provided Celeriac will add the Daikon flag is_property to the list of Daikon flags that are printed for Pure methods variables that are getter properties. Use of this flag requires still that the properties be specified in a purity file found with the --purity-file argument. Use of this flag produces trace files that can only be processed with the enhanced version of Daikon.

--simple-names

If provided, Celeriac will output unqualified names for declared types (dec-types) and type strings in the trace. *This flag is currently enabled for C# (see below).*

--source-language=*CLR*|*CSharp*|*VBasic*|*FSharp*

The name-style to use for declared types (dec-types) and type strings in the trace. CLR (and reflection-style names) is used by default. Note that Daikon has some hard-coded type names such as java.lang.String that have a special meaning. These special names will be output regardless of the source language; the C# format for Daikon converts the Daikon-name back into C#.

Miscellaneous Options

This section lists all other Celeriac options - that is, all options that do not control which program points and variables appear in Celeriac's output.

--dtrace-append

Instead of deleting the existing dtrace file, append the results of this run to the end of the existing file, if any.

--force-unix-newline

Always use the UNIX newline character. If not specified the OS default newline character will be used. This is necessary for passing the unit tests if they are run from a Windows machine, because otherwise the line endings in the generated dtrace files wouldn't match the expected test results.

--output-location=*filename*

Specifies the for the trace output (.dtrace) file. If this is not specified, then the default value of `daikon-output/PROGRAMNAME.dtrace` is used

--robust-mode

During program execution output nonsensical as the variable value instead of crashing if there's an exception during variable visiting. May limit the usefulness of resulting Daikon output.

--verbose=*true*|*false*

Determines whether to include debugging and progress information in the `STDOUT` output, false by default.

--vs-flags

For convenience when using the the Visual Studio add-in. Sets `comparability`, `is_readonly_flags`, `is_enum_flags`, `std_visibility`, and `is_property_flags`.

Program Analysis Commands and Options

This section lists all other the options for Celeriac program analysis. All these options switch Celeriac to offline mode. They all produce produce a file that can be used in the normal running of Celeriac.

--auto-detect-pure

When specified Celeriac will run in offline mode. During rewriting it will detect properties that are pure and construct a purity file containing all such properites. The resulting file will be named ProgramName_auto.pure.

--emit-nullary-info

When specified Celeriac will write to `STDOUT` the qualified types and method names for nullary (0-argument) methods reachable from the assembly. This can be useful for builing a purity file.

--purity-prefix-blacklist=*file*

Must be used with the emit-nullary-info option. Celeriac loads the file at file and processes each line in the file, treating it as a prefix to ignore when computing the list of nullary methods.

--generate-comparability

When specified Celeriac, will run in offline mode and output a file containing comparability information, which can then be used with the comparability flag.

Enter a comment: Hint: You can use Wiki Syntax.

Submit

ContributingToCeleriac
Contributing to Celeriac
Phase-Implementation, Developers, Featured

Updated Today (35 minutes ago) by melonhead901

Introduction

Celeriac is developed and maintained by the Programming Languages and Software Engineering group at the University of Washington. We welcome feedback and contributions from developers such as, these can be submitted to project owners Kellen Donohue (kellend@cs.washington.edu) or Todd Schiller (tws@cs.washington.edu):

- Enhancement requests and bug reports
- Feedback on documentation
- Tips and best practices to help new users
- Example use cases
- Feedback about supporting other languages targeting the CLR (e.g., IronPython)
- Patches and features (if you'd like suggestions on what to work on, please contact us).

This document describes guidelines for contributing code to Celeriac.

Building and Running

The project can be built from Microsoft Visual Studio 2010 or 2012. Open the Celeriac.sln file to get started. The project can then be built as normal. Execute the StartUp project by pressing F5. This will start running Celeriac on a simple HelloWorld program.

Code Guidelines

In general, follow the MSDN C# guidelines.

Issue Tracking and Version Control

The Celeriac project uses Mercurial for version control; we encourage new contributors to clone the project and submit a pull request. You may also request to be granted commit access.

All tasks should be tracked using the issue tracker located at http://code.google.com/p/daikon-dot-net-front-end/issues/. TODOs written in source code should include the associated issue number, preferably in the form "`TODO(#XX): explanation of what is wrong / needs to be done`", where XX is the issue tracker number. This single format makes it easy to search for the relevant code for a particular bug number.

When possible, commit messages should include the relevant issue #, similarly, when an issue is marked as fixed the comment should include the corresponding revision #.

Style

The project uses the style described in http://www.sourceformat.com/pdf/cs-coding-standard-bellware.pdf with the following exceptions:

- Spaces are used instead of tabs; an indent is 2 spaces, and wrapped lines require are doubly indented. Use Visual Studio formatting (Ctrl+E, Ctrl+D) to enforce consistent formatting.
- 100 character column limit.

Code Analysis

Code Analysis should give no warnings with the Microsoft Basic and Extended Correctness and Design Guidelines. It is OK to suppress a warning, however a reason should be included.

Test Suite

Before committing, ensure that you run and pass the test suite

Enter a comment: Hint: You can use Wiki Syntax.

Submit

daikon-dot-net-front-end

Celeriac .NET Front-End for Daikon

CSharpEnhancedDaikon

A description of a fork of Daikon enhanced to work with .NET

Users

Updated Jun 11 (2 days ago) by Todd.Schiller

Daikon was originally developed to work with Java. The University of Washington Programming Languages and Software Engineering group wrote a fork of Daikon which has enhancements the following targeted at generating C# Code Contracts and better capturing the semantics of .NET programs:

- C# Code Contract invariant output format
- Filtering of frame conditions for readonly / immutable expressions
- Interface Contract inference (i.e., support for variables with multiple parent program points)
- Programmatic access to post-processed invariants

The enhanced version of Daikon can be obtained here: https://code.google.com/p/daikon-csharp-changes/. Use of this fork is completely optional for most Celeriac users. However, some command-line flags do require this fork.

DaikonBugsAffectingInvariants

List of Daikon bugs affecting invariants produced from Celeriac traces

Phase-Deploy, Users

Updated Jun 4, 2013 by melonhead901

The following bugs affect Daikon output over sets:

- http://code.google.com/p/daikon/issues/detail?id=7
- http://code.google.com/p/daikon/issues/detail?id=8

Until these bugs are resolved sets will be treated just as normal lists.

daikon-dot-net-front-end

Celeriac .NET Front-End for Daikon

DebuggingStrategies

How to debug errors you encounter in Celeriac

Phase-Support, Phase-Implementation, Developers, Users

Updated Today (18 minutes ago) by melonhead901

Debugging Strategies

It's almost universally useful to isolate the bug to the smallest possible case. Try to determine which classes or methods in the subject program are at issue, and create a program containing only them. This will eliminate other potential bugs, improve iteration speed, and allow you to create a test case to verify the correctness of the fix.

1. First make sure the bug reporter has followed the steps found on the Reporting A Bug page, referring them to that page if necessary.
2. Second, determine if the bug occurs during rewriting or program execution. You can tell this by adding `--save-program` as a command line argument. If the program cannot be successfully rewritten the bug is a rewriting bug, otherwise it's a bug that occurs during execution.
   - Rewriting bugs will give you an exception trace, see if any information in the exception is useful for debugging. It's helpful to look at the `InnerException` property from within Visual Studio's debugger. Bugs of this type probably will be fixed in the `ILRewriter.cs` file.
   - For bugs that occur during execution try first verifying the rewritten program. This can be accomplished from the Visual Studio Command Line tools window. Type `peverify [executable-name]` If there are any errors they may reveal what the problem is. Bugs of this type will probably be fixed in the `ILRewriter.cs` file.
3. If the program passes the IL verifier but produces invalid output determine what about the output is invalid. Running the .dtrace through Daikon will usually produce useful error messages. See which line is causing the problems, and which variable is being instrumented at that line. Set conditional breakpoints in the `VisitVariable` method in `VariableVisitor.cs` or `PrintVariable` method in `DeclarationPrinter.cs` to get you to break at the time in the code when the bug is occurring.

Common Issues

- Converting between CCI Metadata types, Assembly Qualified Names, and .NET types is a historically error prone area.
  - First, determine what the assembly qualified name of the type should be. This can be done by setting a breakpoint in the source program where a variable of the relevant type is in scope, suppose it's name val.
  - Run the source program outside of Celeriac, and when you get to the breakpoint determine `val.GetType().AssemblyQualifiedName`, remember this.
  - Then set a breakpoint in the `ILRewriter` code that calls `TypeManager.ConvertCCITypeToAssemblyQualifiedName()`. There are many callsites, each corresponding to a different variable case. Step through and see if the proper `AssemblyQualifiedName` is being produced.
  - If the proper `AssemblyQualifiedName` is being produced then you must check that it is resolved properly when the rewritten program is running. Set a conditional breakpoint in the `VisitVariable` method with the name of the variable used from the first step. Watch as the `TypeManager.ConvertAssemblyQualifiedNameToType` proceeds, which should point you in the direction of fixing the resolution error.
- A `StackOverflowException` during program execution breaking in Visual Studio to `VariableVisitor.SetInvocationNonce()`, probably means you are re-instrumenting an already instrumented version of your program.
- Getting an error of "Could not load file or assembly 'file:///...' or one of its dependencies. Operation is not supported. (Exception from HRESULT: 0x80131515)" may occur if you get an executable downloaded from the internet or email, in these cases Windows attempts to block the file to prevent virus spread. To fix this problem right click on the executable in Windows Explorer, then click `Properties`. In the `General` tab near the bottom there is a button labeled `Unblock`, click this, then click `OK` to close the properties window.
- An exception with error message like `System.Runtime.InteropServices.COMException was unhandled HResult=-2147418113 Message=Catastrophic failure (Exception from HRESULT: 0x8000FFFF (E_UNEXPECTED))` may indicate that one of the CCI DLL files in the `%CELERIAC_HOME%` folder is not executable.

Submit

DemoWalkThroughs
Sample project walk-throughs
Phase-Deploy, Users

Updated Today (15 minutes ago) by melonhead901

Introduction

This page walks through the process of generating contracts for the sample projects bundled with Celeriac.

Bank Account

This walk-through provides an example of using Celeriac and Daikon to generate Code Contracts for a project. The Bank Account project consists of two parts: (1) an account interface with methods for deposits, transfers, and withdrawals, and (2) a account class with a minimum balance. Additionally, the solution includes an NUnit project to test the basic functionality of the account class.

Setup

Open the `BankAccount` solution in Visual Studio and perform a Debug build of the solution. The test project uses NUnit, so you may need to install NUnit from the Visual Studio package manager.

Since the project uses NUnit, we'll use Celeriac in offline mode. First we'll instrument `BankAccount.dll`, and then run the NUnit tests in `BankAccount.Tests.dll`. Since we're generating a trace from the tests, we'll be instrumenting the `BankAccount.dll` in the output directory of the testing project: `BankAccount.Tests\bin\Debug`.

Generating Purity Information

A purity file adds extra "fields", which are really method calls to a given type, to Celeriac's output. Pure methods are methods which have no external side effects, for example get properties, `ToString()`, and other observer methods. Any zero-parameter method, or method whose only parameter is the type of the receiver, can be added as a pure method. First, we'll have Celeriac list all of the nullary (methods not taking an argument) that could possible appear in the purity file.

```
CeleriacLauncher.exe --emit-nullary-info .\BankAccount.dll
```

This will create the file `BankAccount_auto.pure`. If you open the file, you will see that it lists methods for application types such as `BankAccount` and `IAccount`, as well as system types such as `System.Reflection.MethodBase` and `System.String`. You'll want to remove the lines that correspond to (1) methods at the modify state, and (2) methods you aren't interested in. Note that property getters have the name `get_Property`. For this example, we'll just keep our application-specific methods:

```
BankAccount.BankAccount, BankAccount, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null;get_Balance
BankAccount.BankAccount, BankAccount, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null;get_IsActive
BankAccount.BankAccount, BankAccount, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null;get_MinimumBalance
BankAccount.IAccount, BankAccount, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null;get_Balance
BankAccount.IAccount, BankAccount, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null;get_IsActive
```

and save the file as `BankAccount.pure`. We leave off the `IAccountContracts` methods since they're for the contract class where we'll place Code Contracts for the `IAccount` interface.

Notice that some of the methods, such as `get_Balance`, appear to be listed twice: once for `BankAccount` and once for `IAccount`. Each line in the purity lists a method to be called when the *declared type* of an expression corresponds to the type of the line. Therefore, if an expression is an `IAccount`, only the values methods included for `IAccount` will be output, even if the runtime type of the expression is `BankAccount` or another account type with additional methods (i.e., the minimum balance will not be recorded).

Instrumenting the Assembly

The next step is to instrument the assembly, passing in the purity information. The enhanced version of Daikon we maintain supports additional flags to ensure the proper display of invariants for .NET programs and the filter of uninteresting contracts. Include the `--is_property_flags` and `--is_readonly_flags` options to enable these features.

```
CeleriacLauncher.exe --is_property_flags --is_readonly_flags --purity-file=BankAccount.pure --save-program=BankAccount
```

This will create a metadata file `daikon-output\BankAccount.decls` and an instrumented assembly `BankAccount.dll.instr`. Next, replace the original DLL with the new, instrumented DLL:

```
mv BankAccount.dll BankAccount.dll.orig
mv BankAccount.dll.instr BankAccount.dll
```

Running the Test Suite

To run the test suite, open `BankAccount.Tests.dll` in the NUnit GUI and then run the tests as usual (or use the command line runner). This will create a file `daikon-output\BankAccount.test-domain-BankAccount.Tests.dll.dtrace`.

If you look at the generated trace, you'll see that it contains a section for each method entry and exit; with the first entry being a constructor call to `BankAccount`:

```
BankAccount.BankAccount..ctor(System.Decimal\_minimumBalance,\_System.Decimal\_initialBalance):::ENTER
this_invocation_nonce
2
minimumBalance
0
1
System.Decimal.MaxValue
79228162514264337593543950335
1
System.Decimal.MinusOne
-1
1
System.Decimal.MinValue
-79228162514264337593543950335
1
System.Decimal.One
1
1
System.Decimal.Zero
0
1
initialBalance
0
1
```

By default, Celeriac will output subexpression values up to a depth of two. Therefore, we see entries for `System.Decimal.MaxValue`, etc. since the parameters to the constructor have declared type `decimal`. We don't see entries for fields and pure methods of the account though, since this trace is for the entry to the constructor call.

Generating Code Contracts

To generate Code Contracts from the test suite trace, we'll use Daikon with the `csharpcontract` format specified (remember to change the command to reflect your Daikon JAR location):

```
java -cp C:\lib\daikon\daikon.jar daikon.Daikon --format csharpcontract .\daikon-output\BankAccount.decls .\daikon-output
\BankAccount.test-domain-BankAccount.Tests.dll.dtrace
```

This will output the inferred Code Contracts for each program point. For example, the inferred object invariants for the `BankAccount` class are:

```
BankAccount.BankAccount:::OBJECT

this.balance == this.Balance
this.minimumBalance == this.MinimumBalance
this.balance >= 0.0
this.minimumBalance.OneOf(0.0, 10.0, 50.0)
this.GetType() == typeof(BankAccount.BankAccount)
this.Balance >= 0.0
this.MinimumBalance.OneOf(0.0, 10.0, 50.0)
this.balance >= this.minimumBalance
this.balance >= this.MinimumBalance
this.minimumBalance <= this.Balance
this.Balance >= this.MinimumBalance
```

These contracts are pretty straight-forward: they capture they important fact that the account balance must be greater than the minimum balance, as well as relating the properties to their backing fields.

Insufficient Test Cases

However, the contract `this.MinimumBalance.OneOf(0.0, 10.0, 50.0)` is clearly invalid: the contract is true with respect to the tests we ran, but it doesn't capture the real behavior of the program. By default, Daikon will not generalize to inequalities until after it has seen more than three different values for an expression. Therefore, to generate better invariants, we need to either write test cases that set more than 3 different minimum balances, or override Daikon's default.

Contract Extension Methods

To produce more readable contracts, some contracts in the C# Code Contract format make use of utility extension methods. The `OneOf` method

in the above contracts is an example of one such method.

Interface Contracts

In the Code Contract system, methods that implement an interface or override a method cannot specify additional preconditions (`Contract.Requires` statements). However, the method can promise additional facts/postconditions. Therefore, contracts are generated for the entry points of `IAccount`'s methods, but not for the corresponding implementations in `BankAccount`.

The contracts inferred for interfaces such as `IAccount` must hold true for all implementing methods. For the provided test suite, the following contracts are inferred. As before, the contracts `destination.Balance == 250.0` and `amount == 100.0` suggest that tests should be created to cover more values.

```
BankAccount.IAccount.TransferFunds(BankAccount.IAccount destination, System.Decimal amount):::ENTER

destination != null
destination.GetType() == typeof(BankAccount.BankAccount)
destination.Balance == 250.0
destination.IsActive == true
amount == 100.0
```

Enter a comment: Hint: You can use Wiki Syntax.

Submit

GettingStarted
Setup and use instructions
Phase-Deploy, Featured, Users

Updated Today (21 minutes ago) by melonhead901

Getting Started

The Celeriac tool produces program traces for .NET programs for use with the Daikon dynamic invariant detector. Daikon uses machine learning to infer likely program invariants and properties. The types of properties inferred by Daikon include ".field > abs(y)"; "y = 2*x+3"; "array a is sorted"; "for all list objects lst, lst.next.prev = lst"; "for all treenode objects n, n.left.value < n.right.value"; "p != null ⇒ p.content in myArray"; and many more.

Celeriac currently supports the C#, F#, and Visual Basic .NET languages.

- Getting Started
- Downloading Celeriac
- Setup
- Generating a Trace with Celeriac
  - Online Tracing
  - Offline Tracing
  - Celeriac Output
- Inferring Likely Invariants with Daikon
- Detecting Contracts Involving Getters and Pure Method
  - Creating a Purity File
  - Tracing Pure Methods
- Excluding Information from Traces and Invariants
  - Good Things to Ignore
  - Ignoring Static Fields, Constants, and Enumerations
  - Method Call Sampling

Downloading Celeriac

To build Celeriac, check out the source from the Celeriac repository, open `Celeriac.sln`, and build from Visual Studio. The repository includes DLL's for all referenced dependencies.

Setup

The following setup ensures that Celeriac can properly instrument your run your program:

1. Set the `CELERIAC_HOME` environment variable to the directory containing `Celeriac.dll`.
2. Add `Celeriac.dll` to the Global Assembly Cache. In administrator mode, enter the following command:

```
gacutil /i %CELERIAC_HOME%\Celeriac.dll
```

As an alternative to adding Celeriac to the Global Assembly Cache (GAC), you can copy the Celeriac DLLs to the directory of the assembly you are instrumenting. *NOTE: currently, if the assembly you are instrumenting refers other assemblies that aren't in the GAC, you must copy the files the directory containing the assemblies (Issue #101).*

Generating a Trace with Celeriac

Celeriac supports two modes: online and offline mode. In online mode, Celeriac instruments an executable when the executable is run. In offline mode, Celeriac creates an instrumented copy of the assembly on disk; the instrumented assembly outputs a trace when executed. Offline mode is used when generating traces for class libraries and WPF programs.

While Celeriac is compatible with both debug and release builds, a PDB file is necessary to guarantee the correctness of the program, and to properly name the program points and variables. If a PDB file is not present, Celeriac will attempt to proceed (debugging information is required for

generating comparability information, however).

Online Tracing

To produce a trace for an executable, run the program using the Celeriac Launcher. For example, if you normally run the program using following command:

```
MyProgram.exe arg1 arg2 arg3
```

, instead use the following command:

```
CeleriacLauncher.exe celeriacArg1 celeriacArg2 MyProgram.exe arg1 arg3 arg3
```

A full listing of command line options is located on the CommandLineOptions wiki page.

Offline Tracing

To instrument an assembly to produce a trace when it is run, use Celeriac's `--save-program` option.

```
CeleriacLauncher.exe --save-program=InstrumentedClassLibrary.dll ClassLibrary.dll
```

A full listing of command line options is located on the CommandLineOptions wiki page.

This command will create an instrumented copy of the `ClassLibrary.dll` named `InstrumentedClassLibrary.dll`. If you are instrumented a class library, you will need to replace the original assembly with the instrumented assembly so that it is used by the executable. For example:

```
mv ClassLibrary.dll ClassLibrary.dll.orig
mv InstrumentedClassLibrary.dll ClassLibrary.dll
```

Optional: Verifying the Rewritten Assembly

To verify that Celeriac produced a valid .NET assembly, you can use Microsoft's `PEVerify` tool. On Windows 8, the tool is located at `C:\Program Files\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools\PEVerify.exe`:

```
PEVerify.exe <assembly>
```

Celeriac Output

When run, Celeriac produces two output files in the `daikon-output` directory, which is created wherever `CeleriacLauncher.exe` is executed:

- A metadata file, called a declaration file (with the extension .decls), describing the Daikon program points (method entrances and exits) and all variables in scope
- A data trace (with the extension .dtrace), or record of the program's execution

The metadata file is produced during instrumentation; the trace is produced when the program is run. In online mode, a single command performs both instrumentation and tracing. In offline mode, the metadata file is produced when the assembly is instrumented, and the trace is generated when the program is run. A new trace file is generated for each run unless the `--append-dtrace` option is supplied. A separate trace is generated for each Application Domain (to avoid synchronization across domains).

The declaration file and trace file formats are specified in the Daikon Developer's Manual: Declarations, Trace Records.

Inferring Likely Invariants with Daikon

We maintain a version of Daikon that supports outputting contracts as C# Code Contracts at https://code.google.com/p/daikon-csharp-changes/; the changes will be merged into the mainline Daikon repository after they have passed review.

To generate Code Contracts, but not insert them into the source code, use the `csharpcontract` format:

```
java -cp "C:\path\to\daikon.jar" daikon.Daikon --format csharpcontract AssemblyName.decls AssemblyName.dtrace
```

Note that the metadata file (the `.decls` file) must be provided before any trace files (`.dtrace` files).

See the Daikon User Manual for more information about generating invariants from a trace.

Detecting Contracts Involving Getters and Pure Method

In order to safely generate information about a method (or properties) for an expression, Celeriac must know that the method or property is *pure* -- that the method does not modify the state of the program in a visible way. Additionally, the method must not depend on external resources (e.g., `Date.Now`).

Properties are methods that behave like both methods and fields. Like fields, they get/set a value for the object. However, like methods, they can throw exceptions, modify other program state (e.g., logging), or return a different value each time they are called. Since auto-generated property getters are pure, Celeriac can detect auto-generated property getters. However, in general, you will have to indicate which methods and property getters are pure.

Creating a Purity File

A purity file is a list of assembly qualified types and method names that should be called when outputting information about an expression of the type.

To have Celeriac output the list of nullary methods and properties that are reachable from the assembly (up to a maximum nesting depth), enter the following command:

```
CeleriacLauncher.exe --emit-nullary-info --purity-blacklist=prefix-blacklist.txt <assembly>
```

where the optional blacklist file contains method name prefixes that likely indicate that a method is not pure (e.g., `Write`, `Add`, etc). If none is provided a sensible default list is used.

The output additionally includes unary static methods that act on the same type as the method's declaring type (for example, the `string.IsNullOrEmpty` method)

Tracing Pure Methods

To have Celeriac output trace information for the pure methods, use the `--purity-file` command line option:

```
CeleriacLauncher.exe --purity-file=purity-file.txt ... <assembly>
```

Excluding Information from Traces and Invariants

By default, Celeriac will instrument every method in the assembly and provide information about every field. In some cases, this poses two problems: (1) the instrumented program runs prohibitively slowly, and (2) the generated invariants are uninteresting.

Celeriac provides many command line options for controlling which information it tracks; descriptions of these options are listed on the CommandLineOptions wiki page. This section describes some common usage to get you started.

Good Things to Ignore

Here are some things we've learned to exclude from analysis. In the future, we hope to provide heuristics to handle these automatically:

- Auto-generated Parsers: the comparability analysis, trace generation, and Daikon runs very slowly on them due to their structure. If they're autogenerated, there's generally no need to generate invariants for them anyway.
- Static system variables: unless you know your program uses them in a meaningful way, your best bet is to ignore them using the `--omit-var=` option (see below). (Fixing Issue #99 should eliminate the need to do this.)
- Methods for common system types: unless you know your program uses a specific fact (e.g., the `Trim` method of `string`), don't include the method in the purity file, as it will be calculated for every use of the type.

Ignoring Static Fields, Constants, and Enumerations

Static fields (and enumeration values) include the fully qualified name of they type that declares the variable. Therefore, these variables can be excluded from a trace using the `--omit-var` option and a regular expression based on the qualified name of the type. For example, to exclude system fields and constants, the following regular expression can be supplied to Celeriac:

```
CeleriacLauncher.exe --omit-var="(System\..*?\..*)" ... <assembly>
```

Method Call Sampling

Sampling can reduce the size of the produced trace with little effect on the quality of the produced invariants. To use sampling, use the `--sample-start=_n_` option when instrumenting the assembly. The first *n* calls to each method will be traced. After the first *n* calls, sampling occurs with exponential back-off: first, every *n* th call will be traced, then every *10n* calls, *100n* calls etc. If `--sample-start=10` then the samples are 1, 2, 3, ..., 10, 20, 220, 2020, 20020, and so on. *n* must be greater than 0 for this option to have any effect.

Enter a comment: Hint: You can use Wiki Syntax.

Submit

ReportingABug

Suggestions for writing a bug report that will get the quickest response.

Developers, Users

Updated Jun 5, 2013 by melonhead901

Reporting a Bug

Properly reporting a bug may reveal a mistake or misunderstanding on your part, which is a much faster fix than waiting for a Celeriac developer to help you. A bug report should include:

- The program under investigation and the command line arguments used to run the program and Celeriac. The bug fixer needs the program in order to recreate the environment that caused the bug. An executable will suffice, however the source code is better if it's easy to build.
- If you are reporting a bug with the .dtrace file, describe **exactly** what is wrong with the .dtrace file (i.e., what is there versus what you expected to see).
- If possible, try to isolate the bug and submit submit only the relevant source code or executables required to reproduce the bug.

If you are fixing a bug for someone else, ask the person reporting the bug to make sure they have included this information.

Enter a comment: Hint: You can use Wiki Syntax.

Submit

RunningTests
How to build and run the Celeriac test suite.
Phase-QA, Developers

Updated Today (12 minutes ago) by melonhead901

Introduction

Each test in the test suite runs Celeriac on a source program and compares the generated output to a known correct result. Tests exist for specific features (e.g. the MultipleExits test) as well as for general programs (e.g. the GeoPoint test).

Setting up Daikon

Executing the tests requires installation of Daikon (http://groups.csail.mit.edu/pag/daikon/download/). Follow the installation instructions for Daikon, including setting the `DAIKONDIR` environment variable.

If using the Windows Java executable, add the following line to your `.bashrc` file translate the Java classpath to the Window's format:

```
export CLASSPATH=`cygpath -wp "$CLASSPATH"`
```

Setting up Celeriac and Test System

- Copy `Makefile.user.template` to `Makefile.user`
- In `Makefile.user`, change the value of `CELERIAC_EXE` to the path to `CeleriacLauncher.exe`. Note: you must use the Cygpath (`e.g., /cygdrive/c/...`) since colons on the path confuse Make.
- Ensure that the CCI Metadata DLLs are in the same folder as `CeleriacLauncher.exe`

Running the Tests

All tests are in the CeleriacTests directory. Change into this directory and execute `make`. This will run all the tests, including the GUI tests, and print the result of each to the screen.

Graphical User Interface Testing with Sikuli

Many programs have a Graphical User Interface (GUI), and exercising their features requires interacting with the GUI. To ensure Celeriac runs properly on these programs the test suite contains test programs with GUI's/ To perform automated testing on these kind of applications the Celeriac test framework uses the Sikuli tool.

Sikuli is a script based automation framework. Sikuli scripts are python programs, but can perform image based matching to click on certain button or other visual elements.

Getting Sikuli

Sikuli can be downloaded here; tutorials demonstrating how to use Suikuli are located here.

Adding tests of GUI applications, using Sikuli

The Celeriac testing framework usually runs by executing the CeleriacLauncher on a test source program. Sikuli seems to have errors executing from Cygwin trying to execute on a program launched from Cygwin, so the Sikuli tests launch the CeleriacLauncher and source program in the test script. For GUI applications tests should be run with Sikuli instead of the traditional setup (where a program is simply execture). To specify this add the line

```
SIKULI_RUN = 1
```

to the Makefile for that test. A Sikuli executable named {TestName}.skl must be placed in the test folder and added to the repository. As well, all Sikuli resources, usually in a folder ending in .sikuli, must be added to the repository as well.

Common Problems

Perl "Can't execute sort-dtrace.pl"

Ensure that permissions for the `sort-dtrace.pl` allow it to be executed by Perl. Setting the permissions from Windows may not work, so instead use the `chmod` command, e.g.,:

```
chmod 777 sort-dtrace.pl
```

The System Hangs with 100% CPU Utilization

For some users, running Celeriac from Cygwin causes the Windows Defender real-time protection service to run at 100% utilization. If this occurs, temporarily disable real-time protection while running the test suite.

Known Issues

- The MultipleThreads test fails the dtrace.diff and out.diff, this is expected because thread execution order is not deterministic.