****

Research Report

|  |
| --- |
| (CW228)< Another Bugs Contraption Detector >Supervisor : Christophe MeudecDevelopers : Shengjie Yuan C00132795Date of submission :26.11.2010 |

Context:

**Abstract……………………………………………………………………………………..1**

1. **Introduction……………………………………………………………… …………1**
	1. **What is Bugs Contraption Detector………………………………………1**
	2. **Project Goal:…………………………………………………………………………1**
2. **Similar Application: Pex……………………………………………….……….2**
	1. **What is Pex?...........................................................................................2**
	2. **Simple example about testing code form Pex…………………….…3**
	3. **How Pex work?........................................................................................5**
		1. **Pex`s overview diagram……………………………………………………5**
		2. **Program analysis techniques……………………………………………6**
		3. **Parameterized unit testing** **(PUT)……………………….……………7**
		4. **Symbolic Execution………………………………………………….……….8**

## Dynamic Symbolic Execution…………………………………………………...9

* + 1. **.NET Profiling API............................................................................11**
		2. **MSIL………………………………………………………………………………..13**
		3. **Constraints Solver…………………………………………………………..15**
1. **Processes of testing code………………………………………..……………………..…….16**
2. **Available programming language……………………………………….21**
	1. **JAVA……………………………………………………………….…………………..21**
		1. **Description of Java……………………………….…………………………21**
		2. **Discusses about JAVA………………………………………………………21**
	2. **C#...............................................................................................................22**
		1. **Description of C#.............................................................................22**
3. **Available Complier……………………………………………………………23**
	1. **Netbean…………………………………………………………………………….23**
	2. **Visual Studio……………………………………………………………………..23**
4. **Conclusion…………………………………………………………………………23**

**References………………………………………………………………………………26**

**Abstract**

This document will research necessary information about the project (Another Bugs Contraption Detector) in the following sections, it includes:

* Make a brief introduction about this project and its goal.
* Provide a similar application, make a definition of what is it and through two examples to explain how the work.
* Discuss possible compiling language, such as: JAVA, C#....
* Discuss possible Complier, such as: Netbean, Visual Studio…
1. **Introduction**
	1. **What is Bugs Contraption Detector**

The use area of this Bugs Contraption Detector is about software testing, and the Code Parsing, Code Instrumentation technologies would be used. This project allows automatic test inputs generation from .NET source code, it tries to generate input automatically. The ability to instrument, for test inputs generation purposes, during normal execution, information that are sent to an external constraints solver for automatic test data generation. For the additional function, it will try to make the target programming language useful and/or find problems in real code.

* 1. **Project Goal:**

This project tries to achieve the following goal:

Given a method with parameters, automatically generate a set of input values that, upon execution, will exercise as many statements as possible in the method.

In other words, this project seeks to generate a test suite with high code coverage, where each test case invokes the method-under-test with particular values.

1. **Similar Application: Pex**
	1. **What is Pex?**

“*Microsoft Pex 2010 is a Microsoft® Visual Studio® add-in that provides a runtime code analysis tool for.NET Framework code. The output of the tool helps you understand the behavior of the code, and it also builds automated tests with high code coverage. It is a testing tool that performs systematic code analysis, hunts for boundary conditions and flags exceptions and assertion failures.*”[01]

Through a context menu in the Visual Studio editor, you can invoke Pex to analyze an entire class or a single method. For any method, Pex computes and displays input-output pairs. Pex systematically hunts for bugs, exceptions, or assertion failures. Pex will run as the following diagram.

**Run Pex:**



**Result of test output:**



* 1. **Simple example about testing code form Pex**

This code is a simple example about how pex work, it includes how pex create input, monitors conditions and stop.

|  |
| --- |
| “ Public class Program { public static void Puzzle(int[] v) { if (v != null && v.Length > 0 && v[0] == 1234567890) throw new Exception("hidden bug!"); }} ” [02] |

“*Pex performs the following process in an iterative fashion.*

* *Pex starts by choosing some inputs. Pex always begins with the simplest inputs possible. In the above example, the input would be null for the array parameter.*
* *Then Pex executes the code, and monitors all conditions that are checked, along the execution path that is taken for the chosen inputs.*
* *For the example above, the method returns because v == null.*
* *Pex negates this condition and queries a constraint solver to determine whether there is a solution for the negated condition.*
* *If a solution exists, then this solution represents another test input, which would cause the code to take a different path.*

*And then Pex repeats this process. Internally, Pex represents all conditions that the program checks as a tree.*

 **

*Every time Pex executes the code again, the execution tree might grow, as Pex learns about new behaviors of the code.

After a few iterations, Pex finishes for the following example.*

 **

 *When the tree grows big, Pex can choose during each iteration of the process which direction to extend the tree. For this example, this choice is always simple, because the code has no complicated branching structure, and no loops.*” [03]

* 1. **How Pex work?**
		1. **Pex`s overview diagram**

“*Pex generates input values by analyzing the program code. For every statement in the code, Pex eventually tries to create input values that will reach that statement. Pex performs a case analysis for every conditional branch in the code—for example, if statements, assertions, and all operations that can throw exceptions. In other words, the number of input values that Pex generates depends on the number and possible combinations of conditional branches in the code. Pex operates in a feedback loop: it executes the code multiple times and learns about the program's behavior by monitoring the control and data flow.*

* *After each run, Pex does the following:*
* *Chooses a branch that was not covered previously.*
* *Builds a constraint system that describes how to reach that branch.*
* *Uses a constraint solver to determine new input values that fulfill the constraints, if any exist.*

**

*The code is executed again with the new input values, and the process repeats. On each run, Pex might discover new code and dig deeper into the implementation. In this way, Pex explores the behavior of the code.*” [04]

* + 1. **Program analysis techniques**

Static analysis techniques verify that a property holds on all execution paths.

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.

“*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.*” [04]

API Contracts—Spec#, Eiffel, and so on—specify the behavior of individual API actions from the implementation’s point of view.

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 correct-ness, 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).*”[05]

* + 1. **Parameterized unit testing** **(PUT)**

“*A PUT is simply a method that takes parameters. Developers write PUTs, just like traditional unit tests, at the level of the actual software APIs in the software projects programming language. The purpose of a PUT is to express an APIs intended behavior. For example, the following PUT asserts that after adding an element to a non-null list, the element is indeed contained in the list:*

*void TestAdd(ArrayList list, object element) {*

*Assume.IsNotNull(list);*

*list.Add(element);*

*Assert.IsTrue(list.Contains(element));*

*}*

*This PUT states assumptions on test inputs, performs a sequence of method calls, and asserts properties that should hold in the final state. (The initial assumptions and final assertions are similar to method preconditions and postconditions in the design- by-contract paradigm.)*

*Pex is a tool developed at Microsoft Research that helps developers write PUTs in a .NET language. For each PUT, Pex uses dynamic test-generation techniques to compute a set of input values that exercise all the statements and assertions in the analyzed program. For example, for our sample PUT, Pex generates two test cases that cover all the reachable branches:*

*void TestAdd\_Generated1() {*

*TestAdd(new ArrayList(0), new object());*

*}*

*void TestAdd\_Generated2() {*

*TestAdd(new ArrayList(1), new object());*

*}*

*The first test executes code (not shown here) in the array list that allocates more memory because the initial capacity 0 isn’t sufficient to hold the added element. The second test initializes the array list with capacity 1, which is sufficient to add one element. Pex comes with an add-in for Visual Studio that enables developers to perform most frequent tasks with a few mouse clicks. Also, when Pex generates a test that fails, Pex performs a root cause analysis and suggests a code change to fix the bug.*” [06]

* + 1. **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.

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.

“*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 unfolding. 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.*” [07]

## Dynamic Symbolic Execution

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, they 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. 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.”* [08]

**Why use 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. 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.”* [09]

**Algorithm for Dynamic symbolic execution**

/\*intuitively, J is the set of already analyzed program inputs\*/

Set J : = null ;

loop

Choose program input i such that ¬J(i)

stop if no such i can be found

Output i

Execute P(i); record path condition C /\*C(i) holds\*/

Set J := J \_ C /\*viewing C as the set {i | C(i)}\*/

end loop

* + 1. **.NET Profiling API**

“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.” [10]

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.

**Code instrumentation**
“The instruction sequence of each instrumented method is rewritten. It starts by inserted instructions which check whether monitoring is currently on. If not, the original instructions are executed. It if it turned on, the instrumented version of the instructions is execution, starting with a callback to EnterMethod(m, c), and eventually finished by a callback to LeaveMethod(m), where m identifies the current method, and c is the number of branch identifiers that the instrumented code will refer to within the method body. EnterMethod returns a Boolean value that indicates whether the monitor is interested in the concrete arguments to the method. If it returns true, a sequence of calls to Argument<T>(i, v), ArgumentPtr(i, v, t), ArgumentTypedReference(i, v) and ArgumentByRef<T>(i, a) is inserted, one for each argument, where i is the index of the argument, v is a value, t a type, and a a managed pointer. Each exception filter starts with a callback to StartFilter(r) where r is the reference to the exception object that is currently being filtered. Each exception handler starts with a callback to StartExceptionHandler(r).
Each instruction that can throw a [NullReferenceException](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/NullReferenceException.html), InvalidCastException, IndexOutOfRangeException, ArrayTypeMismatchException, DivideByZeroException or OverflowException is immediately followed by a callback that indicates that the instruction did not throw the exception. Also, all such instructions are surrounded by an exception handler, which catches the corresponding exception type, performs a callback to a method that indicates that the exception was thrown, and rethrows the exception. See the file checks.def in the Microsoft.ExtendedReflection.ClrMonitor project for detailed information about which instructions may throw which exceptions. In general, each instruction causes a callback named after the instruction itself. The callback is issued before the execution of the actual instruction. In order to capture all relevant information to track the execution path taken by the program, some instructions get a [special treatment](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Special%20callbacks.html).” [11]

Pex monitors the execution of a test or program by inserting callbacks into the executed instruction sequence. This is done by the component **Microsoft.ExtendedReflection**, which is a library that offers functionality that greatly extends the standard reflection library, and also lifts the unmanaged CLR profiling API into the managed world. [12]

The **ClrMonitor** [13] is an unmanaged COM component that implements the [CLR profiling API](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/CLR%20profiling%20API.html). “***It uses this API mainly to rewrite*** [***MSIL***](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/MSIL.html) ***instructions of selected methods, types or assemblies on-the-fly at JIT time. It can redirect calls and insert callbacks in the*** [***MSIL***](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/MSIL.html) ***instruction sequence***” [14], enabling the observation of the execution of the program at a fine grained level:

“**CLR**: common language runtime profiling, The Common Language Runtime (CLR) is a core component of [Microsoft's](http://en.wikipedia.org/wiki/Microsoft) [.NET](http://en.wikipedia.org/wiki/.NET_Framework) initiative. It is Microsoft's implementation of the [Common Language Infrastructure](http://en.wikipedia.org/wiki/Common_Language_Infrastructure) (CLI) standard, which defines an execution environment for program code.”[15]

 “**COM**: Component Object Model, Microsoft COM (Component Object Model) technology in the Microsoft Windows-family of Operating Systems enables software components to communicate.” [16]

* (Static) calls can be [redirected](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/CLRMONITOR_SUBSTITUTIONS.html). The redirection target may call the original call target.
* Similar to the standard CLR profiling API, the ClrMonitor monitors when a method is entered or exited.
* It also monitors the concrete parameter values that are passed to a monitored method.
* The execution of every instruction can be monitored.
* The control flow is monitored, i.e. the evaluation of all explicit conditional branches, and of all implicit checks that can cause exceptions.
* Every monitored event causes a callback into the core [**Microsoft.ExtendedReflection**](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Microsoft.ExtendedReflection.html) assembly, which can dispatch the callbacks to a user-provided [Execution Monitor](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Execution%20Monitor.html).

“**Monitoring.Controller** which is a class in [Microsoft.ExtendedReflection](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Microsoft.ExtendedReflection.html) which provides support to check if code in the current process can be monitored, methods to start and stop monitoring in the current process if enabled, and facilities to setup monitoring for a new process.”[17]

* + 1. **MSIL**

“*Common Intermediate Language (CIL, pronounced either "sil" or "kil") (formerly called Microsoft Intermediate Language or MSIL) is the lowest-level* [*human-readable*](http://en.wikipedia.org/wiki/Human-readable)[*programming language*](http://en.wikipedia.org/wiki/Programming_language) *defined by the* [*Common Language Infrastructure*](http://en.wikipedia.org/wiki/Common_Language_Infrastructure) *specification and used by the* [*.NET Framework*](http://en.wikipedia.org/wiki/.NET_Framework) *and* [*Mono*](http://en.wikipedia.org/wiki/Mono_%28software%29)*. Languages which target a CLI-compatible runtime environment compile to CIL, which is assembled into* [*bytecode*](http://en.wikipedia.org/wiki/Bytecode)*. CIL is an* [*object-oriented*](http://en.wikipedia.org/wiki/Object-oriented)[*assembly language*](http://en.wikipedia.org/wiki/Assembly_language)*, and is entirely* [*stack-based*](http://en.wikipedia.org/wiki/Stack_machine)*. It is executed by a* [*virtual machine*](http://en.wikipedia.org/wiki/Virtual_machine)*. CIL was originally known as Microsoft Intermediate Language (MSIL) during the beta releases of the .NET languages. Due to standardization of C# and the Common Language Infrastructure*

*During compilation of .NET programming languages, the* [*source code*](http://en.wikipedia.org/wiki/Source_code) *is translated into CIL code rather than platform or processor-specific* [*object code*](http://en.wikipedia.org/wiki/Object_file)*. CIL is a* [*CPU*](http://en.wikipedia.org/wiki/CPU)*- and platform-independent instruction set that can be executed in any environment supporting the Common Language Infrastructure, such as the* [*.NET runtime*](http://en.wikipedia.org/wiki/Common_Language_Runtime) *on* [*Windows*](http://en.wikipedia.org/wiki/Microsoft_Windows)*, or the* [*cross-platform*](http://en.wikipedia.org/wiki/Cross-platform)[*Mono*](http://en.wikipedia.org/wiki/Mono_%28software%29) *runtime. In theory, this eliminates the need to distribute separate binaries for different platforms and CPU types. CIL code is verified for safety during runtime, providing better security and reliability than natively compiled binaries.*

*The execution process looks like this:*

1. *Source code is converted to Common Intermediate Language, CLI's equivalent to Assembly language for a CPU.*
2. *CIL is then assembled into bytecode and a* [*.NET assembly*](http://en.wikipedia.org/wiki/.NET_assembly) *is created.*
3. *Upon execution of a .NET assembly, its bytecode is passed through the runtime's JIT compiler to generate native code. (Ahead-of-time compilation eliminates this step at run time.)*
4. *The native code is executed by the computer's processor.*” [18]

**Compare MSIL to other language:**

|  |  |
| --- | --- |
| Java code: | MSIL: |
| static void Main(string[] args){outer: for (int i = 2; i < 1000; i++) { for (int j = 2; j < i; j++) { if (i % j == 0) goto outer; } Console.WriteLine(i); }} | . .method private hidebysig static void Main(string[] args) cil managed { .entrypoint .maxstack 2 .locals init (int32 V\_0, int32 V\_1) IL\_0000: ldc.i4.2 stloc.0 br.s IL\_001f IL\_0004: ldc.i4.2 stloc.1 br.s IL\_0011 IL\_0008: ldloc.0 ldloc.1 rem brfalse.s IL\_0000 ldloc.1 ldc.i4.1 add stloc.1 IL\_0011: ldloc.1 ldloc.0 blt.s IL\_0008 ldloc.0 call void [mscorlib]System.Console::WriteLine(int32) ldloc.0 ldc.i4.1 add stloc.0 IL\_001f: ldloc.0 ldc.i4 0x3e8 blt.s IL\_0004 ret } |
| C# source code | MSIL: |
| int i = 10;if(i!=20)i = i\*20;Console.WriteLine(i); | IL\_0000: ldc.i4.s 10IL\_0002: stloc.0IL\_0003: ldloc.0IL\_0004: ldc.i4.s 20IL\_0006: beq.s IL\_000dIL\_0008: ldloc.0IL\_0009: ldc.i4.s 20IL\_000b: mulIL\_000c: stloc.0IL\_000d: ldloc.0IL\_000e: call void [mscorlib]System.Console::WriteLine(int32) |

* + 1. **Constraints Solver**

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**.

“*Pex faithfully encodes all constraints arising in safe .NET programs such that Z3can 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.*”[19]

**Generating Integers and Floats:**

“*Pex'* [*constraint solver*](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Constraint%20Solver.html) *determines* [*test input values*](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Input%20Generation.html) *of primitives types such as byte, int, float, ... in order to trigger different execution paths of the test and the program-under-test. For integers, the constraint solver can reason about all arithmetic operations, treating integers as 32-bit or 64-bit values. Bitwise manipulations are supported as well. For floating point number, the constraint solver approximates the behavior of linear arithmetic with rational numbers. Enums are treated similar to their underlying integer type.*” [20]

**Z3**

“*Z3 is a* [*Satisfiability Modulo Theories (SMT) solver*](http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories)*. That is, it is an automated satisfiability checker for many sorted (i.e., typed) first-order logic with built-in theories, including support for quantifiers.*” [21]

* 1. **Processes of testing code**

This section will provide two simple test code examples to explain how the program handles the test cod step by step.

**Test\_1:**

|  |
| --- |
| DoWork (int x) { int j=0;　　if ( x==4 )　　 j=42; else j=56; return j; } |

|  |
| --- |
| .method private hidebysig instance int32  DoWork(int32 x) cil managed { .maxstack 2  .locals init ([0] int32 j,  [1] int32 CS$1$0000,  [2] bool CS$4$0001)  IL\_0000: nop Stack:[] x:67 l0: l1: l2: IL\_0001: ldc.i4.0 Stack:[0] x:67 l0: l1: l2: IL\_0002: stloc.0 Stack:[] x:67 l0:0 l1: l2: IL\_0003: ldarg.1 Stack:[x] x:67 l0:0 l1: l2: IL\_0004: ldc.i4.4 Stack:[4,x] x:67 l0:0 l1: l2: IL\_0005: ceq Stack:[ if(4==x) -> 1 else 0 ] x:67 l0:0 l1: l2: IL\_0007: ldc.i4.0 Stack:[0, if(4==x) -> 1 else 0] x:67 l0:0 l1: l2: IL\_0008: ceq Stack:[ if[0 == (4==x->1 else 0)]->1 else 0] x:67 l0:0 l1: l2: IL\_000a: stloc.2 Stack:[] x:67 l0:0 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_000b: ldloc.2 Stack:[ if [0 == (4==x->1 else 0) ]->1 else 0] x:67 l0:0 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_000c: brtrue.s IL\_0013 Stack:[ if [0 == (4==x->1 else 0) ]->1 else 0] x:67 l0:0 l1: l2: 0 == (4==x->1 else 0)->1 else 0 if [0 == (4==x->1 else 0) ]->1 branch to IL\_0013.  IL\_000e: ldc.i4.s 42 Stack:[ 42 ] x:67 l0:0 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_0010: stloc.0 Stack:[ ] x:67 l0:42 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_0011: br.s IL\_0016 Branch to target IL\_0016 IL\_0013: ldc.i4.s 56 Stack:[ 56 ] x:67 L0:0 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_0015: stloc.0 Stack:[] x:67 L0:56 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0. IL\_0016: ldloc.0 if( [0 == (4==x->1 else 0) ]->1):Stack:[56] x:67 l0:56 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 If ([0 == (4==x->1 else 0) ]->0):Stack:[42] x:67 l0:42 l1: l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_0017: stloc.1 if([0 == (4==x->1 else 0) ]->1):Stack:[] x:67 l0:56 l1:56 l2: if [0 == (4==x->1 else 0)]->1 else 0If([0 == (4==x->1 else 0) ]->0):Stack:[] x:67 l0:42 l1:42 l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_0018: br.s IL\_001a Branch to target IL\_001a IL\_001a: ldloc.1 if ([0 == (4==x->1 else 0) ]->1):Stack:[56] x:67 l0:56 l1:56 l2: if [0 == (4==x->1 else 0)]->1 else 0 If([0 == (4==x->1 else 0) ]->0):Stack:[42] x:67 l0:42 l1:42 l2: if [0 == (4==x->1 else 0)]->1 else 0 IL\_001b: ret Return from method, possibly with a value.} // end of method Class1::DoWork |

* When the code has been created in Visual Studio, user can test the code use our project program. It also create MSIL, like the above provide, **Reflector tool** [22] can be use to check the MSIL code in text file.
* In order to run the code for the first time, program needs to supply some argument values to **Complicated**—for example, *x=67*(arbitrary values).
* Program execute DoWork( 67 )
	+ - 1. **J=0**, program will push 0 onto stack, and pop it from stack into local variable 0.

( IL\_0001: ldc.i4.0 )

* + - 1. And then load argument X onto stack. For example x=67.

(IL\_0003: ldarg.1)

* + - 1. Because the condition **if (x==4)**, value 4 will be push onto the stack.

(IL\_0004: ldc.i4.4)

* + - 1. Compare 2 elements in the stack, 67!=4, so push 0 onto the stack. (IL\_0005: ceq )
			2. And then still push 0 onto stack, and compare them. Because 0=0. So, 1 will be pushed onto the stack.

(IL\_0007: ldc.i4.0)

* + - 1. The execution will jump to other code.

(IL\_000c: brtrue.s IL\_0013)

* + - 1. Push 56 onto the stack

(IL\_0013: ldc.i4.s 56)

* + - 1. Pop 56 from stack into local variable 0.

(IL\_0015: stloc.0)

* + - 1. Load local variable 0 onto stack, 56 local onto stack again.

(IL\_0016: ldloc.0)

* + - 1. Pop 56 from stack into local variable 1.

(IL\_0017: stloc.1)

* + - 1. Load local variable 1 onto stack, 56 local onto stack again.

(IL\_001a: ldloc.1)

* Program use CLRMonitor monitors the execution of the code, following the execution into and out of method calls.
* During the process of execution, the CLRMonitor will tell you which instruction is executed
* After Monotint execution of DoWork(67)
* Path condition:

Random create input value: x=67;

The path (0== (4==x-->1 else 0) -->1 else 0) -->1 has been coverage.

Now the program needs to create a new x input in order to coverage the other path, such as:

 (0== (4==x-->1 else 0) -->1 else 0)-->0;

x=4

This process is done by **contain solver**.

And then program execute DoWork(4);

* Because all statements have been covered now, program will stop.

****

**Test2:**

|  |
| --- |
| Int Complicated (intx,inty){If(x==Obfuscate(y))ThrownewRareException();return0;}Int Obfuscate(inty){ return(100+y)\*567%2347;} |

**Start 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=572andy=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≠ (100+y)\* 567%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 ≠(100+y) \* 567mod2347. So, in order reach the other branch of the if-statement, Pex builds the negated condition, x=(100+y) \* 567mod2347, 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) \* 567mod2347 and y = 152.

This time, the throw statement will be covered.

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

* 1. **Available programming language**
	2. **JAVA**
		1. **Description of Java:**

“*Java is a* [*programming language*](http://en.wikipedia.org/wiki/Programming_language) *originally developed by* [*James Gosling*](http://en.wikipedia.org/wiki/James_Gosling) *at* [*Sun Microsystems*](http://en.wikipedia.org/wiki/Sun_Microsystems) *(which is now a* [*subsidiary*](http://en.wikipedia.org/wiki/Subsidiary) *of* [*Oracle Corporation*](http://en.wikipedia.org/wiki/Oracle_Corporation)*) and released in 1995 as a core component of Sun Microsystems'* [*Java platform*](http://en.wikipedia.org/wiki/Java_%28software_platform%29)*. The language derives much of its* [*syntax*](http://en.wikipedia.org/wiki/Syntax_%28programming_languages%29) *from* [*C*](http://en.wikipedia.org/wiki/C_%28programming_language%29) *and* [*C++*](http://en.wikipedia.org/wiki/C%2B%2B) *but has a simpler* [*object model*](http://en.wikipedia.org/wiki/Object_model) *and fewer low-level facilities. Java applications are typically* [*compiled*](http://en.wikipedia.org/wiki/Compiler) *to* [*bytecode*](http://en.wikipedia.org/wiki/Java_bytecode) *(*[*class file*](http://en.wikipedia.org/wiki/Class_%28file_format%29)*) that can run on any* [*Java Virtual Machine*](http://en.wikipedia.org/wiki/Java_Virtual_Machine) *(JVM) regardless of* [*computer architecture*](http://en.wikipedia.org/wiki/Computer_architecture)*. Java is general-purpose, concurrent, class-based, and object-oriented, and is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere*". [23]

Java is an object-oriented programming language. It is security and dynamic, multi-thread. Java is simple, because its syntax comes from C and C++. It is running in Java Virtual Machine. It will not contact to user’s personal operating system, so it is multi-system.

* + 1. **Discusses about JAVA**

JAVA was designed as an object-oriented language, it has similar syntax and grammars as C++, on that base simplify the complicated algorithm and structure, more efficient and easy to grasp for programmer. Because the characteristics of JAVA, it can multi-system support and “Write once, run anywhere”, diverse and powerful development tools to support, Security and High Portability. As those advantages of JAVA, many applications designed using JAVA.

Slow, as we all know, Java programs running depends on the Java Virtual Machine, so compared to written by other languages (C, C + +) programs, JAVA programs are slowly, because it is not direct implement machine code. Because Java takes into account on cross-platform, so it does not like the language (Example: C) as closer to the operating system, also can not deal with the underlying operating system. However, this program has to translate the code to MSIL and then output the result; it also has to use .net Profiling API, so it is difficult to bring into play advantages of JAVA.

* 1. **C#**
		1. **Description of C#:**

“C# (pronounced "see sharp") is a [multi-paradigm programming language](http://en.wikipedia.org/wiki/Multi-paradigm_programming_language) encompassing [imperative](http://en.wikipedia.org/wiki/Imperative_programming), [declarative](http://en.wikipedia.org/wiki/Declarative_programming), [functional](http://en.wikipedia.org/wiki/Functional_programming), [generic](http://en.wikipedia.org/wiki/Generic_programming), [object-oriented](http://en.wikipedia.org/wiki/Object-oriented_programming), and [component-oriented](http://en.wikipedia.org/wiki/Component-based_software_engineering) programming disciplines. It was developed by [Microsoft](http://en.wikipedia.org/wiki/Microsoft) within the [.NET](http://en.wikipedia.org/wiki/.NET_Framework) initiative and later approved as a standard by [Ecma](http://en.wikipedia.org/wiki/Ecma_International) and [ISO](http://en.wikipedia.org/wiki/International_Organization_for_Standardization). C# is one of the programming languages designed for the [Common Language Infrastructure](http://en.wikipedia.org/wiki/Common_Language_Infrastructure). C# is intended to be a simple, modern, general-purpose, object-oriented programming language. Its development team is led by [Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg).

By design, C# is the programming language that most directly reflects the underlying [Common Language Infrastructure](http://en.wikipedia.org/wiki/Common_Language_Infrastructure) (CLI). Most of its intrinsic types correspond to value-types implemented by the CLI framework. However, the language specification does not state the code generation requirements of the compiler: that is, it does not state that a C# compiler must target a Common Language Runtime, or generate [Common Intermediate Language](http://en.wikipedia.org/wiki/Common_Intermediate_Language) (CIL), or generate any other specific format. Theoretically, a C# compiler could generate machine code like traditional compilers of C++ or FORTRAN.” [24]

1. **Available Complier**
	1. **Netbean**

“*The NetBeans Platform allows applications to be developed from a set of modular software components called modules. Applications based on the NetBeans platform (including the NetBeans IDE) can be extended by third party developers. NetBeans began in 1996 as Xelfi (word play on Delphi), a Java IDE student project under the guidance of the Faculty of Mathematics and Physics at Charles University in Prague. Sun open-sourced the NetBeans IDE in June of the following year. The NetBeans community has since continued to grow, thanks to individuals and companies using and contributing to the project.*” [25]

NetBeans is designed for software developers a free and open IDE (integrated development environment), we can get many of the tools what we need form there, including the establishment of desktop applications, enterprise-class applications, WEB development and JAVA mobile application development, C / C + +, or Ruby. NetBeans can be very easily installed at a variety of operating platforms, including Windows, Linux, Mac OS, and Solaris operating systems.

* 1. **Visual Studio**

**What is Visual Studio?**

“Microsoft Visual Studio is an [integrated development environment](http://en.wikipedia.org/wiki/Integrated_development_environment) (IDE) from [Microsoft](http://en.wikipedia.org/wiki/Microsoft). It can be used to develop [console](http://en.wikipedia.org/wiki/Console_application) and [graphical user interface](http://en.wikipedia.org/wiki/Graphical_user_interface) [applications](http://en.wikipedia.org/wiki/Application_software) along with [Windows Forms](http://en.wikipedia.org/wiki/Windows_Forms) applications, [web sites](http://en.wikipedia.org/wiki/Web_site), [web applications](http://en.wikipedia.org/wiki/Web_application), and [web services](http://en.wikipedia.org/wiki/Web_service) in both [native code](http://en.wikipedia.org/wiki/Native_code) together with [managed code](http://en.wikipedia.org/wiki/Managed_code) for all platforms supported by [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows), [Windows Mobile](http://en.wikipedia.org/wiki/Windows_Mobile), [Windows CE](http://en.wikipedia.org/wiki/Windows_CE), [.NET Framework](http://en.wikipedia.org/wiki/.NET_Framework), [.NET Compact Framework](http://en.wikipedia.org/wiki/.NET_Compact_Framework) and [Microsoft Silverlight](http://en.wikipedia.org/wiki/Microsoft_Silverlight). Visual Studio includes a [code editor](http://en.wikipedia.org/wiki/Code_editor) supporting [IntelliSense](http://en.wikipedia.org/wiki/IntelliSense) as well as [code refactoring](http://en.wikipedia.org/wiki/Code_refactoring). The integrated [debugger](http://en.wikipedia.org/wiki/Microsoft_Visual_Studio_Debugger) works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building [GUI](http://en.wikipedia.org/wiki/GUI) applications, web designer, [class](http://en.wikipedia.org/wiki/Class_%28computing%29) designer, and [database schema](http://en.wikipedia.org/wiki/Database_schema) designer. It accepts plug-ins that enhance the functionality at almost every level—including adding support for [source-control](http://en.wikipedia.org/wiki/Source_control) systems (like [Subversion](http://en.wikipedia.org/wiki/Subversion_%28software%29) and [Visual SourceSafe](http://en.wikipedia.org/wiki/Visual_SourceSafe)) and adding new toolsets like editors and visual designers for [domain-specific languages](http://en.wikipedia.org/wiki/Domain-specific_language) or toolsets for other aspects of the [software development lifecycle](http://en.wikipedia.org/wiki/Software_development_lifecycle).” [26]

**Why use Visual Studio to programming this project?**

“Visual Studio supports different [programming languages](http://en.wikipedia.org/wiki/Programming_language) by means of language services, which allow the code editor and debugger to support (to varying degrees) nearly any [programming language](http://en.wikipedia.org/wiki/Programming_language), provided a language-specific service exists. Built-in languages include [C](http://en.wikipedia.org/wiki/C_%28programming_language%29)/[C++](http://en.wikipedia.org/wiki/C%2B%2B) (via [Visual C++](http://en.wikipedia.org/wiki/Visual_C%2B%2B)), [VB.NET](http://en.wikipedia.org/wiki/VB.NET) (via [Visual Basic .NET](http://en.wikipedia.org/wiki/Visual_Basic_.NET)), [C#](http://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29) (via [Visual C#](http://en.wikipedia.org/wiki/Visual_C_Sharp)), and [F#](http://en.wikipedia.org/wiki/F_Sharp_%28programming_language%29) (as of Visual Studio 2010[[2]](http://en.wikipedia.org/wiki/Visual_Studio#cite_note-1)). Support for other languages such as [M](http://en.wikipedia.org/wiki/M_%28programming_language%29), [Python](http://en.wikipedia.org/wiki/IronPython), and [Ruby](http://en.wikipedia.org/wiki/IronRuby) among others is available via language services installed separately. It also supports [XML](http://en.wikipedia.org/wiki/XML)/[XSLT](http://en.wikipedia.org/wiki/XSLT), [HTML](http://en.wikipedia.org/wiki/HTML)/[XHTML](http://en.wikipedia.org/wiki/XHTML), [JavaScript](http://en.wikipedia.org/wiki/JavaScript) and [CSS](http://en.wikipedia.org/wiki/Cascading_Style_Sheets). Individual language-specific versions of Visual Studio also exist which provide more limited language services to the user: Microsoft Visual Basic, Visual J#, Visual C#, and Visual C++.” [27]

1. **Conclusion**

This document is making the research on necessary information of project, summarize the above sections:

* Make a brief representation on what this project is, simple analysis on demand and its goal.
* Analysis a similar application: Pex, discuss how pex work by overview diagram, provide a simple example for test, brief analysis project techniques:
	+ - **Parameterized unit testing**: 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.
		- **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.
		- **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.
		- **.NET Profiling API:** Pex monitors the execution of a .NET program through code instrumentation. Pex plugs into the .NET profiling API.
		- **MSIL techniques:** The ClrMonitor is an unmanaged COM component that implements the [CLR profiling API](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/CLR%20profiling%20API.html). “It uses this API mainly to rewrite [MSIL](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/MSIL.html) instructions of selected methods, types or assemblies on-the-fly at JIT time. It can redirect calls and insert callbacks in the [MSIL](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/MSIL.html) instruction sequence
		- **Constraints Solver:** For each chosen unexplored branch, Pex builds a formula that represents the condition under which this branch can be reached
* Provide two simple tests to discuss the processes of the code. It include how to translate the code to MSIL code, how Constraints Solver work
* Discuss possible compiling language, such as: JAVA, C#....
* Discuss possible Complier, such as: Netbean, Visual Studio…

**References:**

[01]: Pex`s definition

<http://www.pexforfun.com/Page.aspx#3fb3105f-905b-4b5d-a368-d1924a9d1e71>

[02]: Pex Code example:

<http://www.pexforfun.com/Page.aspx#3fb3105f-905b-4b5d-a368-d1924a9d1e71>

[03]: Pex example

<http://www.pexforfun.com/Page.aspx#3fb3105f-905b-4b5d-a368-d1924a9d1e71>

[04]: Pex`s overview diagram

<http://www.pexforfun.com/Documentation.aspx>

[04]: Pex example, Parameterized Unit Testing with Microsoft Pex, Page 3 <http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>,

[05]: Parameterized Unit Testing with Microsoft Pex, Page4 <http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>,

[06]:<< Automating Software Testing Using Program Analysis>>, Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Page 33, 34 <http://research.microsoft.com/en-us/um/people/pg/public_psfiles/ieeesw2008.pdf>

[07]: Symbolic Execution, Parameterized Unit Testing with Microsoft Pex, Page 5

<http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>

[08]: Dynamic Symbolic Execution, Parameterized Unit Testing with Microsoft Pex, Page 6

<http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>

[09]: Parameterized Unit Testing with Microsoft Pex, Page 8 <http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>

[10]: .NET profiling API, Parameterized Unit Testing with Microsoft Pex, Page 10

<http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>

[11]: [Pex - Automated Whitebox Testing for .NET](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/WebHome.html),

<http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/microsoft.extendedreflection.clrmonitor.html>

[12]:[Microsoft.ExtendedReflection.](http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Microsoft.ExtendedReflection.ClrMonitor.html)

<http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Microsoft.ExtendedReflection.html>

[13]: ClrMonitor <http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/microsoft.extendedreflection.clrmonitor.html>

[14]: Rewrite MSIL Code on the Fly with the .NET Framework Profiling API

<http://msdn.microsoft.com/en-us/magazine/cc188743.aspx>

[15]: CLR,

<http://en.wikipedia.org/wiki/Common_Language_Runtime>

[16]: COM,

<http://www.microsoft.com/com/default.mspx>

[17]: Monitoring.Controller

<http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Microsoft.ExtendedReflection.html>]

[18]: Wikipedia, MSIL <http://en.wikipedia.org/wiki/Common_Intermediate_Language>

[19]: Constraint Solving, Parameterized Unit Testing with Microsoft Pex, Page 14

<http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf>

[20]: Generating Integers and Floats <http://research.microsoft.com/en-us/um/redmond/projects/pex/wiki/Generating%20Integers%20and%20Floats.html>

[21]: Constraints Solver, Z3 <http://research.microsoft.com/en-us/um/redmond/projects/z3/intro.html>

[22]: Reflector tool:

<http://en.wikipedia.org/wiki/.NET_Reflector>

[23]: Wikipedia, Java

<http://en.wikipedia.org/wiki/JAVA>

[24]: Wikipedia, C#

[http://en.wikipedia.org/wiki/C#](http://en.wikipedia.org/wiki/C)

[25]: Wikipedia, Netbean

[http://en.wikipedia.org/wiki/**Netbean**](http://en.wikipedia.org/wiki/Netbean)

[26]: Wikipedia, Visual Studio

<http://en.wikipedia.org/wiki/Visual_Studio>

[27]: Wikipedia, Visual Studio

http://en.wikipedia.org/wiki/Visual\_Studio