

Automated Testing of Environment-Dependent Programs – A case study of modeling the File System for Pex

Soonho Kong
ROSAEC Center, Seoul National University

Nikolai Tillmann, Jonathan de Halleux
Microsoft Research, Redmond WA 98052, USA

Abstract

Programs that interact with the file system are a classical challenge for automated software testing. A common approach to handling this problem is to insert an abstraction layer between the application and the file system. However, even with a well-defined abstraction layer, the burden on the software developer or tester is still high: they have to understand the subtleties of the file system to craft a meaningful set of test cases. The file system is accessed through a complex API, which often causes developers to overlook obscure yet possible corner cases.

In this paper, we present a parameterized model of the file system that can be used in conjunction with Pex, an automated test generation tool, to test code that depends on the file system.

Keywords: Automated Testing, Unit Testing, Test Generation, Environment Modeling, Dynamic Symbolic Execution

1. Introduction

Programs that interact with the environment are a classical challenge to automated software testing. Testing code that interacts with the file system is a very common instance of this problem. By directly using the operating-system level file system APIs, the code becomes less testable, since it introduces an unconditional dependency on the state of the physical file system at the time tests are executed.

A classic symptomatic example is a test that passes the first time it is run, and then fails on subsequent runs. The first run generated files which broke subsequent runs.

To deal with these issues, programmers usually introduce a level of abstraction between the environment, e.g. the file system, and the code.

In this article, we study an abstraction of the file system for .NET programs that was defined in the *CodePlex Client* project [2].

However, even with a clear abstraction layer, the burden on the developer is still high. They have to understand the subtleties of the file system in order to craft a relevant and comprehensive set of test cases. The

file system is a complex system, which often causes programmers to overlook obscure yet possible corner cases. Moreover, expressing each scenario in code usually involves significant work to *mock* [3] the behavior of the file system. (In traditional unit testing, mocking a dependency involves the creation of a lightweight alternative implementation, which validates inputs and provides hardcoded outputs.)

To address these issues, we will show how Pex can be used to define a *parameterized model* [10, 11, 12] of the file system. Pex is an automated white box testing tool for .NET [1, 7]. Using the model, the developer can write test cases for *any* initial file system state rather than for a particular state. Then automated test generation tools can be employed to explore possible behaviors of the file system in combination with the code under test. As a result, the work performed by the developer is dramatically decreased.

We will illustrate the parameterized model by implementing a `CopyFiles` method that copies files from one directory to another. We will describe how the model can be implemented as a non-deterministic program, whose parameters can be explored by automated test generation tools. We will give an overview of our model implementation, and compare it to related work.

In the following, all code examples are expressed in C#. Visibility annotation might be omitted for brevity.

2. Example: Untestable `CopyFiles` method

Copying files from one folder to another is a representative example of a non-trivial interaction with the file system. In .NET, a developer can use the `Directory`, `File` and `Path` classes provided by the runtime to implement such functionality:

```
// simulating "copy sourcePath\* targetPath"
static void CopyFiles(string sourcePath,
    string targetPath) {
    var sources = Directory.GetFiles(sourcePath);
    foreach (var source in sources) {
        string target = Path.Combine(
            targetPath,
            Path.GetFileName(source));
        // copy and do not overwrite
        File.Copy(source, target, false);
    }
}
```

```
}
```

This implementation is problematic for testing because it relies on the file system API of the .NET base class library which simply calls the Win32 file system API. As consequence, the code depends directly on the state of the file system.

3. File System Abstraction Layer

In order to make the `CopyFiles` method testable, we need to create an abstraction layer of the file system. We use an already existing abstraction of the file system from the CodePlex Client project [2], where they published an interface, `IFileSystem`, that represents most of the operations that are commonly performed on a file system. The project also provides an implementation that accesses the physical file system, called `FileSystem`.

Using the `IFileSystem` interface, the `CopyFiles` method can be refactored as follows:

```
static void CopyFiles(IFileSystem fs,
    string sourcePath,
    string targetPath) {
    var sources = fs.GetFiles(sourcePath);
    foreach (var source in sources) {
        string target = fs.CombinePath(targetPath,
            fs.GetFileName(source));
        fs.CopyFile(source, target);
    }
}
```

4. Traditional Mock-based Testing

After this refactoring, the developer can provide specialized implementations of `IFileSystem` that will cause the program under test to exhibit different behaviors. For example, in traditional unit testing with so-called mock objects, the developer might write the following code to test a “happy” successful scenario with a hand-written mock file system holding a single file:

```
class NoSourceFileSystem : IFileSystem {
    string[] GetFiles(string path) {
        return new string[] { "file.txt" };
    }
    string CopyTarget;
    void CopyFile(string source, string target) {
        this.CopyTarget = target; // record value
    }
    ...
}

[TestMethod]
void SourcePathDoesNotExist() {
    var fs = new NoSourceFileSystem();

    CopyFiles(fs, "source", "target");

    Assert.AreEqual(
        fs.CopyTarget, "target\\file.txt");
}
```

This scenario covers only one out of many possible behaviors of the file system. Although Mock Frameworks such as Moq [5], Rhino Mocks [4], and NMock [6], provide convenient APIs to define particular behaviors of the mocks in a compact way, none of them actually solve the problem of covering all the relevant scenarios.

5. Parameterized Unit Testing with Pex

Pex is an automated white box testing tool for .NET. From a hand-written parameterized unit test, Pex generates test inputs that exercise many statements in the code, using a technique called dynamic symbolic execution, a combination of dynamic and static analysis [8]. Pex saves the results as a unit test suite, which the user can later execute and debug without Pex. The following example shows a parameterized unit test, designated as such with the `[PexMethod]` attribute. It adds an element to a dictionary and ensures that the element was properly stored in the dictionary.

```
[PexMethod] // hand-written
void AddItem(int key, object value) {
    var dic = new Dictionary();
    dic[key] = value;
    Assert.AreEqual(value, key[value]);
}
```

From this parameterized unit tests, Pex generates multiple closed unit tests with relevant values to cover the statements of the `Dictionary` implementation. Each unit test is tagged with the `[TestMethod]` attribute.

```
[TestMethod] // automatically generated
void AddItem01() {
    this.AddItem(0, null);
}
[TestMethod]
void AddItem02() {
    this.AddItem(1, null);
}
...
```

6. Choices

In the example above, the inputs of the test were supplies by parameters of the `AddItem` method. Pex also provides a class, `PexChoose`, to provide new inputs on demand along the execution of the test [11]. By performing multiple queries, `PexChoose` can be used to build non-deterministic models. For example, the `ReadAllText` method of `IFileSystem`, which reads the entire content of a file, could be trivially implemented as follows:

```
string ReadAllText(string path) {
    // ask Pex to choose a string value
    return PexChoose.ChooseResult<string>();
}
```

When `ReadAllFiles` is called, Pex will create a string value, and Pex static and dynamic analysis will track it as if it was just another input of the test. As a result, Pex will generate different string values when necessary to cover more statements of the code under test.

6. From Choices to Parameterized Models

In the context of unit testing, developers often make the (implicit) assumption that the file system is not modified concurrently by other processes. Under that assumption, the simple implementation of `ReadAllText` above is too simplistic since Pex might decide to return a different value on each call to `ReadAllText`, even for the same file name. Unless the file is explicitly modified by the test, the developer would expect `ReadAllText` to return the same value. These kinds of relaxed-robustness constraints are not captured by the simplistic implementation, which is only restricted by the type system so far.

One may argue that such implicit assumptions are wrong, and that thorough testing should include all possible concurrent scenarios. And doing so is easily possible with a permissive model and Pex. However, in practice, testing is foremost focused on realistic common scenarios. Even without taking into account concurrent file system modifications, there are many surprising yet realistic scenarios, as we will see in the following.

In order to model a stateful file system, we enhanced the implementation of `PFileSystem` by adding state that remembers the initial choices and all subsequent modifications. In that sense, we wrote a *Parameterized Model*[10]: *parameterized* in the sense that it uses external inputs to provide the initial state, *model* in the sense that it has a mutable state and different possible transitions based on the current state.

Based on how the code under test uses the interface, Pex will generate different initial file system states.

7. Writing Tests with Parameterized Models

Using the parameterized model, a developer can write test cases that should hold for *any* state of a file system.

```
[PexMethod]
void CopyFilesModel() {
    var fs = new PFileSystem();
    string sourcePath = @"\src";
    string targetPath = @"\tar";

    // CopyFiles
    CopyFiles(fs, sourcePath, targetPath);

    // Assert
    var sources = fs.GetFiles(sourcePath);
```

```
foreach (var source in sources) {
    string target = fs.CombinePath(
        targetPath,
        fs.GetFileName(source));
    // files in sourcePath
    // should exist in targetPath
    Assert.IsTrue(fs.FileExists(target));

    // each copy has the same contents
    // as original.
    byte[] scontent = fs.ReadAllBytes(source);
    byte[] tcontent = fs.ReadAllBytes(target);
    Assert.AreEqual(
        scontent,
        tcontent,
        (b1, b2) => b1 == b2);
}
}
```

It is important to notice that, contrary to mock-based testing, this test does not require any initialization of the file system mocks. The newly created `PFileSystem` instance represents any possible initial state of the file system, and thus our `CopyFiles` implementation should be able to deal with it. The developer does not need to be an expert in file systems and can rely on the model to explore the many different behaviors.

Moreover the assertions that were written in the test are more general, as they should hold for any initial state of the file system.

After exploring the test with Pex, we get 10 passing tests and 8 failing tests as depicted in the table below. The file names which Pex comes up with such as “; \0;” or “\\$ \;” are valid file names even if it looks strange at first.

	files in source	files in target	Summary/Exception
❌ 1			DirectoryNotFoundException
❌ 2			DirectoryNotFoundException
❌ 3			DirectoryNotFoundException
✅ 4	{}	{}	
❌ 5			DirectoryNotFoundException
❌ 6			UnauthorizedAccessException
✅ 7	{\src\===}	{\tar\===}	
✅ 8	{\src\===}	{\tar\===}	
✅ 9	{\src\===}	{\tar\===}	
✅ 10	{\src\===}	{\tar\===}	
✅ 11	{\src\===}	{\tar\===}	
✅ 12	{\src\===}	{\tar\===, \tar\0}	
✅ 13	{\src\===}	{\tar\===, \tar\0000\000}	
✅ 14	{\src\===}	{\tar\===, \tar\0}	
❌ 15			IOException
✅ 16	{\src\===}	{\tar\===, \tar\;0;0}	
❌ 17			DirectoryNotFoundException
✅ 18	{\src\===}	{\tar\===, \tar\\$\$, \tar\=}	
❌ 19			UnauthorizedAccessException

Failed tests are categorized into three cases which are not properly handled by the `CopyFiles` method.

- 1, 2, 3, 5, 17: source path “\src” or target path “\tar” does not exist in file system,

- 6, 19: under the target path, there is a directory whose name is the same as the file we attempt to copy.
- 15: The file to be copied already exists in target path.

Note that with mock-based testing, the developer would have had to implemented a specialized implementation of `IFileSystem` and encode the scenario for each of those cases, knowing exactly what the corner cases of the file system are.

Based on this feedback, we can modify `CopyFiles` to handle such exceptional cases.

```
static void CopyFiles(IFileSystem fs,
    string sourcePath,
    string targetPath) {
    if (fs.DirectoryExists(sourcePath) throw ...;
    if (fs.DirectoryExists(targetPath) throw ...;
    var sources = fs.GetFiles(sourcePath);
    foreach (var source in sources) {
        string target = fs.CombinePath(
            targetPath,
            fs.GetFileName(source));
        if (fs.FileExists(target)) throw ...;
        if (fs.DirectoryExists(target)) throw ...;
        fs.CopyFile(source, target);
    }
}
```

Exploring the same test `CopyFilesModel`, Pex generates 16 passing tests for the corrected implementation, shown in the following table.

	files in source	files in target	Summary/Exception
1			DirectoryNotFoundException
2			DirectoryNotFoundException
3			DirectoryNotFoundException
4	{}	{}	
5			UnauthorizedAccessException
6	{\src\=}	{\tar\=}	
7	{\src\=}	{\tar\=}	
8			IOException
9	{\src\=}	{\tar\=}	
10	{\src\=}	{\tar\=}	
11	{\src\=}	{\tar\=}	
12	{\src\=}	{\tar\=, \tar\0}	
13	{\src\=}	{\tar\=, \tar\0;}	
14	{\src\=}	{\tar\=, \tar\ \$ }	
15			UnauthorizedAccessException
16			UnauthorizedAccessException

8. Model Implementation

Our model is available in source code as part of the Pex distribution [1]. The file system interface consists of 34 methods, which allow the inspection and modification of directories, files, their contents, and path names. Our model implementation consists of 2529 lines of C# code,

spread over five classes. We wrote and let Pex explore 33 parameterized unit tests which describe high-level properties of the file system, e.g. when `Open(fileName)` succeeds, then `FileExists(fileName)` must succeed as well.

9. Related Work

The idea of writing models to simulate the environment is not new. Dynamic symbolic execution can be seen as an instance of model checking, where this problem has been studied for more than a decade. A recent practical approach tries to address this problem for compositional model checking of Java programs by semi-automatically *generating* environments [9], combining high-level user supplied environment models with automatically inferred environment properties. The automatic inference performs a static analysis of the code that implements the environment behavior. This approach does not apply for the file system, as the .NET file system implementation is merely a shallow wrapper of Win32 APIs. In the case study presented in this paper, we wrote the environment model by hand. The size of our model seems reasonable (about 2500 lines of C# code).

10. Conclusion and Future Work

In this article, we have studied the implementation of a parameterized model for the file system on top of an automated white box testing tool, Pex. Parameterized models are reusable across all tests and solve the shortcomings of today’s mock infrastructure; instead of relying on the developer to define scenarios, the model can be explored by Pex to generate all meaningful scenarios.

In future work, we will model other environment facing APIs such as database accesses, possibly leveraging higher level models, and automatic environment inference. Also, we will investigate how initial model states can be translated into corresponding states of the physical environment, which will allow the validation of the behavior defined by the model, and it will allow the transformation of unit tests into integration tests.

References

- [1] Microsoft Research Redmond, “Pex”, <http://research.microsoft.com/pex>
- [2] C. C. Team, “Codeplex Client 2007”, <http://www.codeplex.com/CodePlexClient>
- [3] Wikipedia, “Mock Object”, http://en.wikipedia.org/wiki/Mock_Object
- [4] Ayende Rahiem, “Rhino Mocks”, <http://ayende.com/projects/rhino-mocks.aspx>.
- [5] “Moq”, <http://code.google.com/p/moq/>
- [6] “NMock”, <http://nmock.org>

- [7] Nikolai Tillmann, Jonathan de Halleux, “Pex – White Box Test Generation for .NET”, Proc. of TAP 2008, LNCS, vol. 4966, pages 134-153, April 2008
- [8] Patrice Godefroid, Nils Klarlund, Koushik Sen, “DART: directed automated random testing”, SIGPLAN Notices, Vol. 40-6, pages 213-223, 2005
- [9] Oksana Tkachuk, Matthew B. Dwyer, “Automated environment generation for software model checking”, In Proceedings of the 18th International Conference on Automated Software Engineering, pages 116-129, 2003
- [10] Patrice Godefroid, Model Checking for programming languages using VeriSoft, Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174-186, 1997.
- [11] Nikolai Tillmann, Wolfram Schulte, Mock-object generation with behavior, Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pages 365-368, September 2006.
- [12] Christopher Colby, Patrice Godefroid, Lalita Jategaonkar Jagadeesan, Automatically Closing Open Reactive Programs, Proceedings on Programming Language Design and Implementation (PLDI), pages 345-357, 1998.