PeX - Parameterized Unit Tests in Visual Studio

Michael Ziller
University of Karlsruhe
Formal Software Development Seminar

8th July 2008

1 Overview

The following seminar paper is dealing with the recently released new test tool PeX by Microsoft Reseach, which stands short for Program Exploration. The main idea behind it is based on Parameterized Unit Tests, a concept trying to get along with some issues evolving from standard unit tests by supporting the developer with the possibility to let PeX automatically explore the possible execution paths of the code under test and examine the results against a self written specification of the expected behaviour. This white-box testing approach often results in a high code coverage and can help find issues with unhandled exceptions or missed corner cases in the range of possible inputs.

The paper is structured as follows: Section 2 shows some of the issues that could arise using classic Unit Tests and tries to motivate solutions while Section 3 introduces the concepts of Parameterized Unit Tests in more detail. The now introduced ideas and concepts leave some degrees of freedom, so Section 4 takes a closer look how these parts are realized in PeX. Since the tool is fully integrated into Microsoft Visual Studio as a plug-in, Section 5 examines further how this took place, and gives a small example how a developer is able to find corner cases of his code under test.

Since the first announcements and releasing of small insights to the project, including several presentations and papers, there seems to be a lot of misunderstandings concerning the question what PeX is and what it is not, and the variety of reactions range from excitement to the reproach of just being another clone of another already existing test tool, so to enlight this situation, Section 6 takes a closer look at related tools, projects and ideas.

Finally Section 7 deals with possible future research and ideas how to evolve the existing version of PeX.

Used throughout this paper is one theoretical scenario to take a closer look on different aspects of the ideas and concepts of PUTs and PeX to get a bigger picture how useful they can be.

```
Informations about the customers, their balance,
open orders, rebates etc. are stored in an extern
database system. Users transfer money into their
account first before they can start buying stuff from
the shop. Several shipping options with different
costs are available.
```

Web Shop: Basics

2 Motivation

Since software systems become more and more complex, the need to very carefully test the produced code for bugs, missing exception handling or just to be sure if the implemented software really meets the intended specification becomes more and more critical. Several approaches in methodology today are build around the fact that testing is vital to software development, for instance Test Driven Development (TDD), where the tests are written before the code itself is implemented. The most common way to test parts of software during the development is to write suitable test suites including given test inputs from the domain. This approach has lead to a lot of frameworks and test tools which simplify the task of writing such Unit Tests called dummy methods and to execute them. Especially the possibility to evolve growing repeatable test suites for whole parts of a software system offers an easy way to determine how changes to the code affect the systems behaviour, for instance during maintenance.

Figure now the following constraint in the Web Shop scenario when a user wants to finish the order at the check-out:

```
When a user commits a new order its value is not
allowed to be greater than the user's balance.
```

Web Shop: Ordering Items

The decision if the user's balance satisfy this constraint depends on a lot of factors, including the shipping option, the items weight, different taxes for states, rebates etc. resulting in many different possible input combinations. So how to decide which inputs should be used for a classic unit test suite? Since we could expect a lot of branches, loops etc. in such an implementation it is reasonable to select the inputs in a way that regards each possible behaviour respectively execution path.

But what happens if the implementation is changed during a review or maintenance work? Is the set of unit tests in the old suite still testing every possible execution path?

Although high code coverage alone is not a proof of high quality and perfect software and often hard to achieve if even possible, it is a reasonable goal to test as much possible inputs as necessary but not too few.

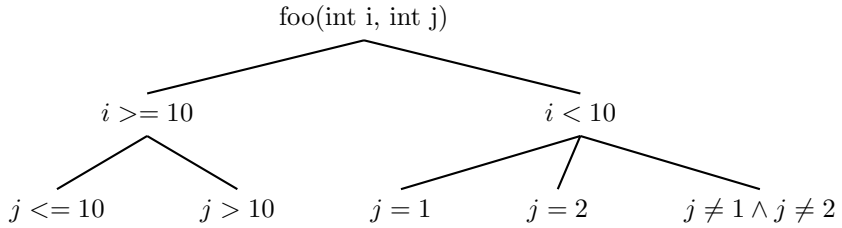
3 Parameterized Unit Tests - Abstract

In the former section we have seen some open questions concerning the selection of suitable inputs for unit tests. Parameterized Unit Tests try to deal with these issue by automatized exploration of the code under test by generating constraints on how inputs should look like. This is achieved by introducing symbolic variables for every variable in the code under test and then constructing symbolic expressions over them for every branch encountered during the exploration. An IF-Statement for instance introduces two symbolic expressions, one for the IF-Part and one for the ELSE-Part. The resulting structure is similar to a decision tree and each path from the root to a leaf represents a possible execution path.

```
public void foo(int i, int j) {
    if (i >= 10) {
        block A;
        if (j <= 10) { block B;
        }
        else { block C;
        }
    }
    else {
        switch (j) {
            case 1:
                block D;
                break;
            case 2:
                block E;
                break;
            default:
                block F
                break;
        }
    }
}
```

Example Code

The following figure shows a simple decision tree for the small code example above including several IF-Statements and a SWITCH-Statement.



Each path from the root to a leaf corresponds to one execution path in the method, so conjoining every symbolic expression on this path leads to a condition which determines what inputs are needed to take this execution path.

So this Symbolic Execution called apporach evaluates a set of possible inputs which cover each execution path. How would a unit test process a single input from this suite? It would take several objects like the customer, the order etc. and then execute the methods under test and examine the results with respect to the constraints from the specification. The following small code example shows how such a test could look like:

```
public void TestBalance(customer c, order o) {
    Assume.IsTrue(c.ID != 0); ** precondition
    Assume.IsTrue(o.Value > 0);
    int balanceBefore = c.GetBalance();
    if (CommitOrder(c,o)) { ** postcondition commit
        Assert.IsTrue(c.GetBalance() >= 0);
        Assert.IsTrue(c.GetBalance() == balanceBefore -
            o.Value);
    }
    else { **postcondition commit failed
        Assert.IsTrue(c.GetBalance == balanceBefore);
    }
}
```

Web Shop: Ordering Items

The Unit Test first checks if the customer has a valid ID and if the order has a value greater than zero. It temporarily stores the customers balance and then calls the method bool CommitOrder(customer c, order o) and dependent from its return value checks if the booking of the order has been correct.

Since this procedure is not directly interlocked with the exploration of the code under test it is possible to seperate two concerns: The specification of the expected behaviour and the search for reasonable test inputs. A specification of

behaviour as seen in the example before has still to be written by the user, but the search for a suitable set of test inputs can be done automatized by Symbolic Execution as demonstrated in this chapter.

On the other hand some open question arise with this approach, for instance it is not always possible to explore each execution path since their number must not be finite due to loops, recursion and so on. Also the question how deep the symbolic execution should examine the code under test, for instance if the methods in the Unit Test themselves call other methods dependent on its inputs:

```
public bool CommitOrder(customer c, order o) {
    ...
    if (o.Shipping == ShippingOption.Express) {
        o.ShippingCosts = CalculateShipCosts(c.State);
    }
    ...
}
```

Method CommitOrder

Dependent from the selected shipping option the method might call another method parameterized with its own input. Should this other method also be explored during Symbolic Execution? These questions leave some degrees of freedom how to exactly implement Parameterized Unit Tests in a tool and the next section shows how PeX is handling these cases.

4 Parameterized Unit Tests - PeX

As seen before, the realization of Parameterized Unit Tests into a tool requires some basic decision on how to implement the degrees of freedom. One is concerning the search strategy on the code under test during Symbolic Execution. The developers of PeX chose an approach similar to a depth-first-search with backtracking: The input variables are first instantiated with default values and the unit test is executed with these parameters.

While executing, it examines the branches taken and remembering which path has been taken including the condition for the alternative path. After this run has been finished, it manipulates the variables in a way to take another execution path and again, monitoring each branch and alternative condition it passes during the next execution. This dynamic approach to Symbolic Execution often results in a good branch coverage.

The other degree of freedom which has been introduced in the former section is related to the exploration depth during symbolic execution. By default, PeX is only examining the methods direclty called from the unit test, but if other

method calls occur, which is very likely, PeX notifies the user in its output, and the user can decide which of those methods can be ignored or which methods should be also explored in another test run.

5 Integration in Visual Studio 2008

In this section, there is a short introduction how PeX is fully integrated into Microsoft's IDE Visual Studio 2008 and some information about how it is simplifying some of the test tasks. Figure 1: PexMethod demonstrate how the add-in knows which methods are intended to be testet and explored, a simple right-click and selecting the "Run Pex Exploration" is enough to start a test.

```
[PexMethod]
public string ClassifyInteger(int i) {
    if (i == 0) { return "Power of two"; }
    if (i % 2 != 0) { return "Uneven"; }
    while (i > 1)
    {
        if (i % 2 != 0) { return "Even"; }
        i = i / 2;
    }
    return "Power of two";
}
```

Figure 1: PexMethod

In this case, it is a small simple method to test whether a given integer is uneven, even or a power of two. A look at the resulting table (Figure 2) for the test demonstrate how PeX has explored the method under test: It first instantiated the input variable with the default for integer, zero. Since the first branch tested if the input is zero, this path had been taken in the first run, and after it finished, PeX manipulated the variable to integer not zero, one. Again, this has lead to another execution path and another unexplored branch.

What else becomes clear from the results is that there is one corner case missing in the implementation: Every even negative integer is claimed as power of two! We simply forgot to take negative integers in account.

By introducing a test for negative integers and throwing an ArgumentException if one occurs, we can fix that behaviour. The thrown ArgumentException in the next test run is causing the test to fail, since the unit test is not expecting an exception of this type. In this way, it is possible to explore and correct the

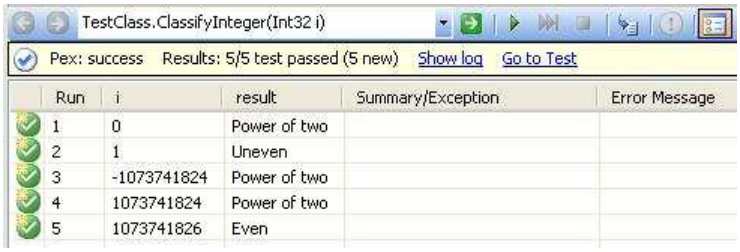


Figure 2: PexResult

implemented code iteratively, pointing the user to unexpected behaviour.

Also it is possible to let PeX directly know which methods should never be instrumented, like complete .NET framework packages or other functionality, so that they don't have to be deselected in every new test run.

There are also several other concepts supported by the PeX add-in for Visual Studio 2008, like mock objects to simulate expected behaviour of not yet implemented functionality. The interesting part about the PexMock objects is that they use reflection to explore those mock objects in a way somehow similar to the normal code exploration, and also take those in account for generating tests. The resulting mock objects are called Parameterized Mock Objects. More information about this technique can be found in [5].

6 Related Tools

Since there are a lot of different test tools and frameworks available, it is clear that every new released test tool should be carefully compared in its functionality, useability and more to be in the position to decide which tool fits best for a team, for a project etc.

Some of the concepts introduced in PeX look on the first glance similar to some already available functions of existing and wide used test tools. Especially at the beginning there seems to be a lot of misunderstandings what PeX is exactly doing, [6] compared the "White-Box-Test Generating Inputs" with "RowTest" from [7], but in contrast to a Data-Driven-Set, there is no need to fill a set with own test inputs, PeX itself is exploring the code under test for reasonable inputs.

A similar situation about mock objects, there are several free tools available like [9], but like Data-Driven-Set tests, one has to manually instrument the mock objects how to behave and react while PeX Parameterized Mock Objects are explored and used in a fashion like generating test inputs. Again, more

information about Parameterized Mock Objects can be found in [5].

The concept of Parameterized Unit Tests itself has already been used for another programming paradigm, for the functional programming language Haskell: Like in PeX, QuickCheck[8] seperates the specification from test inputs, and tests every reasonable and possible input against the code under test.

The way PeX is exploring the path conditions during a test run is path-bounded model-checking, and hence also related to other model-checking tools.

7 Conclusion & Future Work

PeX is using the concepts of Parameterized Unit Tests to simplify the task to find reasonable test inputs which result in high code coverage and finding corner cases of the code under test. It is fully integrated into Microsoft Visual Studio 2008. In [1] the developers describe how they were able to even find bugs in an well-tested, long in use part of the .NET framework.

It is also a reasonable tool to test unknown code, and see how this code under test behaves, see what other methods it is instrumenting and so on. Further interesting parts are the possibility to use Parameterized Mock Objects to set up an environment for testing bigger parts with a lot of interaction.

Future work for the project includes the possibility to use PeX on multi-threaded code, use information about the structure of the system to simplify the task of write method sequences for test methods.

So far, PeX is released as version 0.5, and it is still more an experimental approach from Microsoft Research to examine how the use of Parameterized Unit Tests and Mock Objects can help during software development.

References

1. Tillmann, N., Schulte, W.: Parameterized unit tests. Proceedings of the 10th European software engineering conference, pp. 263-272. ACM Press, New York (2005)
2. Tillmann, N.: Nikolai Tillman's Blog. http://blogs.msdn.com/nikolait/
3. de Halleux, J.: Peli's Farm. http://blog.dotnetwiki.org/
4. Tillmann, N., de Halleux, J.: Pex - White Box Test Generation for .NET. Proceedings of the 2nd International Tests and Proofs Conference, pp. 134-153. Springer Verlag (2008)

5. de Halleux, J., Tillmann, N., Schulte, W.: Microsoft Pex Tutorial. http://research.microsoft.com/pex/articles/pextutorial.pdf

6. Simser, B.: Pex - A Tool in Search of an Identity. http://weblogs.asp.net/bsimser/archive/2008/02/05/pex-a-tool-in-search-of-an-identity.aspx

7. NUnit development team. NUnit. http://www.nunit.org/

8. QuickCheck development team. QuickCheck. http://www.cs.chalmers.se/ rjmh/QuickCheck/

9. Rhino Mock development team. Rhino Mocks. http://www.ayende.com/Blog/archive/2007/03/28/Rhino-Mocks-3.0-Released.aspx