

Pex, Extensions Writer Handbook

Peli de Halleux and Nikolai Tillmann,
Research in Software Engineering,
Microsoft Research.

Preliminary Draft
Copyright Microsoft Corporation.

January 27, 2010

This document provides coding guidelines for writing Pex extensions, such as custom test framework support or custom loggers. It also shows to architecture code so that it behaves nicely under the analysis of Pex.

1 Guidelines for Efficient Frameworks

To perform the runtime white box analysis, Pex *interprets* every MSIL instruction that gets executed along the program. This means that the more instructions gets executed during the test, the more work (and time) Pex will take to analyze the code. Unit test frameworks and Mock frameworks have started to use advanced features of the .NET platform (dynamic code generation, expression trees) to simplify test authoring. These new features comes with a major overhead in terms of executed MSIL instruction: what was a couple dozens of instructions becomes millions of executed instructions.

For example, when a mock framework compiles an `Expression` tree to MSIL code, it usually involves a mini-compiler. This introduce a (huge) constant overhead to the Pex runtime analysis, which in many case will simply *defeat* Pex. As a result, Pex is likely to reach exploration boundaries (such as maximum number of conditions used, etc...) much faster.

This section provides guidelines on building code so that it will be efficient under a Pex analysis. The following guidelines can be achieved without taking any dependency on the Pex assemblies.

1.1 Instrumented code

As mentioned above, Pex performs the runtime analysis by inserting callbacks to track every MSIL instruction. To do so, Pex uses the .NET profiling APIs to introduce those callbacks before every method is compiled by the Just-In-Time compiler.

Of course, this instrumentation is not turned on all the time, otherwise Pex which is also written in .NET, would be monitoring itself. Therefore, the interpreting state can be efficiently turned on and off when moving from the Pex runtime to the test code and so forth. To avoid this re-entrance problems, Pex also keeps a copy of the original method body to execute when it is not interpreting the execution:

```
{
    if (IsInterpreting)
        InstrumentedBody();
    else
        OriginalBody();
}
```

Pex provides many ways to specify which type or method should be instrumented (which is beyond this document). A convenient way to specify that a type should be instrumented is to use the `__InstrumentAttribute` attribute. The rewriting profiler uses this annotation to determine that a method needs to be instrumented. Since it does a simple name matching, this attribute can be redefined in any assembly. This is specially important for dynamically generated code (i.e. `Reflection.Emit`) that holds important program logic.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
sealed class __InstrumentedAttribute : Attribute { }

// usage example
[__Instrumented]
class SomeImportantCode {
    ...
}
```

1.2 Protected code

Another instrumentation mode is to protect the method body, i.e. unconditionally turn off interpreting while running the method body:

```
{
    using (DisableInterpreter())
        OriginalBody();
}
```

This mode is particularly useful to hide complex computation that do not matter to the test. For example, to compile `Expression trees`, `Reflection.Emit` methods, etc, ...

Classes can be marked as protected. By doing so, all method bodies from that class will be protected. The Pex rewriter profiler uses name-matching and will protect any class or struct marked with the `__ProtectAttribute` attribute. This means that you can define this attribute in your framework assembly (make it internal) and avoid taking a dependency on the Pex assemblies.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
```

```

sealed class __ProtectAttribute : Attribute {}

// usage example
[__Protect]
class MyCompiler {
    ...
}

```

Correctly *protecting* your framework code should take care of most of the problems.

1.3 Avoid DynamicMethod

`DynamicMethod` are not reported to the profiler and cannot be instrumented by Pex (and any other profiler for that matter). If important branching logic occurs in them, Pex will not be able to track the constraints in them.

1.4 Cache dynamically generated code

Pex executes the test in a loop and tries to cover every branch. If the mock framework does not cache the generated code, it will create more and more *new* branches on each run that Pex will try to cover. It is critical to cache the generated code so that Pex considers it as the same.

2 Writing Extensions

The extensions defined in this section require a dependency on the Pex and Extended Reflection assemblies.

2.1 A overview of the Pex Architecture

Pex uses a component + services architecture. Extensions can integrate into the system on each layer through additional services or registering to various events. The component model is divided into three layers:

execution services that live during the lifetime of the Pex execution,

exploration services that live during the lifetime of an exploration,

path services that live during the lifetime of a single test run.

2.2 Extensibility Mechanisms

Extensions can be loaded into the Pex engine by implementing custom attributes and using those attributes on the test assembly (There are other ways to register the extensions, see below). These extensions are referred as *packages* in the code. When loaded, the packages can add new components to the container or use services that are already present in the engine.

- execution layer: attribute inherits from `PexPackageAttributeBase`.
- exploration layer: attribute inherits from `PexExplorationAttributeBase`.
- path layer: attribute inherits from `PexPathAttributeBase`.

The following exploration package hooks to the generated test event, and dumps the generated test code to the console:

```
// write the code of the generated tests to the console
class ConsoleLoggerAttribute
    : PexExecutionPackageAttributeBase {
    protected override object BeforeExecution(IPexComponent host)
    {
        host.Log.GeneratedTestHandler +=
            (e) => Console.WriteLine(e.GeneratedTest.MethodCode);
        return null;
    }
}
```

To load this package, the attribute can be added at the assembly level to the test assembly or the package assembly (see below):

```
[assembly: ConsoleLoggerAttribute]
```

2.3 Packaging Extensions

Multiple packages can be bundled into a single assembly, referred as a *package assembly*. This assembly can be annotated with assembly-level package attributes which will be loaded by Pex .

This assembly should contain an implementation of the `PexPackageAssemblyAttribute` that specifies that the assembly contains package. This attribute can be used to specify to Pex to load the packages from an assembly.

The following attribute specifies that the enclosing assembly is a package assembly:

```
class BundledPackageAttribute
    : PexPackageAssemblyAttribute {
    public BundledPackageAttribute() {}
}
```

This attribute can be used on the test assembly to load all the packages that are defined on the package assembly. In this case, it will load the console logger package:

```
// in test assembly,
[assembly: BundledPackage]

// in bundled assembly,
[assembly: ConsoleLoggerAttribute]
```

2.4 Registering Extensions

The registry based registration is not supported anymore.

2.5 General Tips

Instrumented APIs are significantly slower than their original version, since all optimizations have been turned off and compilation time significantly increased. To mitigate this issue, Pex provides a number of fundamental data-structures that can be used in place of the BCL. This section also contains the description of additional useful API for writing extensions.

- Use the `SafeYYY` collections instead of the BCL collections.
- Do not return `T[]` as `IEnumerable<T>` as their runtime wrapper also gets instrumented. Use `Indexable.Array` instead.
- Avoid API that involve a lot of dynamic code such as `XmlSerializer`, `XsltCompiledTransform` or `Regex`.
- Use `SafeDebug.AssumeYYY` for method preconditions.
- Use `SafeDebug.AssertYYY` for code assertions.
- Use **Extended Reflection metadata** (`Metadata<T>`, `MetadataFromReflection`, `MetadataFromRuntimeHandles`) **instead of Reflection**.