Advanced Concepts: Parameterized Unit Testing with Microsoft Pex

Version 0.93 – August 3, 2010

Abstract

A parameterized unit test is a method that takes parameters, calls the code under test, and states assertions. Given a parameterized unit test written in any .NET Framework language, Microsoft Pex automatically produces a small test suite with high code and assertion coverage. By performing a systematic program analysis—similar to path-bounded model-checking—Pex also suggests a bug fix when a generated test fails.

This document provides extensive information for developers already familiar with using Microsoft Pex as part of their testing methodology. Specifically:

- How various techniques work in whitebox software testing.
- How Pex crafts inputs to parameterized unit tests.

This guide is Technical Level 500. To take advantage of this content, you should be an experienced .NET developer and have experience applying the concepts and capabilities discussed in these documents:

“Exploring Code with Microsoft Pex”
“Parameterized Unit Testing with Microsoft Pex”
“Parameterized Test Patterns for Microsoft Pex”

Note:

- Most resources discussed in this paper are provided with the Pex software package. For a complete list of documents and references discussed, see “Resources and References” at the end of this document.
- For up-to-date documentation, Moles and Pex news, and online community, see http://research.microsoft.com/pex
- If you have a question, post it on the Pex and Moles forums.
Contents

Understanding Whitebox Software Testing .......................................................... 3
  Analysis Techniques ......................................................................................... 3
  Specification Techniques ................................................................................ 4
  The Testing Problem ........................................................................................ 4
  Whitebox Test Input Generation ...................................................................... 5
  Symbolic Execution .......................................................................................... 5
  Dynamic Symbolic Execution .......................................................................... 6
  Example: How Pex Handles Program 1 ............................................................ 7
Understanding Pex ............................................................................................... 8
  Why Dynamic Symbolic Execution ................................................................. 8
  Exercise 1: The Path Condition ...................................................................... 9
  Monitoring by Instrumentation ...................................................................... 10
  Symbolic State Representation ...................................................................... 11
  Exercise 2: Heap Constraints ...................................................................... 12
  Search Strategy ............................................................................................... 13
  Exercise 3: Search Frontiers .......................................................................... 13
  Constraint Solving .......................................................................................... 14
  Exercise 4: Constraint Solving ...................................................................... 14
  Implicit Branches ............................................................................................ 14
  Exercise 5: Implicit Branches ...................................................................... 15
Understanding Assumptions, Assertions, and Test-Case Failures ...................... 15
  Assumptions and Assertions ......................................................................... 15
  When Does a Test Case Fail? ......................................................................... 16
  When Does Pex Emit a Test Case? ................................................................. 17
  Exercise 6: Test Case Emission Filters ............................................................ 18
  When Does Pex Stop? ..................................................................................... 18
  How Does Pex Suggest Fixes for Errors? ......................................................... 21
Understanding Complex Objects, Explorables, Invariants, and Limitations ...... 22
  Complex Objects ............................................................................................ 22
  Explorables ..................................................................................................... 22
  Invariants ........................................................................................................ 23
  Limitations ...................................................................................................... 24
Resources and References .................................................................................... 25

Disclaimer. This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, IntelliSense, SharePoint, Visual Studio, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.
Understanding Whitebox Software Testing

This article supplements information about parameterized unit tests with Microsoft Pex. This article assumes that you are experienced using Microsoft Pex and understand the concepts related to parameterized unit tests.

Analysis Techniques

In general, all program analysis techniques fall between the following two ends of the spectrum:

- **Static analysis techniques** verify that a property holds on all execution paths.
  Because the goal is program verification, these techniques are usually overly conservative and flag possible violations as errors, leading to “false positives.”

- **Dynamic analysis techniques** verify that a property holds on some execution paths.
  Testing is a dynamic analysis technique that aims at detecting bugs, but usually cannot prove the absence of errors. Thus, these techniques often fail to detect all errors.

It is often not possible to detect bugs precisely when applying only static analysis or employing a testing technique that is not aware of the structure of the code. Consider the following example in Program 1.

**Program 1. Difficult-to-Analyze Code**

```c
int Complicated(int x, int y) {
    if (x == Obfuscate(y))
        throw new RareException();
    return 0;
}

int Obfuscate(int y) {
    return (100 + y) * 567%2347;
}
```

- Static analysis techniques tend to be overly conservative, so the non-linear integer arithmetic present in `Obfuscate` causes most static analysis techniques to give up and issue a warning about a potential error in `Complicated`.

- Random testing techniques have very little chance of finding a pair of `x` and `y` values that triggers the exception.

Microsoft Pex implements an analysis technique that falls between these two extremes: dynamic symbolic execution, which is a whitebox test generation technique. Similar to static analysis techniques, Pex proves that a property holds for most feasible paths—those within specified exploration bounds. Similar to dynamic analysis techniques, Pex reports only real errors and no false positives.
Specification Techniques

All program analysis techniques try to validate and/or falsify certain specified properties of a given program. There are different ways to specify program properties:

- **API Contracts**—Spec#, Eiffel, and so on—specify the behavior of individual API actions from the implementation’s point of view.
  
  Their goal is to guarantee robustness, in the sense that operations do not crash and data invariants are preserved. A common problem of API contracts is their narrow view on individual API actions, which makes it hard to describe system-wide protocols. Complex extensions like model-fields are necessary to raise the abstraction level.

- **Unit Tests**—JUnit, NUnit, and so on—are exemplary usage scenarios from the point of view of a client of the API.
  
  Their goal is to guarantee functional correctness, in the sense that the interplay of several operations behaves as intended. A common problem of unit tests is that they are detached from the details of the implementation. Measuring the code coverage achieved by the unit tests gives only a rough idea of how adequate the unit tests are.

Pex enables parameterized unit testing, which unites both techniques. Supported by a test-input generation tool such as Pex, this methodology combines the client and the implementation points of view. The functional correctness properties (parameterized unit tests) are checked on most corner cases of the implementation (test input generation).

The Testing Problem

Starting from parameterized unit tests as specification, we can state the testing problem as follows:

Given a sequential program $P$ with statements $S$, compute a set of program inputs $I$ such that for all reachable statements $s$ in $S$ there exists an input $i$ in $I$ such that $P(i)$ executes $s$.

Remarks

- By **sequential** we mean that the program is single-threaded.

- We consider failing an assertion or violating an implicit contract of the execution engine—for example, raising `NullReferenceException` when null is dereferenced—as special statements.

- Because reachability is not decidable in general, we aim for a good approximation in practice: high coverage of the statements of the program. Instead of statement coverage, other coverage metrics such as arc coverage can be used.
Whitebox Test Input Generation

Whitebox testing leverages information about how a software system is implemented in order to validate or falsify certain properties. Whitebox testing can involve the analysis of data flow, control flow, exception handling, or other details of the implementation. In order to obtain the necessary information, whitebox testing requires access to the software’s source code or the compiled binary.

Pex is a whitebox analysis tool that analyzes the compiled instructions sequence of a .NET program—which is, the Microsoft Intermediate Language (MSIL) instructions.

The opposite of whitebox testing is blackbox testing, which usually amounts to using randomly generated test data.

Symbolic Execution

Pex implements a whitebox test input generation technique that is based on the concept of symbolic execution. The goal for Pex is to automatically and systematically produce the minimal set of actual parameters needed to execute a finite number of finite paths.

In general, symbolic execution works as follows:

- For each formal parameter, a symbolic variable is introduced.
- When a program variable is updated to a new value during program execution, then this new value is often expressed as an expression over symbolic variables.
- When a statement in the program has more than one possible successor, execution is forked into two paths.
- For each code path explored by symbolic execution, a path condition is built over symbolic variables.

For example, the Add-method of the ArrayList implementation shown in Program 2 contains an if-statement whose condition is this._items.Length==this._size, where:

- _items denotes the array holding the array list’s elements.
- _size denotes the number of elements currently contained in the array list.

Program 2. ArrayList Implementation in .NET

```csharp
public class ArrayList
{
    private Object[] _items = null;
    private int _size, _version;
    ...
    public virtual int Add(Object value) {
        if (_size == _items.Length) EnsureCapacity(_size + 1);
        _items[_size] = value;
        _version++;
        return _size++;
    }
}
```
The symbolic execution conjoins this condition to the path condition for the _then_-path and the negated condition to the path condition of the _else_-path. In this manner, all constraints are collected that are needed to deduce what inputs cause a code path to be taken.

A constraint solver or automatic theorem prover is usually used to decide the feasibility of individual execution paths and to obtain concrete test inputs as representatives of individual execution paths.

Analysis of all paths cannot always be achieved in practice. When loops and recursion are present, an unbounded number of code paths might exist. In this case, loops and recursion are usually analyzed only up to a specified number of unfoldings. Even if the number of paths is finite, solving the resulting constraint systems is sometimes computationally infeasible, depending on the employed constraint solver or automatic theorem prover.

Symbolic execution in its original form is a static program analysis, because it does not actually execute the program but merely analyzes possible execution paths.

**Dynamic Symbolic Execution**

Applying symbolic execution as described in the previous section to a real-world program is problematic, because such a program’s interaction with a stateful environment cannot be “forked”.

Pex explores the behaviors of a parameterized unit test using a technique called dynamic symbolic execution. This test-generation technique consists of:

- Executing the program, starting with very simple inputs,
- While simultaneously performing a single-path symbolic execution to collect symbolic constraints on the inputs obtained from predicates in branch statements along the execution, and
- Then using a constraint solver to infer variants of the previous inputs in order to steer future program executions along alternative program paths.

In this way, all program paths will be exercised eventually. Operations that are implemented by the external environment are not tracked symbolically. Instead, the actually observed input/output values become part of the constraints.

As a result, dynamic symbolic execution extends static symbolic execution with additional information that is collected at runtime, which makes the analysis more precise. By continuation, all analyzed execution paths are feasible, which avoids the problem of spurious error reports common to overly conservative static analysis techniques.

Although additional information is collected on the level of individual execution traces that characterize individual execution paths, knowing the structure of the program still enables the analysis of many execution paths at once.

Algorithm 1 shows the general dynamic symbolic execution algorithm. The choice of new inputs in Step 1 decides in which order the different execution paths of the program are visited. When deciding on the next program inputs, Pex uses several
heuristics that take into account the structure of the program and the already covered branches.

The goal of the Pex strategy is to achieve high statement coverage quickly. As an effect, you only have to set a time limit or another rough analysis bound. Most other tools explore the execution paths in a fixed search order, and they require that you define detailed bounds on the size and structure of the input data.

Algorithm 1. Dynamic Symbolic Execution

Step 0  Set $J := \emptyset$  
Intuitively, $J$ is the set of already analyzed program inputs.

Step 1  Choose program input $i \not\in J$  
Stop if no such $i$ can be found.

Step 2  Output $i$

Step 3  Execute $P(i)$; record path condition $C$  
In particular, $C(i)$ holds.

Step 4  Set $J := J \cup C$  
Viewing $C$ as the set \{ $i$ | $C(i)$ \}

Step 5  Goto Step 1

Example: How Pex Handles Program 1

In this example, we explain how Pex handles Program 1, starting from the method Complicated.

1. In order to run the code for the first time, Pex needs to supply some argument values to Complicated—for example, $x = 572$ and $y = 152$ (arbitrary values).

2. Pex monitors the execution of the code, following the execution into and out of method calls.
   
   With randomly chosen values, it is very unlikely that we will trigger the rare exception.

3. Pex remembers that we did not cover the throw statement. Pex also remembers all conditions that were evaluated; here, it remembers that $x \neq (100 + y) \cdot 567 \mod 2347$.

4. Knowing that we have not yet covered the throw statement, Pex looks at the program to find the branch that guards that statement.
   
   Last time, we had $x \neq (100 + y) \cdot 567 \mod 2347$. So, in order to reach the other branch of the if-statement, Pex builds the negated condition, $x = (100 + y) \cdot 567 \mod 2347$, and hands it to a constraint solver.
   
   In this case, it is quite simple to solve the constraint system, because you just have to supply any value for $y$ to compute $x$.

5. Pex runs the code again for with the new inputs—say, $x = (100 + 152) \cdot 567 \mod 2347$ and $y = 152$.
   
   This time, the throw statement will be covered.

6. Because all statements have been covered now, Pex will stop.
Understanding Pex

This section explains various aspects of Microsoft Pex, the tool that is used to craft inputs to parameterized unit tests.

Why Dynamic Symbolic Execution

Symbolic execution was originally proposed as a static program analysis technique, which is an analysis that only considers the source code of the analyzed program. This approach works well as long as all decisions can be made on basis of the source code alone.

It becomes problematic when:

- The program contains constructs that make reasoning hard—for example, accessing memory through arbitrary pointers.
- When parts of the program are actually unknown—for example, when the program communicates with the environment, for which no source code is available and whose behavior has not been specified.

Many .NET programs use unsafe features such as arbitrary memory accesses through pointers for performance reasons. And most .NET programs interact with other unmanaged—that is, non .NET—components for legacy reasons.

Although static symbolic execution algorithms do not have or use any information about the environment into which the program is embedded, dynamic symbolic execution does leverage dynamic information that it observes during concrete program executions—information about actually taken execution paths, and the data that is passed around between the analyzed program and the environment.

Knowing the actually taken execution paths allows Pex to prune the search space. When the program communicates with the environment, Pex builds a model of the environment from the actual data that the environment receives and returns. This model is an under-approximation of the environment, because Pex does not know the conditions under which the environment produces its output.

The resulting constraint systems that Pex builds might no longer accurately characterize the program’s behavior, and as a consequence Pex prunes such paths. Thus, Pex always maintains an under-approximation of the program’s behavior, which is appropriate for testing.

Tip: The exercises in this document are for experienced .NET developers and Pex users.

Unlike the Pex and Moles tutorials, these exercises offer challenges for you to explore concepts in working code, rather than offering step-by-step solutions.
Exercise 1: The Path Condition

**Part 1. Solve the Constraints of Program 1**

Call Program 1 from a parameterized unit test, and inspect which values Pex uses as test inputs:

- Do the values match your expectations?
- Would another set of values be more appropriate for testing?
- If so why? If not, why not?

**Part 2. Play with Program 1**

Revisit Program 1 and see how Pex handles this example:

1. Add a new test in the project, copy the source of Program 1, and turn it into a parameterized test.

2. To gain more insight on what is happening, explore the `GetPathConditionString` method of the `PexSymbolicValue` class that returns the current path condition.

   The following code adds a column to the parameter table and fills it with the path condition:

   ```java
   int Complicated(int x, int y) {
       if (x == Obfuscate(y))
           throw new RareException();

       // logging the path condition
       string pc = PexSymbolicValue.GetPathConditionString();
       PexObserve.ValueForViewing("pc", pc);

       return 0;
   }
   ```

   If you run the above code as part of a parameterized unit test, you will notice that the path condition is only added to the table when no exception is thrown.

   To make sure that the path condition is added in any case, you can embed the logging code into a `try-finally` block.

3. Execute Pex and analyze the results. Do they match your understanding?

4. To learn more about the Pex symbolic analysis, look at the `ToString` method of the `PexSymbolicValue` class that returns a string that represents the Pex current knowledge about how the given value was derived from the inputs:

   ```java
   int Complicated2(int x, int y) {
       if (x == Obfuscate(y)) {
           // logging the path condition
           string obfuscateY = PexSymbolicValue.ToString(Obfuscate(y));
           PexObserve.ValueForViewing("obfuscate(y)", obfuscateY);
           throw new RareException();
       }

       return 0;
   }
   ```
Part 3. Try Additional Path Condition Samples

Use the `GetPathConditionString` and `ToString` methods of the `PexSymbolicValue` class with other examples.

1. Explore dealing with loops:

```csharp
public void PathConditionLoop(int x, int y) {
    for (int i = 0; i < x; ++i)
        if (x == y + 1) {
            string pc = PexSymbolicValue.GetPathConditionString();
            PexObserve.ValueForViewing("pc", pc);
        }
}
```

2. Explore dealing with strings:

```csharp
void PathConditionSubstring(string s, int x, string y) {
    if (s.Substring(x) == y) {
        string pc = PexSymbolicValue.GetPathConditionString();
        PexObserve.ValueForViewing("pc", pc);
    }
}

void PathConditionStringMatch(string s) {
    if (s == "Hello"){
        string pc = PexSymbolicValue.GetPathConditionString();
        PexObserve.ValueForViewing("pc", pc);
    }
}
```

Monitoring by Instrumentation

Pex monitors the execution of a .NET program through code instrumentation. Pex plugs into the .NET profiling API. Before any method is compiled from the MSIL to native machine code by the Just-In-Time compiler of .NET, the Pex profiler gets a callback to inspect and rewrite the instructions. All .NET language compilers translate into this instruction set.

Pex operates on the level of these instructions. Pex does not care in which high level language the program was written. However, when Pex gives feedback to you in the form of code snippets, Pex supports only C# syntax at this time.

The instrumented code drives a “shadow interpreter” in parallel to the actual program execution. The “shadow interpreter”:

- Constructs symbolic representations of the executed operations over logical variables instead of the concrete program inputs.
- Maintains and evolves a symbolic representation of the entire program’s state at any point in time.
- Records the conditions over which the program branches.

The Pex interpreter models the behavior of all verifiable .NET instructions precisely, and also models most unverifiable (involving unsafe memory accesses) instructions.
Symbolic State Representation

A symbolic program state is a predicate over logical variables together with an assignment of locations to expressions over logical variables, just as a concrete program state is an assignment of locations to values. The locations of a state can be static fields, instance fields, method arguments, locals, and positions on the operand stack.

The Pex expression constructors include primitive constants for all basic .NET data types—integers, floating point numbers, object references—and functions over those basic types representing particular machine instructions such as addition and multiplication. Pex uses tuples to represent .NET value types (“structs”).

Pex uses maps to represent instance fields and arrays, similar to the heap encoding of ESC/Java: an instance field of an object is represented by a single map that associates object references with field values. Constraints over the .NET type system and virtual method dispatch lookups are encoded in expressions as well. Predicates are represented by Boolean-valued expressions.

Pex implements various techniques to reduce the enormous overhead of the symbolic state representation. Before building a new expression, Pex always applies a set of reduction rules that compute a normal form. A simple example of a reduction rule is constant folding; for example, 1+1 is reduced to 2. All logical connectives are transformed into a Behavior Driven Development (BDD) representation with if-then-else expressions.

All expressions are hash-consed, which means that only one instance is ever allocated in memory for all structurally equivalent expressions. Pex also employs independent constraint optimization.

Based on the already accumulated path condition, expressions are further simplified. For example, if the path condition already established that \( x > 0 \), then \( x < 0 \) simplifies to false.

Although strings can be represented in a way similar to arrays of characters, Pex represents certain string constraint as expressions using specialized string-functions. When running the following example, Pex will print to the console how the state is represented symbolically, including the path condition, at a particular program point.

```csharp
static int Global;
[PexMethod]
public void SymbolicStateExample(int x, int y)
{
    Global = 1 + x + x * x + x * x - 1;
    if (y > Global)
    {
        Console.WriteLine("Global=");
        PexSymbolicValue.ToRawString(Global);
        Console.WriteLine("x=");
        PexSymbolicValue.ToRawString(x);
        Console.WriteLine("y=");
        PexSymbolicValue.ToRawString(y);
        Console.WriteLine("pc=");
        PexSymbolicValue.GetRawPathConditionString();
    }
}
```
ToRawString and GetRawPathConditionString return expressions representing symbolic values and the path condition, formatted as S-expressions. Here, Pex will print the following:

\[
\begin{align*}
\text{Global} &= (\text{Add} (\text{Mul} (\text{Exp} \ x \ 2) \ 2) \ x) \\
\ x &= x \\
\ y &= y \\
\ pc &= (\text{Clt} (\text{Add} (\text{Mul} (\text{Exp} \ x \ 2) \ 2) \ x) \ y)
\end{align*}
\]

Add, Mul, Exp are binary functions representing addition, multiplication, and exponentiation, and Clt is a predicate that compares whether the left operand is less than the second operand. On the right sides, x and y represent the symbolic test inputs.

Note that the expressions have been normalized by Pex.

Exercise 2: Heap Constraints
1. Consider the following type:

```csharp
public class C { public int F; }
```

This type is used in the following method:

```csharp
public void AliasChallenge(C x, C y) {
    if (x != null) {
        if (y != null) {
            x.F = 42;
            y.F = 23;
            // if (x.F == 42) throw new Exception("boom");
        }
    }
}
```

- How many execution paths will Pex have to explore?
  
Pex only generates different test inputs when they exercise different execution paths.

- How many are there if we include the commented code?

2. Again, we can inspect the constraints that Pex collects internally:

GetPathConditionString shows us a pretty-printed string that bends the Pex internal representation into a C#-like syntax.

3. Try out GetRawPathConditionString instead to get a view on the raw expressions that Pex uses internally. These raw expressions are formatted as S-expressions.

In this example, you will see functions called select and update. These functions operate on maps that describe the evolution of and accesses to the heap.
Search Strategy

Pex builds the (reachable portion of the) execution tree of the program from all previously discovered execution paths. In each step of the test generation algorithm, Pex picks an outgoing unexplored branch of the tree, representing an execution path that has not been discovered yet.

Pex implements a fair choice between all such unexplored branches. Pex includes various fair strategies that partition all branches into equivalence classes, and then chooses a representative of the least often chosen class.

The equivalence classes cluster branches by mapping them to the following:

- The branch statement in the program of which the execution tree branch is an instance. Each branch statement might give rise to multiple execution tree branches—for example, when loops are unfolded.
- The stack trace at the time the execution tree branch was created.
- The depth of the branch in the execution tree.

Pex combines all such fair strategies into a meta-strategy that performs a fair choice among the strategies.

For a detailed description of the search strategies in Pex, see “Fitness-Guided Path Exploration in Dynamic Symbolic Execution,” listed in “Resources and References” later in this document.

Exercise 3: Search Frontiers

1. Consider the following parameterized unit test. It has two loops, each of which can make 32 separate decisions:

   ```csharp

   [PexMethod, PexSearchFrontierDepthFirst]
   public void TestWithLoops(int x, int y) {
       int xbits = 0;
       for (int i = 0; i < 32; i++)
           if (((1 << i) & x) != 0) xbits++;

       int ybits = 0;
       for (int i = 0; i < 32; i++)
           if (((1 << i) & y) != 0) ybits++;

       PexObserve.ValueForViewing("xbits", xbits);
       PexObserve.ValueForViewing("ybits", ybits);
   }
   ```

2. Try the different search frontiers.

   Additional search frontier implementation can be found under the `Microsoft.Pex.Framework.Strategies` namespace.

3. Can you explain what you see?

   Note that, by default, Pex does not emit a test for each path it explores.
Constraint Solving

For each chosen unexplored branch, Pex builds a formula that represents the condition under which this branch can be reached.

Pex employs Z3 as its constraint solver. Z3 is a Satisfiability Modulo Theories (SMT) solver, an automated satisfiability checker for first-order logic with several built-in theories. This makes Z3 an efficient automatic theorem prover. Historically, SMT solvers have been used mainly to perform program correctness proofs, for example as part of ESC/Java and Spec. In addition to deciding satisfiability, Z3 can also compute a model (satisfying assignments) for a satisfiable formula, which makes it an ideal constraint solver for Pex.

Pex faithfully encodes all constraints arising in safe .NET programs such that Z3 can decide them with its built-in decision procedures for propositional logic, fixed-sized bit-vectors, tuples, arrays and quantifiers. Pex also has a specialized string solver that is integrated with Z3. Arithmetic constraints over floating point numbers are approximated by a translation to rational numbers, and heuristic search techniques are used outside of Z3 to find approximate solutions for floating point constraints. Pex encodes the constraints of the .NET type system and virtual method dispatch lookup tables as universally quantified formulae.

Exercise 4: Constraint Solving

1. Use Pex to solve constraint systems written in C#.
   
   For example, Pex can find the two prime factors of 52605271 with a test like the following. Random testing would most likely not be able to solve this problem:

   ```csharp
   [PexMethod]
   public void DetermineFactors(int x, int y) {
     if (x > 1 && x < 10000 &&
         y > 1 && y < 10000 &&
         x * y == 52605271)
       throw new Exception("foundit");
   }
   ```

2. What are the two prime factors?
3. Can you write other interesting constraint systems and solve them with Pex?

Implicit Branches

Pex treats all possible deterministic exceptional control flow changes in the code like all other explicit branches: It tries to explore the successful case as well as the exceptional outcome.

The following exceptions that the CLR execution engine can throw fall into this category:

- `NullReferenceException`
- `IndexOutOfRangeException`
- `OverflowException`
- `InvalidCastException`
- `DivisionByZeroException`
Pex does not systematically try to throw the following exceptions that the execution engine might raise non-deterministically:

- `OutOfMemoryException`
- `StackOverflowException`
- `ThreadAbortException`

In fact, by default Pex tries hard to avoid them so that only perfectly deterministic execution paths are explored.

**Exercise 5: Implicit Branches**

1. Consider the following method, which has two execution paths:

   ```csharp
   public void ImplicitNullCheck(int[] a) {
       int x = a.Length;
   }
   ```

2. How many paths will Pex explore in the following method?

   Note that Pex checks for each possible exception type separately and considers checks for different exception types as different branches:

   ```csharp
   public void ImplicitIndexOutOfRangeCheck(int[] a) {
       int x = a[0];
   }
   ```

   Pex understands checked code as well. Pex finds input that will cause the following method to throw an `OverflowException`.

   ```csharp
   public void ImplicitOverflowCheck(int x, int y) {
       int z = checked(x + y);
   }
   ```

3. Can you write a parameterized unit test that could cause an `InvalidCastException`?

**Understanding Assumptions, Assertions, and Test-Case Failures**

**Assumptions and Assertions**

You can use assumptions and assertions to express preconditions (assumptions) and post-conditions (assertions) of tests. Although Pex tries different argument values during the exploration of the code, Pex might inadvertently violate an assumption. When that happens, Pex remembers the condition that caused the assumption violation, so that Pex will not generate another test that violates the assumption. The test case itself that violated the assumption is silently pruned.

The concept of assertions is well known in unit test frameworks. Pex understands the built-in `Assert` classes provided by each supported test framework. However, most frameworks do not provide a corresponding `Assume` class. For that case, Pex provides the `PexAssume` class:

```csharp
using Microsoft.Pex.Framework;

[PexClass]
public partial class MyTests{
    [PexMethod]
    public void Test1(object o) {
```
If you do not want to use an existing test framework, Pex also has the **PexAssert** class. The non-nullness assumption can also be encoded as a custom attribute:

```csharp
using Microsoft.Pex.Framework;
public partial class MyTests {
    [PexMethod]
    public void Test2([PexAssumeNotNull] object o) {
        // precondition: o should not be null
        ...
    }
}
```

When you write an assertion, Pex will not only passively detect assertion violations, but Pex will in fact actively try to compute test inputs that will cause the assertion to fail. The reason is simply that the assert statement is implemented as a conditional branch, which throws an exception in the failure case, similar to the following code:

```csharp
public class PexAssert {
    public static void IsTrue(bool condition) {
        if (!condition)
            throw new PexAssertionViolationException();
    }
}
```

Just like for any other branch in the code, Pex will build constraint systems that aim to invalidate the asserted condition for each calling context. When Pex can solve the constraint system for the negated condition, we get test inputs that exhibit an error. Just as an assertion might throw a **PexAssertFailedException**, Pex uses a **PexAssertFailedException** internally to stop a test case when an assumption fails.

### When Does a Test Case Fail?

Pex considers the exceptional behavior of a test—whether the test throws an exception that is not caught—to decide whether a test case fails or succeeds:

- If the test does not throw an exception, it succeeds.
- If the test throws an exception that is:
  - A **PexAssumeFailedException**, it succeeds. But it is usually filtered out, unless the **TestEmissionFilter** is set to **All**.
  - A **PexAssertFailedException** or any other assertion violation exception of other unit test framework, it fails.
  - Neither an assumption nor an assertion violation exception, it depends on further annotations whether the test passes or fails.
You can annotate the test—or the test class, or the test assembly—with one of the following attributes to indicate which exception types can or must be thrown by the test in order to be considered successful:

- **PexAllowedExceptionAttribute** indicates that a test method, or any other method that it calls directly or indirectly, can throw a particular type of exception for some test inputs.

- **PexAllowedExceptionFromTypeAttribute** indicates that any method of a specified type can throw a particular type of exception for some test inputs.

- **PexAllowedExceptionFromTypeUnderTestAttribute** indicates that any method of the designated type under test can throw a particular type of exception for some test inputs.

- **PexAllowedExceptionFromAssemblyAttribute** indicates that any method located in a specified assembly can throw a particular type of exception for some test inputs.

**When Does Pex Emit a Test Case?**

Pex supports different filters that decide when generated test inputs will be emitted as a test case. You can configure these filters with the **TestEmissionFilter** property that you can set for example in the **PexMethod** attribute. Possible values are the following.

- **All**
  Emit every generated test input as a test case, including those that cause assumption violations.

- **FailuresAndIncreasedBranchHits** (default)
  Emit tests for all unique failures, and whenever a test case increases coverage, as controlled by the **TestEmissionBranchHits** property (see below).

- **FailuresAndUniquePaths**
  Emit tests for all failures Pex finds, and also for each test input that causes a unique execution path.

- **Failures**
  Emit tests for failures only.

Regarding increased branch coverage, the **TestEmissionBranchHits** property controls how a branch is covered. For example:

- **TestEmissionBranchHits=1**: Whether Pex should just consider whether a branch was covered at all.
  
  **TestEmissionBranchHits=1** gives a very small test suite that covers all branches Pex could reach. In particular, this test suite also covers all reached basic blocks and statements.

- **TestEmissionBranchHits=2**: Whether a test covered it either once or twice.
  The default is **TestEmissionBranchHits=2**, which generates a more expressive test suite that is also better suited to detect future regression errors.
Exercise 6: Test Case Emission Filters

1. Consider the following `max` methods:
   ```csharp
   int max(int x, int y) {
     if (x > y)
       return x;
     else
       return y;
   }

   int max(int a, int b, int c, int d) {
     return max(max(a, b), max(c, d));
   }
   ```

2. Consider the following `PexMethod`:
   ```csharp
   [PexMethod]
   public void MaxTest(int a, int b, int c, int d) {
     int e = max(a, b, c, d);
     PexObserve.ValueForViewing("max", e);
   }
   ```

3. How many execution paths does it have?
   Can you make Pex emit test cases for all execution paths?
   How many tests will Pex emit with `TestEmissionBranchHits=1`?
   What is the relation between those two numbers?

When Does Pex Stop?

If the code under test does not contain loops or unbounded recursion, Pex will typically stop quickly because there is only a (small) finite number of execution paths to analyze. However, most interesting programs contain loops and/or unbounded recursion. In such cases the number of execution paths is (practically) infinite, and it is in general undecidable whether a statement is reachable. In other words, Pex would take forever to analyze all execution paths of the program.

In order to make sure that Pex terminates after a reasonable amount of time, there are several exploration bounds. All bounds have predefined default values, which you can override to let Pex analyze more and longer execution paths. The bounds are parameters of the `PexMethod`, `PexClass`, and `PexAssemblySettings` attributes. There are different kinds of bounds: Constraint Solving Bounds apply to each attempt of Pex to determine whether an execution path is feasible.

Pex might need several constraint solving attempts to compute the next test inputs:

- **ConstraintSolverTimeOut**
  Seconds the constraint solver has to figure out inputs that will cause a different execution path to be taken.

- **ConstraintSolverMemoryLimit**
  Megabytes the constraint solver can use to figure out inputs.
Exploration Path Bounds apply to each execution path that Pex executes and monitors. These bounds make sure that the program does not get stuck in an infinite loop, or recursive method:

- **MaxBranches**
  Maximum number of branches that can be taken along a single execution path.

- **MaxCalls**
  Maximum number of calls that can be taken during a single execution path.

- **MaxStack**
  Maximum size of the stack at any time during a single execution path, measured in number of active call frames.

- **MaxConditions**
  Maximum number of conditions over the inputs that can be checked during a single execution path.

Exploration Bounds apply to the exploration of each parameterized unit test.

- **MaxRuns**
  Maximum number of runs that will be tried during an exploration (each run uses different test inputs; not every run will result in the emission of a new test case).

- **MaxRunsWithoutNewTests**
  Maximum number of consecutive runs without a new test being emitted.

- **MaxRunsWithUniquePaths**
  Maximum number of runs with unique execution paths that will be tried during an exploration.

- **MaxExceptions**
  Maximum number of exceptions that can be found over all discovered execution paths combined.

- **MaxExecutionTreeNodes**
  Maximum number of conditions over the inputs that can be checked during all discovered execution paths combined.

- **MaxWorkingSet**
  Maximum size of working set in megabytes.

- **TimeOut**
  Seconds after which exploration stops.

The following example shows a parameterized test that involves a loop. The loop bound depends on the test inputs, and the lurking exception can only be triggered if the loop is unfolded a certain number of times. Here, we used an explicit bound of 10 runs—**MaxRuns=10**—to let Pex finish quickly. However, with this bound, Pex will most likely not be able to trigger the exception, as Pex will not unroll the loop sufficiently many times:

```csharp
[PexMethod(MaxRuns = 10)]
public void TestWithLoop(int n) {
    var sum = 0;
    for (int i = 0; i < n; i++)
        sum++;
}```
if (sum > 20) throw new Exception();

In the Pex tool bar, you will see that the boundary button is enabled:

![Figure 1. Boundary button](image)

When you click the button, Pex shows which bounds were exceeded, and it offers several actions to increase the bounds:

![Figure 2. Pex message for exceeded bounds](image)

Select SetMaxRuns=20 to double the bound. This will update the specified bounds in the source code. Run Pex again. If Pex is still not able to trigger the exception, you might have to double the bound again.

The following example shows another parameterized test that involves a loop, but this loop does not depend on the test inputs. Here, we used an explicit bound of 10 branches—MaxBranches=10—to let Pex finish quickly. However, with this bound, Pex cannot even once execute the code from beginning to end, as executing the embedded loop will cause more than 10 branches to be executed:

```csharp
[PexMethod(MaxBranches=10)]
public void TestWithFixedLoop(int j) {
    var sum = 0;
    for (int i = 0; i < 15; i++)
        sum++;
    if (j == 10) throw new Exception();
}
```

In those cases, where a particular run exceeded some path-specific bounds, we get a special row with the words “path bounds exceeded” in the table of all generated test cases. Click the row to see more detailed information about the exceeded bound.

The Set MaxBranches= button can be used to increase the bounds. The button in the tool bar is enabled as well.
Click the button, increase the bounds, and run Pex again. If Pex can still not find the exception, you might have to increase the bounds again.

How Does Pex Suggest Fixes for Errors?

Sometimes, when a test run fails, Pex can suggest a potential change to the source code to prevent the same failure from happening again.

Algorithm 2 shows how Pex tries to locate the failure cause.

Algorithm 2. Fix It

1. Pex determines the point in the execution trace of the test case where the failure manifested itself—where an exception was thrown that was not caught.
2. Pex looks back in the execution trace for the last public method call.
3. Pex computes a condition under which the failure happened, relative to the last public method call.
4. If the condition only involves parameters, Pex will suggest the negation of the condition as a missing precondition.
5. Otherwise, Pex will try to express the condition in terms of the fields of the class of the last public method call.
   Pex will suggest the negation of the condition as a missing invariant—a condition over the fields of the class which should always hold. Furthermore, Pex will execute the test again, and try to find which public method of this class first left the object behind in a state that violates the suggested invariant.
   If Pex finds such a method, Pex will start over at Step 3.

This algorithm terminates, because in each iteration, Pex tries to find the failure cause at an earlier point in the execution trace. If Pex suggests a missing precondition or invariant, following that advice is guaranteed to prevent the same failure from happening again, under the proviso that Pex could monitor all relevant parts of the program. If this proviso is violated, Pex could suggest a precondition or an invariant that is inconsistent, which basically means that Pex suggests to not run this code again.
Understanding Complex Objects, Explorables, Invariants, and Limitations

Complex Objects

Pex monitors the executed instructions when it runs a test and the code-under-test. In particular, it monitors all field accesses. It then uses a constraint solver to determine new test inputs—including objects and their field values—such that the test and the code-under-test will behave in other interesting ways.

Thus, Pex needs to create objects of certain types and set their field values. If the class is visible and has a visible default constructor, Pex can create an instance of the class. If all the fields of the class are visible, Pex can set the fields automatically. If the type is not visible, or the fields are not visible, Pex needs help to create objects and bring them into interesting states to achieve maximal code coverage.

There are two ways to help Pex:

- The first is for the user to provide parameterized factories for the complex objects such that Pex can explore different object states, starting from factory methods.
- The second is to define the invariants of the object’s private fields, such that Pex can manufacture different object states directly.

Explorables

Consider the following class:

```java
public class MyCounter {
    private int _count;
    public MyCounter(int initialCount) {
        if (initialCount < 0) throw new ArgumentException();
        this._count = initialCount;
    }
    public MyCounter() {
        this._count = 0;
    }
    public void Increment() { this._count++; }
    public int Count { get { return this._count; } }
}
```

Consider the following test where something bad happens when the counter reaches the number 99:

```java
[PexMethod]
public void TestMyCounter([PexAssumeNotNull]MyCounter counter) {
    if (counter.Count == 99) throw new Exception();
}
```

When you run Pex, it might not find the exception, but instead Pex might show you a warning such as the following:

```
Pex guessed how to create instances of MyCounter: new MyCounter()
```

This means that, not knowing all ways to construct meaningful counters, Pex chose the default constructor that takes no parameters. Of course, with that constructor we would have to call the `Increment()` method 99 times in order to hit the exception, which Pex would do only after a very long exploration time.
We can improve the situation by telling Pex to use the other constructor. You can define a factory method that Pex can use to create instances of a given type. In fact, when you use Pex from Visual Studio, you get the option to create a factory method when necessary. The default code and attributes for such a factory are similar to the following:

```csharp
public static class Factory {
    [PexFactoryMethod(typeof(MyCounter))]
    public static MyCounter CreateMyCounter() {
        return new MyCounter();
    }
}
```

You can change this factory method as desired. You can add parameters, and then Pex will determine which values are relevant. The following is a factory method that creates `MyCounter` instances by calling the constructor that takes one parameter:

```csharp
[PexFactoryMethod(typeof(MyCounter))]
public static MyCounter CreateMyCounter(int x) {
    return new MyCounter(x);
}
```

With these annotations in place, when you run Pex again, it will explore the constructor that takes an initial count. Pex will find a test case such as the following:

```csharp
[TestMethod]
[PexRaisedException(typeof(Exception))]
public void TestMyCounter_MyCounter_71115_194002_0_02() {
    MyCounter mc0 = new MyCounter(99);
    this.TestMyCounter(mc0);
}
```

As an alternative to factory methods, you can indicate in a declarative way which constructor to use with the following attribute:

```
[assembly: PexExplorableFromConstructor(            typeof(MyCounter),            typeof(int))]
```

The first argument indicates the type to construct. The following arguments are the parameter types for the desired constructor.

### Invariants

**Tip:** The section about invariants is for advanced developers.

It is often sufficient, and much easier, if you can get Pex to create objects through the explorables described earlier in the “Explorables” section.

Even when you tell Pex which constructor to use, or even when you write a factory method by hand, Pex might have a hard time exploring all combinations of the constructor and the mutator methods.

The example that we showed earlier in “Explorables” had a nice property: All possible and legal configurations of the `MyCounter` class could be constructed by calling the constructor. The constructor was written to throw an exception when an attempt is made to configure an invalid object. When a class does not have such a constructor, you can always write such a special constructor for testing purposes: a constructor that allows you to configure the object in all possible and legal ways.
To describe what is possible and legal, you have to explicitly define the condition under which the fields of an object are properly configured. This condition is called the class invariant. You can write an invariant as a private parameterless instance method that returns `bool`. For example, the invariant of the array list class can be expressed as follows:

```csharp
private bool Invariant() {
    return this._items != null &&
    this._size >= 0 &&
    this._items.Length >= this._size;
}
```

Now you can define the special public constructor for testing purposes. It simply receives all field values as parameters. When the supplied parameter values do not satisfy the invariant, the constructor is aborted by throwing an exception. One way to make sure that this constructor is used only for testing purposes is to define it conditionally:

```csharp
#if DEBUG
public ArrayList (object[] items, int size){
    this._items=items;
    this._size=size;
    if (!this.Invariant()) throw new InvalidOperationException();
}
#endif
```

Another way to make sure this constructor is used only for testing purposes is to define it as `internal`, and use the `InternalsVisibleToAttribute` attribute to expose it to the assembly that contains the tests.

Now you can tell Pex to use this constructor, as discussed in the “Explorables” section:

```csharp
[assembly: PexExplorableFromConstructor(typeof(ArrayList),
    typeof(object[]), typeof(int), typeof(int))]
```

Pex will explore the constructor and the conditions in the `Invariant` method, filtering out all configurations where the invariant does not hold.

As an alternative to defining `DEBUG`-only constructors, Pex can also leverage designated invariant methods that can be specified in conjunction with the Code Contracts framework. For information about how to write and leverage invariants with code contracts, see “Parameterized Unit Testing with Microsoft Pex” in the Microsoft Pex documentation.

### Limitations

There are certain situations in which Pex cannot analyze the code properly:

- **Nondeterminism.** Pex assumes that the analyzed program is deterministic. If it is not, Pex will prune non-deterministic execution paths, or it might go in cycles until it hits exploration bounds.

- **Concurrency.** Pex does not handle multithreaded programs. It might work in a scenario where the main thread execution is deterministic, independent of the behavior of other spawned threads.
CHESS is a project that is similar to Pex, but instead of exploring the non-determinism induced by input values, CHESS explores the non-determinism induced by different thread-interleavings in multithreaded applications.

- **Native Code or .NET code that is not instrumented.** Pex does not understand native code—that is, x86 instructions called through the Platform Invoke (P/Invoke) feature of the .NET Framework.
  
Pex does not know how to translate such calls into constraints that can be solved by a constraint solver. And even for .NET code, Pex can only analyze code it instruments.
  
However, even if some methods are uninstrumented, Pex will still try to cover the instrumented code as much as possible.

- **Language.** In principle, Pex can analyze arbitrary .NET programs, written in any .NET language. However, the Visual Studio add-in and Pex code generation only support C#.

- **Symbolic Reasoning.** Pex uses an automatic constraint solver to determine which values are relevant for the test and the code-under-test. However, the abilities of the constraint solver are, and always will be, limited. In particular, Z3 cannot reason precisely about floating point arithmetic.

---

**Resources and References**

**Pex Resources, Publications, and Channel 9 Videos**

Pex and Moles at Microsoft Research

http://research.microsoft.com/pex/

Pex Documentation Site

**Pex and Moles Tutorials**

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Technical Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting Started with Microsoft Pex and Moles</td>
<td>200</td>
</tr>
<tr>
<td>Getting Started with Microsoft Code Contracts and Pex</td>
<td>200</td>
</tr>
<tr>
<td>Unit Testing with Microsoft Moles</td>
<td>200</td>
</tr>
<tr>
<td>Exploring Code with Microsoft Pex</td>
<td>200</td>
</tr>
<tr>
<td>Unit Testing Asp.NET applications with Microsoft Pex and Moles</td>
<td>300</td>
</tr>
<tr>
<td>Unit Testing SharePoint Foundation with Microsoft Pex and Moles</td>
<td>300</td>
</tr>
<tr>
<td>Unit Testing SharePoint Foundation with Microsoft Pex and Moles <img src="II" alt="II" /></td>
<td>300</td>
</tr>
<tr>
<td>Parameterized Unit Testing with Microsoft Pex</td>
<td>400</td>
</tr>
</tbody>
</table>

**Pex and Moles Technical References**

- Microsoft Moles Reference Manual | 400 |
- Microsoft Pex Reference Manual | 400 |
- Microsoft Pex Online Documentation | 400 |
- Parameterized Test Patterns for Microsoft Pex | 400 |
- Advanced Concepts: Parameterized Unit Testing with Microsoft Pex | 500 |
Community

- Pex Forum on MSDN DevLabs
- Pex Community Resources
- Nikolai Tillmann’s Blog on MSDN
- Peli de Halleux’s Blog

References and Related Work

Testing with algebraic specifications was started by Gaudel et al. They use axioms in various ways:

- To describe the test purpose.
- To obtain concrete data, which is necessary for the instantiations of the axioms.
- To derive new theorems, which when tested should uncover errors in state not captured by the model. For deriving those theorems, they introduced regularity assumptions.

A parameterized unit test is a way to write algebraic specification as code. Another name for it is theories in the JUnit test framework, where Saff et al. found them to be effective. They appear as row tests in MbUni, and under other names in various other unit test frameworks. It has also been investigated how such algebraic axioms can be synthesized automatically from a given implementation, possibly supported by a given test suite.

In order to generate test inputs for parameterized unit tests, Pex uses dynamic symbolic execution. A recent overview on the combination of static and dynamic analysis can be found in Godefroid et al. The basic idea of symbolic execution was introduced more than three decades ago. By combining it later with other work on dynamic test generation, DART was the first practical tool implementing dynamic symbolic execution for C programs.

Many other implementations have been recently developed as well: CUT, EXE, KLEE for C programs, jCUTE for Java programs, and SAGE for x86 code.

Related to dynamic symbolic execution is model-checking of programs. XRT is a model checker for .NET programs, JPF for Java programs. Both JPF and XRT have extensions for symbolic execution. However, both can only perform static symbolic execution, and they cannot deal with stateful environment interactions.

Many other approaches to automatic test generation exist. For example, Randoop is a tool that generates new test-cases by composing previously found test-case fragments, supplying random input data.


