Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger

Nikolai Tillmann
Microsoft Research
Redmond, WA, USA
nikolait@microsoft.com

Jonathan de Halleux
Microsoft Research
Redmond, WA, USA
jhalleux@microsoft.com

Tao Xie
University of Illinois at Urbana-Champaign
Urbana, IL, USA
taoxie@illinois.edu

ABSTRACT

Producing industry impacts has been an important, yet challenging task for the research community. In this paper, we report experiences on successful technology transfer of Pex and its relatives (tools derived from or associated with Pex) from Microsoft Research and lessons learned from more than eight years of research efforts by the Pex team in collaboration with academia. Moles, a tool associated with Pex, was shipped as Fakes with Visual Studio since August 2012, benefiting a huge user base of Visual Studio around the world. The number of download counts of Pex and its lightweight version called Code Digger has reached tens of thousands within one or two years. Pex4Fun (derived from Pex), an educational gaming website released since June 2010, has achieved high educational impacts, reflected by the number of clicks of the "Ask Pex!" button (indicating the attempts made by users to solve games in Pex4Fun) as over 1.5 million till July 2014. Evolved from Pex4Fun, the Code Hunt website has been used in a very large programming competition. In this paper, we discuss the technology background, tool overview, impacts, project timeline, and lessons learned from the project. We hope that our reported experiences can inspire more high-impact technology-transfer research from the research community.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

General Terms

Languages, Experimentation

Keywords

Testing; Symbolic execution; Technology transfer

1. INTRODUCTION

Producing industry impacts (such as producing successful technology transfer and adoption) has often been an important task for academic or industrial researchers when conducting research. However, it is generally challenging to transfer research results into industrial practices. In recent years, a set of promising research results around automated software engineering tools have been produced by the research community and demonstrated to be useful on various real-world open source projects. There are substantial opportunities for exploiting these research results to improve industrial practices of automated software engineering. Furthermore, there are substantial demands from software practitioners to address urgent and critical issues in their industrial practices.

However, demonstrating effectiveness of proposed research tools on open source code or even industrial code (as often reported in the research literature) does not naturally lead to successful transfer or adoption of the tools in industrial practices. The research community [27, 48, 78] has already realized gaps between academic research and industrial practices, and has called for training and education of researchers and practitioners in conducting successful technology transfer and adoption. Furthermore, there have been various cases of successful technology transfer on static bug finding [23, 81], dynamic bug finding [26], and software analytics [31, 32, 46, 76, 77].

In this paper, we report experiences on successful technology transfer of Pex [63], denoting the (**P**)rogram (**Ex**)ploration tool, and its relatives (tools derived from or associated with Pex) from Microsoft Research and lessons learned from more than eight years of research efforts by the Pex team in collaboration with academia. Moles [34], a tool associated with Pex, was shipped as Fakes with Visual Studio since August 2012, benefiting a huge user base of Visual Studio around the world. The number of download counts of Pex and its lightweight version called Code Digger [15] has reached tens of thousands within one or two years. Pex4Fun [66, 74] (derived from Pex), an educational gaming website released since June 2010, has achieved high educational impacts, reflected by the number of clicks of the "Ask Pex!" button (indicating the attempts made by users to solve games in Pex4Fun) as over 1.5 million till July 2014. Evolved from Pex4Fun, the Code Hunt [62, 64] website has been used in a very large programming competition.

The rest of the paper is organized as follows. Section 2 presents the technology background for the Pex project. Section 3 presents the overview of Pex and its relatives. Section 4 presents the industry, educational, and research impacts of the Pex project. Section 5 illustrates the project timeline. Section 6 discusses the lessons learned and Section 7 concludes.

2. TECHNOLOGY BACKGROUND

In this section, we present the technology background underlying the Pex project: Dynamic Symbolic Execution (Section 2.1), which is the technology that Pex has realized; parameterized unit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642941.

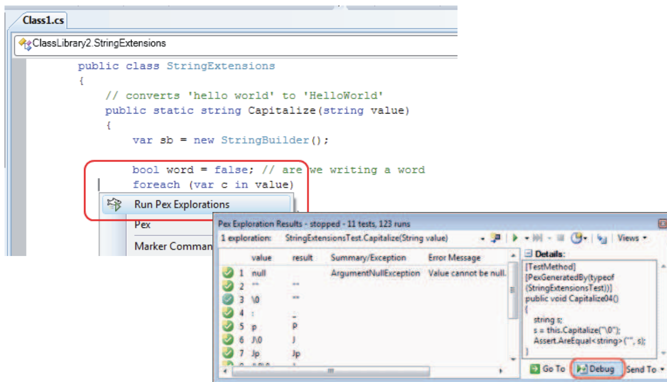


Figure 1: User interface of applying Pex

testing (Section 2.2), which is the technology that provides a playground and motivation for Pex; fitness-guided path exploration (Section 2.3), which is a key technology that allows Pex to improve its effectiveness and efficiency.

2.1 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) [38, 54] is a variation of symbolic execution [29,43] and leverages runtime information from concrete executions. DSE is often conducted in iterations to systematically increase code coverage such as block or branch coverage. In each iteration, DSE executes the program under test with a test input, which can be a default or randomly generated input in the first iteration or an input generated in one of the previous iterations. During the execution of the program under test, DSE performs symbolic execution in parallel to collect symbolic constraints on program inputs obtained from predicates in branch statements along the execution. The conjunction of all symbolic constraints along an executed path is called the path condition. Then DSE flips a branching node in the executed path to construct a new path that shares the prefix to the node with the executed path, but then deviates and takes a different branch. DSE relies on a constraint solver such as Z3 [35] to (1) check whether such a flipped path is feasible; if so, (2) compute a satisfying assignment — such assignment forms a new test input whose execution will follow along the flipped path.

2.2 Parameterized Unit Testing

A key methodology that Pex supports is parameterized unit testing [67], which extends previous industry practice based on closed, traditional unit tests (i.e., unit test methods without input parameters). In parameterized unit testing, unit test methods are generalized by allowing parameters to form parameterized unit tests. This generalization serves two main purposes. First, parameterized unit tests are specifications of the behavior of the methods under test: not only exemplary arguments to the methods under test, but also ranges of such arguments. Second, parameterized unit tests describe a set of traditional unit tests that can be obtained by instantiating the methods of the parameterized unit tests with given argument-value sets. An automatic test generation tool such as Pex can be used to generate argument-value sets for parameterized unit tests.

2.3 Fitness-Guided Path Exploration

DSE, or symbolic execution in general, suffers from the path exploration problem: there are typically a huge or infinite number of paths in the code under test for the DSE engine to explore. To tackle this exploration problem, the Pex team developed the Fitnex search strategy [75] for guiding the DSE engine (Pex) to achieve the target test coverage quickly. The guided search provided by the Fitnex strategy alleviates issues encountered by previous DSE search strategies such as (bounded) exhaustive search [38, 54] or random search [28]. In particular, the Fitnex strategy assigns to already explored paths *fitness values* computed by program-derived fitness functions. (Fitness functions have been traditionally used in search-based test generation [47].) A *fitness function* measures how close an explored path is in achieving the target test coverage. A *fitness gain* is also measured for each explored branch: a higher fitness gain is given to a branch if flipping a branching node for the branch in the past helped achieve better fitness values. Then during path exploration, the Fitnex strategy prefers to flip a branching node whose corresponding branch has a *higher fitness gain* in a previously explored path with a *better fitness value*. The Pex team integrated the Fitnex strategy with other fair-choice search strategies [75], which work well for other types of exploration problems. Such integration of the Fitnex and other strategies achieves the effect of getting the best of both in practice.

3. PEX AND ITS RELATIVES

In this section, we present the overview of Pex (Section 3.1) and its relatives, i.e., tools that are derived from or associated with Pex, including Code Digger (Section 3.2), Moles/Fakes (Section 3.3), and Pex4Fun/Code Hunt (Section 3.4).

3.1 Pex

Based on DSE, Pex [63] is an automatic white-box test generation tool for .NET., which has been integrated into Microsoft Visual Studio as an add-in. Starting from a method that takes parameters (either a parameterized unit test or a method under test), Pex performs path-bounded model checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along different execution paths, following the idea of DSE. Pex uses the theorem prover and constraint solver Z3 [35] to reason about the feasibility of execution paths, and to obtain ground models for constraint systems (Pex was one of the major reasons for initiating the Z3 project). The unit tests that Pex generates can be executed by various unit test frameworks (such as MSTest, NUnit, and xUnit) without Pex. Figure 1 shows the user interface of applying Pex in Visual Studio. Pex can be invoked anywhere in the Visual Studio code editor (via clicking the context menu item of "Run Pex Explorations") and does not require any pre-existing tests. The results are presented as an input-output table as shown in the lower-right part of Figure 1. Each row represents a set of inputs to the method under test and the observed return value or exception. The results can be easily saved in the Visual Studio solution as traditional unit tests. The generated tests can also be debugged just like any other unit tests.

3.2 Code Digger

Code Digger [15], the first Visual Studio extension from the Pex team for Visual Studio 2012, generates test data that show different behaviors of the .NET code under test. Figure 2 shows the user interface of applying Code Digger in Visual Studio. Code Digger can be invoked through clicking the context menu item "Generate Inputs / Outputs Table" in the Visual Studio code editor. The generation result is a table showing for which inputs the code under test produces which outputs, as shown in the lower-left part of Figure 2. The table helps the developers understand the behavior of the code, and it may also uncover hidden faults. Under the hood, Code Digger uses Pex to systematically explore paths in the code, trying to generate a test suite that achieves high code coverage.

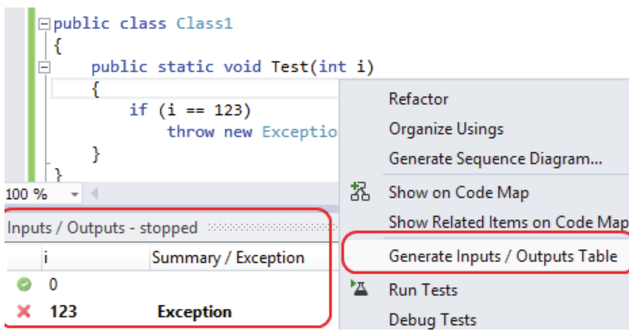


Figure 2: User interface of applying Code Digger

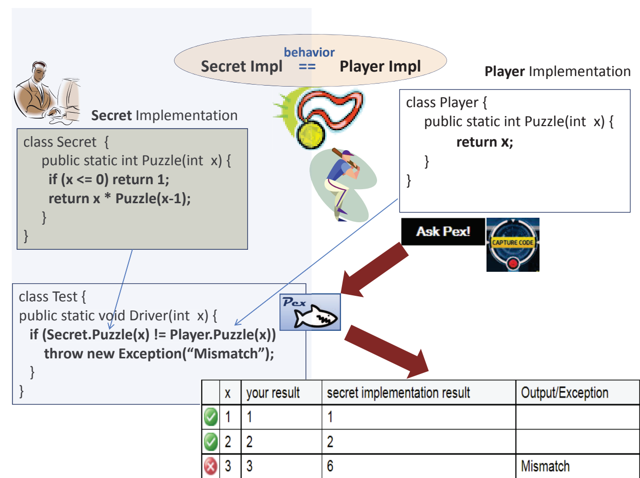
Out of the box, Code Digger works on only public .NET code that resides in Portable Class Libraries. But options are available to allow the developers to configure Code Digger to explore other .NET projects. By restricting the code exploration to Portable Class Libraries, Code Digger avoids problems with code that has dependencies on a particular platform that the Pex engine does not understand (Portable Class Libraries are a neat way in .NET to ensure that there are no external dependencies). In addition, when using Code Digger, developers do not need to change their project under test (e.g., without the need of creating a new test project, as needed in Pex); therefore, Code Digger can be conveniently used by some developers in a project team even when some other developers in the same project team do not use Code Digger.

3.3 Moles/Fakes

In software testing especially unit testing, it is often desirable to test individual units (such as classes) in isolation. In other words, all irrelevant environment dependencies should be mocked or simulated, so that the unit tests for the code under test run quickly and give deterministic results. Ideally, to address the problem, developers can refactor the code under test by introducing explicit interface boundaries and allowing different interface implementations. However, in practice, it is often not feasible to refactor existing legacy code under test, especially the code from a third party.

To address the issue, the Pex team developed a new lightweight framework called Moles [34]. The Moles framework allows the test code to provide alternative implementations for non-abstract methods of any .NET type. Using code instrumentation, the Moles framework redirects calls of a method to its alternative implementation. The Moles framework has been designed to work together with Pex to enable automated test generation. Pex can analyze only .NET managed code. Consequently, when the program execution invokes a method (e.g., a system-library method) not implemented in .NET managed code, Pex cannot infer the constraints that reflect the behavior of that method, and thus Pex cannot generate test data to achieve high code coverage. Moles allows developers to replace any method at test time with .NET code that simulates its behavior, which in turn facilitates test generation with Pex.

Achieving high industry impacts (see Section 4.1 for details), Moles has been shipped with Visual Studio (since Visual Studio 2012) with the new name of Microsoft Fakes [17]. Similar to Moles, Fakes comes in two flavors: a stub and a shim. A stub replaces a class with a small substitute that implements the same interface. To use stubs, developers have to design their application so that each component (e.g., a class or group of classes designed and updated together and typically contained in an assembly) depends only on interfaces, and not on other components. A shim modifies the compiled code of developers' application at run time so that instead of making a specified method call, it runs the shim code that



	x	your result	secret implementation result	Output/Exception
✓	1	1	1	
✓	2	2	2	
✗	3	3	6	Mismatch

Figure 3: Workflow of playing an example coding duel in Pex4Fun or Code Hunt

Figure 4: User interface of playing a coding duel in Code Hunt

the developers' test provides. Shims can be used to replace calls to assemblies (such as .NET assemblies) that the developers cannot modify.

3.4 Pex4Fun/Code Hunt

Pex4Fun [66, 74] (`http://www.pexforfun.com/`) is an interactive gaming platform for teaching and learning programming and software engineering (supporting .NET programming languages such as C#, Visual Basic, and F#). It is a browser-based teaching and learning environment [24] with target users as teachers, students, and even software practitioners, etc. The core type of Pex4Fun games is a *coding duel* [65] where the player has to solve a particular programming problem. Figure 3 shows the workflow of playing an example coding duel in Pex4Fun or Code Hunt [62, 64] (evolved from Pex4Fun, with more details in the end of this section). A coding duel created by a game creator (who could be any user of Pex4Fun) consists of two methods with the same method signature and return type[1]. One of these two methods is the secret (golden) implementation (shown in the top-left part of Figure 3), which is not visible to the player. The other is the player implementation (shown in the top-right part of Figure 3), which is visible to the player and can be an empty implementation or a faulty implementation of the secret implementation. The player imple-

[1]The method signature of a coding duel must have at least one input parameter. The return type of a coding duel must not be void.

mentation can include optional comments to give the player some hints in order to reduce the difficulty level of gaming.

After a player selects a coding-duel game to play, the player's winning goal is to modify the player implementation (visible to the player) to make its behavior (in terms of the method inputs and results) to be the same as the secret implementation (not visible to the player). Apparently, without any feedback or help, the player has no way to guess how the secret implementation would behave. The player can get some feedback by clicking the button "Ask Pex" (or "Capture Code" in Code Hunt) to request the following two types of feedback: (1) under what sample method input(s) the player implementation and the secret implementation have the same method result and (2) under what sample method input(s) the player implementation and the secret implementation have different method results.

Pex4Fun applies Pex on the synthesized test driver code (for comparing the returns of the two implementations, as shown in the lower-left part of Figure 3) to generate such feedback (as shown in the lower-right part of Figure 3) and determine whether the player wins the game: the player wins the game if Pex cannot generate any method input to cause the player implementation and the secret implementation to have different method results.

Evolved from Pex4Fun, Code Hunt [62, 64] (`http://www.codehunt.com/`) instills more fun and entertaining effects, adds hint generation, adds language support to Java, etc. There are four steps to follow in playing games in Code Hunt. In Step 1, from the suggested sequence of sectors in the game, the player discovers a secret code segment by selecting a sector. In Step 2, after the player clicks a sector, the player is presented with the player's code, as shown in the left part of Figure 4. Then the player clicks the "Capture Code" button (as shown in the top-middle part of Figure 4, equivalent to the "Ask Pex" button in Pex4Fun) to analyze the behavioral differences of the secret code segment and the player's code. Then Code Hunt displays the feedback on the behavioral differences (as shown in the right part of Figure 4). In Step 3, based on the feedback, the player then modifies the player's code to match the secret code segment's behavior. Then the player iterates through Steps 2 and 3 until no behavioral differences of the secret code segment and the player's code can be found by Code Hunt. In this case, the player reaches Step 4, winning the game.

4. IMPACTS

In this section, we present the impacts that the Pex project has made in the industry community (Section 4.1), the education community (Section 4.2), and the research community (Section 4.3).

4.1 Industry Impacts

Inside Microsoft, Pex was applied to a core component of the .NET architecture, which had already been extensively tested over five years by approximately 40 testers within Microsoft. The component is the basis for other libraries, which are used by thousands of developers and millions of end users. Pex found various issues in this core component, including a serious issue. Furthermore, there have been many other successful cases for applying Pex on Microsoft code bases.

In the broad software industry, Pex has had high download counts in industry and academia, e.g., 30,388 during a 20-month period of February 2008 - October 2009, including 17,366 downloads with the academic license and 13,022 downloads with the Devlabs industrial license. The release of Pex (including Moles) has led to an active user community (including members largely from the industry). From the period of October 2008 till November 2010, the MSDN forum dedicated to Pex and Moles included more than 1,400 forum posts made by the user community.

Microsoft Fakes was shipped with the Ultimate editions of Visual Studio since Visual Studio 2012, benefiting a huge user base of Visual Studio around the world. The substantial benefits and impacts of Microsoft Fakes were also reflected by the Visual Studio user community's request to "Provide Microsoft Fakes with all Visual Studio editions". The request post states "Include Microsoft Fakes with all editions of Visual Studio including Professional edition rather than being restricted to developers with Ultimate editions of Visual Studio. This will allow all Visual Studio developers to leverage the capabilities of the Fakes mocking library since mocking/stubbing is a necessary part of every developer's Unit Testing toolbelt." This request got 1,457 votes from the Visual Studio user community.

Since Code Digger [15] was shipped to the Visual Studio Gallery in April 2013, the number of download counts of Code Digger is 31,165 (as of July 2014). There have been many very positive user comments on Code Digger. Some quotes are below:

- "Very easy to use and quickly see the potential input sanitation problems!"
- "What an awesome tool.. Help us to explore our logic by providing accurate input parameter for each logic branch.. You should try this as one of your ultimate tool :) It really saves a lot of our time to explore every logic branch in our apps.."
- "Great tool to generate unit tests for parameter boundary tests. I like to see it integrated into Visual Studio and the testing features as far as in ReSharper! :)"
- "What a fantastic tool. Whilst it's not bullet proof, it shows amazing promise. I ran the Code Digger over a number of real-world methods and it immediately identified dozens of edge cases we hadn't thought of. This is getting rolled-out to my team TODAY! Well done. Brilliant. Really brilliant."
- "Top stuff here. Very anxious for more of the Pex features that were available in VS 2010 Pex & Moles (like auto-gen unit tests). This tool is poised to become indispensable for anyone writing solid suites of unit tests."

4.2 Educational Impacts

Pex4Fun [66, 74] has been gaining high popularity in the community: since it was released to the public in June 2010, the number of clicks of the "Ask Pex!" button (indicating the attempts made by users to solve games in Pex4Fun) has reached over 1.5 million (1,544,979) as of July 28, 2014. Pex4Fun has provided a number of open virtual courses (similar to MOOCs in spirit) including learning materials along with games used to reinforce students' learning.

In May 2011, Microsoft Research hosted a contest on solving coding duels [8] at the 2011 International Conference on Software Engineering (ICSE 2011). During the main ICSE program, conference attendees could register a nickname in Pex4Fun and complete as many coding duels as possible within the ICSE 2011 main conference period. Whoever solved the most coding duels by the end of the period won the contest. The ICSE 2011 coding-duel contest received 7,000 Pex4Fun attempts, 450 duels completed, and 28 participants (though likely more, since some did not actually enter the official ICSE 2011 course to play the coding duels designed for the contest).

Recently, Pex4Fun inspired the new contest form [22] in the 2013 ICFP Programming Contest [7]. Competing entirely over the Internet, more than 300 participating teams of programmers from around the world were asked to complete a series of programming

tasks, using any programming languages and tools that they desired, to address an extremely challenging scenario in program synthesis. Results were assessed using Microsoft Research's Z3 [35] running in Windows Azure to compare submitted solutions to actual solutions to determine correctness, in a similar way as coding duels in Pex4Fun. The generic problem description was "guess the implementation of a black-box function implemented in a simple functional language through querying a web server for information about the input/output behavior of the function." Over the contest's 72 hours, Z3 received about a million requests and successfully decided all, except about 300 problem instances, within an imposed time limit of 20 seconds, the overwhelming majority within a matter of a few milliseconds.

Various Pex4Fun users posted their comments on the Internet to express their enthusiasm and interest (even addiction) to Pex4Fun [74]. Here we included some examples. "PEX4fun could become a better FizzBuzz than FizzBuzz.", "it really got me *excited*. The part that got me most is about spreading interest in/teaching CS: I do think that it's REALLY great for teaching | learning!", "Frankly this is my favorite game. I used to love the first person shooters and the satisfaction of blowing away a whole team of Noobies playing Rainbow Six, but this is far more fun.", "Teaching, learning - isn't this really the same, in the end? In fact, for me personally, it's really about leveraging curiosity, be it mine or someone else's - at best both! And PexForFun (+ all the stuff behind) is a great, promising platform for this: you got riddles, you got competition, you get feedback that makes you think ahead...", "I'm afraid I'll have to constrain myself to spend just an hour or so a day on this really exciting stuff, as I'm really stuffed with work", "PexForFun improves greatly over projecteuler w.r.t. how proposed solutions are verified; in fact what it adds is that you don't just get a 'nope' but something more articulate, something you can build on. That's what I think is really great and exciting - let's push it even further now!"

Evolved from Pex4Fun, Code Hunt [62, 64] offers more fun and entertaining effects, hints for players, language support for Java besides C#. In April 2014, Code Hunt was used at a very large competition called Beauty of Programming in the Greater China Region. In three rounds, 2,353 students scored in the game, with an average 55.7% puzzles solved across this large number. Code Hunt is being offered for more competitions, as ongoing efforts.

4.3 Research Impacts

The Pex team has published a set of papers on Pex or its relatives along with experimental extensions of Pex [20]. Work on experimental extensions of Pex was mostly conducted in collaboration with the Automated Software Engineering research group led by the third author at North Carolina State University (before July 2013) and then at the University of Illinois at Urbana-Champaign (since July 2013).

As of July 2014, the major publication on Pex [63] published in 2008 has got 464 citations. The major publication on parameterized unit testing [67] published in 2005 has got 172 citations. The major publication on the fitness-guided path exploration [75] published in 2009 has got 102 citations. All the preceding citation counts were derived from Google Scholar in July 2014.

Pex itself is not open source and only some extensions of Pex are open source [19] (including extensions [75] being part of Pex releases and other extensions [30, 53, 58, 60, 61, 72, 79] as research exploration). However, Pex has been used by other third-party researchers from academia (who did not have direct collaboration with the Pex team on carrying out the academic research) in different ways. First, some researchers (e.g., [49–52, 80]) made extensions of Pex by leveraging the APIs of Pex (without requiring to access the Pex source code). Second, some researchers (e.g., [33, 37, 56]) compared Pex (as a stand-alone state-of-the-art test generation tool) with their new approaches in empirical evaluations.

5. PROJECT TIMELINE

The initial idea of Pex arose from the Spec Explorer project [70], which produced a model-based testing tool with an embedded model checker for (a subset of) .NET. At the core of Spec Explorer was XRT [40], a self-contained runtime of (a subset of) .NET that enabled the dynamic analysis and exploration of model programs written in (a subset of) .NET. While it is possible to apply this engine not just on specially written model programs, but also on carefully chosen real-world .NET code that stays in the supported .NET subset [67, 68], it became also clear that it would be a very laborious task to extend the self-contained runtime XRT to support the full, or at least a very broad, subset of the .NET runtime. The required work would have clearly exceeded the manpower of the research team. It was under this consideration that the Pex project was started based on the idea of instrumenting, monitoring, and guiding execution paths in the commercial .NET runtime. We next list the timeline of the Pex project.

In 2005, Pex was started by the Pex team (at Microsoft Research), which initially included only the first author. Then in September 2006, the second author joined the Pex team.

In March 2007, the Pex team blogged about the Pex tool to the public [2].

In May 2008, the first version of Pex (under a Microsoft Research License) was released to the public. It was integrated with Visual Studio 2008 Professional.

In September 2008, Pex integrated the Fitnex [75] search strategy in combination with other fair-choice strategies (the source code of Fitnex was also released as open source [18]).

In October 2008, an early version of Code Digger was released for supporting Visual Studio 2008 and Visual Studio 2010. Code Digger allows to start the path exploration from the code under test, not from an already existing unit test or even parameterized unit test.

In October 2008, Pex became one of the first three DevLabs projects [11], and available as a Microsoft Download for Visual Studio 2010 Community Technology Preview (CTP). The new Microsoft Download comes under a Microsoft Pre-Release Software License for Pex. This license is different from the Microsoft Research License agreement: this license does not explicitly rule out all forms of commercial use. Note that some of DevLabs projects may turn into features in Microsoft's existing shipping products; some may be open-sourced by Microsoft to the community.

In October 2008, Stubs, a simple framework for .NET test stubs was released with Pex.

In May 2009, Pex included a new feature, *Unit Tests as Inputs*. Before using heavy-weight constraint solving to explore hard-to-reach execution paths, Pex can leverage already existing unit tests that call parameterized unit tests: Pex scans their body to extract the parameter values, and then Pex uses these values to seed the exploration. (In the past, Pex would have seeded the exploration by simply using the default values for all parameters, and nothing else.)

In September 2009, Moles [34], a lightweight detour framework, was released with Pex. Moles is a new extension of the Stubs framework: it lets developers replace any .NET method (including static methods) with their own delegate.

In December 2009, an extension of Stubs to write models, was released with Pex. With stubs and moles, the Pex team provided

a framework to write record/replay tests (i.e., mock-based tests) against any .NET type, interface or not. The extension further allows developers to write simple implementations with behavior that can be used to replace the external system during testing. The benefit of using the extension is that the resulting test cases are much more robust to code changes since the tests specify the state of the system, rather than a sequence of method calls and outcomes.

In January 2010, the Stubs framework was renamed to the Moles framework. The Pex team decided to make the Moles the center of the framework and as a consequence, renamed "Stubs" to "Moles".

In March 2010, after more than a year of community feedback on DevLabs, Pex and Moles took the next step and had become Visual Studio 2010 Power Tools.

In April 2010, a Facebook page on Pex and Moles [5] was launched to better build its user community.

In May 2010, Moles was released as a standalone tool on Visual Studio Gallery [13].

In June 2010, Pex integrated Rex [71], being smarter about regular expressions.

In June 2010, Pex4Fun was announced, being a web site that brings code to life. Pex4Fun on the web was a radically simplified version of the fully featured Pex Power Tool for Visual Studio. The website came with a set of code puzzles, where a player's task is to simply guess what a given Puzzle method does, by studying the code of the Puzzle method.

In July 2010, the game type of coding duels, being interactive puzzles, was introduced in Pex4Fun. Coding duels are different from those simple puzzles announced in June 2010.

In August 2012, Visual Studio 2012 was released, including Microsoft Fakes, which was evolved from the Moles framework. The Microsoft Visual Studio product team took over and owned the code base of Moles/Fakes, relieving the Pex team from subsequent maintenance efforts of the code base.

In April 2013, Code Digger, an extension for Visual Studio 2012, was shipped to the Visual Studio Gallery [15]. After shipping the Moles framework as Fakes in Visual Studio 2012, this effort on Code Digger was the subsequent step of bringing the Pex project to the latest and greatest development environment. Since then, ongoing efforts have been taken for further technology transfer of Code Digger/Pex.

In February 2014, Code Hunt, evolved from Pex4Fun, was released to the public [4] and formally announced in May 2014 [3]. It is a significant extension of Pex4Fun by instilling more fun and entertaining effects, adding hint generation, adding language support to Java, etc.

6. LESSONS LEARNED IN TOOL TRANSFER

In this section, we illustrate main lessons learned from the Pex project in terms of technology transfer and adoption.

6.1 Evolving "Dreams"

Tool researchers typically would like their "dreams" of tool adoption to become true. However, turning "dreams" to be true can be a long process (or even may not be possible at times) and tool researchers would need to evolve their dreams by being adaptive to evolve their focus to different types of tools or variants of their initial tools.

As discussed in the project timeline (Section 5), before the Pex project was started, the first author, along with some other researchers at Microsoft Research, worked on the Spec Explorer project [70], a model-based testing tool with an embedded model checker for (a subset of) .NET. Although the Spec Explorer project led to substantial adoption in the Microsoft's Protocol Documentation program [39], the Spec Explorer tool did not reach wide adoption by practitioners. During the promotion of adopting the Spec Explorer tool (e.g., within Microsoft), it was observed that systematic training of test engineers played a critical role in their adoption of model-based testing methodologies or tools [39]. The adoption of the Spec Explorer tool at Microsoft was driven by having more experienced test engineers to mentor less experienced ones; however, experienced test engineers might not be commonly available, thus becoming a bottleneck in the tool-adoption process.

After observing the difficulties of "shaping" the target tool users (e.g., via training), the first author moved on to propose and focus on a more lightweight formal-testing methodology, parameterized unit testing [67, 68], which relies on specifications written in unit tests to provide test oracles, and relies on an automatic white-box test generation tool to provide test data. At this timing, the Pex project was born (in 2005). Although the Pex team and the third author had great passion and high hope in promoting parameterized unit testing (in combination with automatic test generation provided by Pex) in practice, such methodology seemed to get well accepted and adopted by only a relatively small subset of advanced developers. A large portion of developers did not seem to popularly write parameterized unit tests, likely due to higher abstract thinking skills required when writing specifications encoded in parameterized unit tests.

Then the Pex team relaxed the usage of Pex to be beyond the usage scenario of requiring parameterized unit tests before running Pex: developers can simply run Pex even when they do not write any parameterized unit tests. Such new mode of Pex along with Code Digger was provided to attract more developers as tool users.

Furthermore, after observing the strong needs of tool support for mocking (independent of whether Pex or another automatic test-generation tool is used), the Pex team further invested efforts to develop Moles [34], a lightweight detour framework. Only some time after that point, in August 2012, Moles was shipped as Fakes with Visual Studio, benefiting a huge user base of Visual Studio around the world (being the first such shipped tool from the Pex project).

Although Moles has relatively simple technologies behind the scene, interestingly its timing of being shipped with Visual Studio was earlier than other more sophisticated tools such as Pex from the Pex project. In addition, Pex4Fun and Code Hunt, educational websites based on Pex, also got high educational impacts. However, in the future, Pex itself may get as wide adoption as (or even wider adoption than) Moles/Fakes or Pex4Fun/Code Hunt.

6.2 Dealing with "Chicken and Egg" Problem

In the Pex project setting, there are two types of tool transfer and adoption: (1) convincing the target users such as developers in Microsoft product teams or developers in the broad .NET user community to use Pex or its relatives; (2) convincing target tool vendors such as the Microsoft Visual Studio product team to ship parts of Pex or its relatives with future releases of Visual Studio.

Typically, the second type of tool transfer and adoption would succeed only after the first type has succeeded, because the target tool vendors would like to see a large enough user base of the tools before committing to ship such tools in their future product releases. Tools with public releases already typically will first be shipped at Microsoft DevLabs [11] or Microsoft Visual Studio Gallery [14] for some time before being considered for being shipped with Visual Studio.

In fact, in early phases of transferring a tool, accomplishing the first type of tool transfer and adoption is quite challenging. The *chicken and egg problem* occurred when developers were faced with the decision on adopting Pex in early phases. Below are example conversations that occurred in early stages of tool adoption between a developer or project manager and the Pex team:

- Developer/manager: "Who is using your tool?"
- Pex team: "Do you want to be the first?"
- Developer/manager: "I love your tool but no."

Developers or project managers in industry tend to be conservative and might wait and see more substantial success stories of using a tool before they commit to adopt the tool. It is generally difficult to recruit the first early tool-adopters but having success stories of these early tool-adopters could help convince others to follow the adoption.

To address such challenges, the Pex team was persistent in tackling real-world challenges, finding early adopters, addressing technical and non-technical barriers for technology adoption in industry (e.g., providing a tool license that does not prohibit commercial use), etc. In addition, the Pex team adopted the strategy of incremental shipping, e.g., shipping Code Digger before aiming to ship the full-fledge Pex.

To convince target users (e.g., developers from industry) to become early adopters of the Pex tool, the Pex team selected and demonstrated how Pex can be applied on complex real-world classes such as the ResourceReader class [1] from the .NET framework. It is important to go beyond simple textbook classes (such as simple well-isolated data structures as often used in testing research papers) because the target users might have a pre-impression that a research tool might not work well on real-world cases.

To convince the Microsoft Visual Studio product team to ship Moles as Fakes in Visual Studio, the Pex team selected and demonstrated successful and important scenarios of applying Moles such as assisting unit testing of SharePoint code. Among Microsoft Patterns & Practices[2], the SharePoint Guidance [12] highlighted the use of Moles for testing SharePoint code. Successfully addressing pain points of testing such important type of code provided strong incentives for the Visual Studio product team to ship Fakes. In general, it is very important to demonstrate that the tool to be transferred can be applied to address one important scenario very well: the more important the scenario is and the better the scenario is addressed, the stronger case it is to convince the target tool vendors to ship the tool.

In addition, to make a strong case, it is important to provide quantitative information (for reflecting the tool's importance or benefit extent) to the target tool vendors. It is a common practice of releasing a tool as an installation file via the Internet and keeping track of the download counts, without keeping track of which organization's members downloaded the tool or whether/how the downloaded tool was used. However, based on our experiences, there are three main issues with such common practice. First, a member of an organization (e.g., a developer from a company) can download the tool's installation file and share the file with other members of the organization. Thus, counting only one download in this case does not faithfully reflect the reality. Second, those who downloaded or installed the tool do not necessarily use the tool in their work (in a regular basis). Thus, it is important to incorporate a mechanism of collecting tool usage information (such as those in the Microsoft Customer Experience Improvement Program [10]). Third, not all tool downloads or usages are equal: downloads or usages by members of important customer companies are valued more by the target tool vendors than downloads or usages by others. Thus, it is important to identify the organization affiliations of those who downloaded or used the tool.

[2]Microsoft Patterns & Practices provide popular recommendations on designing and developing custom applications using the Microsoft platform.

6.3 Considering Human Factors

In the research literature, there was little discussion on human consumption of the generated tests (user studies on test generation tools conducted by Fraser et al. [36] briefly touched on such issue). Based on substantial user feedback on Pex, it is important to generate user-friendly tests for users to digest and interact with.

As a past anecdotic example, a user of Code Digger asked a question on string values generated by Code Digger previously [15]: "Code digger generates a lot of "\0" strings as input. I can't find a way to create such a string via my own C# code. Could any one show me a C# snippet? I meant zero terminated string." The Pex team responded "In C#, a \0 in a string does not mean zero-termination. It's just yet another character in the string (a very simple character where all bits are zero), and you can create as Pex shows the value: "\0"."

As another past anecdotic example, below are conversations between a developer and the Pex team on desired representative values for generated test inputs by Pex:

- Developer: "Your tool generated "\0""
- Pex team: "What did you expect?"
- Developer: "Marc."

More recently, Pex (inheritably along with Code Digger) has been improved with a number of techniques to take human factors into account. For example, the latest version of Pex tries to use human-friendly characters and integers. Pex now prefers human-readable characters over strange Unicode characters, if both cover the same execution path. Similarly, Pex now prefers small integers that do not cause overflows over big integers that cause overflows, if both cover the same execution path.

To accomplish these techniques, Pex modifies some outputs of Z3 instead of taking Z3's outputs directly as test data. Pex includes a wrapper around Z3 to give additional guidance to Z3 in its constraint solving. For example, Pex feeds Z3 the original constraints from path exploration along with additional basic range constraints such integers being between 0-100; if no solution is provided by Z3, then Pex falls back to the default setting without additional guidance. In some other cases, Pex starts with small test data and iteratively queries Z3 to see whether the provided small test data (such as the array length being a small value) can be the solution to the original constraints; if not, Pex increases the value of the test data and queries Z3 again.

Because users need to interact with the generated tests, even the naming convention of the generated tests would matter to the users. Below are conversations between a developer and the Pex team on desired naming of generated tests by Pex:

- Developer: "Your tool generated a test called Foo001. I don't like it."
- Pex team: "What did you expect?"
- Developer: "Foo_Should_Fail_When_The_Bar_Is_Negative."

When developers write traditional test methods manually, they use meaningful naming conventions for these test methods. It is natural for developers to expect to see meaningful naming for generated test methods (especially when no parameterized unit tests are written, and the generated test methods are used for robustness checking as directly inspected by the developers). In response to user feedback like the above, the Pex team improved the naming

convention in various ways, e.g., when a test throws an exception, 'Throws' + ExceptionTypeName is added to the name of the generated test method.

Pex incorporates a number of techniques to minimize the users' inspection or investigation efforts. Pex includes techniques to produce fewer "Object Creation" messages that are not actually relevant to better engage the users to cooperate with Pex to address those challenges faced by Pex [72]. Pex includes techniques to suppress warnings that are hard to understand (they are still shown in the diagnostic mode) or do not matter. Pex also allows users to manually suppress "Object Creation" messages that are not relevant in the users' context by clicking the "ignore" button. Pex also provides convenient views such as the Global Event View, the Exploration Tree View, and the Exception Tree View to enable users to conveniently navigate through the path-exploration results.

Pex incorporates various convenient mechanisms to seek feedback from users. For example, the Pex team added the thumbs-up and thumbs-down buttons along with a link on "Send Feedback" (for asking questions in the Pex forum) in the Pex result view.

6.4 Performing Well on Best-Case Scenarios While Averagely on Worst-Case Scenarios

In general, a proposed technique, such as a search strategy, may often have its best-case application scenarios and its worst-case scenarios. For example, most previous approaches to DSE [38,54] typically use a fixed "depth-first, backtracking" search strategy, where the next execution path would always share the longest possible prefix with the previous execution path. Therefore, much resource may be allocated to analyze small parts of the program under test before moving on. The worst-case scenarios for such search strategy would be code including a loop with its loop bound dependent on program inputs, because such search strategy would try to unfold the same loop forever.

A patch to such search strategy (as often adopted by previous approaches to DSE) is to impose a fixed bound on the number of iterations of a loop. But then the worst-case scenarios for such search strategy would be code including branches whose coverage would require the number of iterations of a loop to be beyond the imposed fixed bound. Some other well-known search strategies, such as breadth-first search, do not get stuck in the same way as depth-first search, but it does not take into account the structure of the program, having its own worst-case scenarios.

To allow Pex to be applied generally well on a variety of real-world code, the Pex team could not afford to adopt any of the above-mentioned search strategies, which typically work quite well on their best-case scenarios but quite poorly on their worst-case scenarios. To address such issues, before September 2008, Pex preferred a fair choice between all such unexplored branches of the explored execution tree. Pex included various fair strategies, which partition all branches into equivalence classes, and then picked a representative of the least often chosen class. The equivalence classes clustered branches by mapping them according to different criteria.

In September 2008, Pex combined the new Fitnex [75] strategy with its above-mentioned fair-choice strategy. Such combination in fact enables to perform well on best-case scenarios while performing averagely on worst-case scenarios. The best-case scenarios for Fitnex include code whose coverage is amenable to fitness functions, and the worst-case scenarios include code whose coverage is not amenable to fitness functions. For Fitnex's worst-case scenarios, the search-strategy combination can make Pex to perform very similarly to its fair-choice strategy.

Although such search-strategy combination does not read like a significant portion of our proposed work in the published paper [75] on Fitnex, the combination design is very critical to enable the integration of Fitnex to Pex releases. To our best knowledge, there was no or little discussion on such desirable characteristic of "performing well on best-case scenarios while averagely on worst-case scenarios" for a proposed new technique in the research literature. However, such desirable characteristic is a precondition for a technique to be integrated into a tool aiming for adoption in practice.

6.5 Dealing with Tool Users' Stereotypical Mindset or Habits

Tool users may have specific stereotypical mindsets. In fact, different subgroups of tool users may have different stereotypical mindsets. For example, one stereotypical mindset towards an automated test generation tool is that using such tool would be simply one mouse click and then everything would work just perfectly. A developer with such stereotypical mindset could easily get frustrated and give up in continuing using a tool when the tool faces challenges in dealing with the real-world code base that the developer intends to apply the tool on.

Therefore, setting realistic expectations right away is very important [73] when introducing a tool such as Pex to the target users, e.g., developers. Although it is important to illustrate the potential benefits of the tool to the target users, the tool will typically have limitations, and these limitations must be clearly communicated to the target users and how the target users can deal with such limitations. In other words, the target users should be informed and trained on helping the tool to deal with those faced limitations in a cooperative way, as advocated in the cooperative testing methodology [72]. For example, training the target users to know how to use Moles or Fakes to isolate environment dependencies is very important to enable successful application of Pex on real-world code bases, which tend to have many environment dependencies.

Note that tool users often may not realize the hidden complexity of the code under test brought by the invocation of some third-party API method and then blame the tool for the achieved low code coverage [73]. Below is an example code snippet under test that a developer applied Pex on and complained about achieved low code coverage of:

```
void Sum(int[] numbers) {
      string sum = "0";
      foreach(int number in numbers) {
        sum = (int.Parse(sum) + number).ToString()
      }
      if (sum == "123")
          throw new BugException();
}
```

In fact, invoking simple API methods such as `int.Parse` and `int.ToString` can incur challenges for Pex or any other test generation tool, because the implementation code of these API methods can be complex, including a huge or infinite number of paths for Pex to explore.

In addition, some tool users may have a stereotypical mindset on thinking or expecting that a test generation tool (capable of achieving high code coverage) would detect all or most kinds of faults in the code under test. It is also important to emphasize the limited kinds of faults (e.g., crashing or uncaught exceptions) that can be detected by the tool when the users do not write any specifications (such as properties in production code under test or parameterized unit tests) to reflect the intended behavior of the code under test. For example, it was observed that a developer complained "Your tool only finds null references." When that developer was asked whether he wrote any assertions, he answered with "Assertion???".

Some tool users may have existing habits on sticking to particular development styles such as Test-Driven Development (TDD). Then convincing them to change their development styles can be quite difficult. For example, a developer, being a unit testing enthusiast, stated that "I do not need test generation; I already practice unit testing (and/or TDD)". A developer, being a TDD convert, stated that "Test generation does not fit into the TDD process". It is not easy to change the habit or philosophy of these developers. Therefore, it is important to highlight how a new advanced technology relates to earlier approaches, emphasizing on complementary aspects instead of differences or total replacement. For example, if a developer has adopted TDD, it should be emphasized how parameterized unit testing (in combination of Pex) is a natural generalization of TDD, and not a radically new one or replacement.

During the evolution of Pex, before May 2009, the released versions of Pex were agnostic of the traditional unit tests manually written by developers, and would create test data from the scratch. Then the usage of parameterized unit tests with Pex did not have complementary or cooperative nature with manually written unit tests, which quite some developers typically would write. Then the adoption of Pex faced quite some resistance. To address such issue, in May 2009, the Pex team added a new feature, *Unit Tests as Inputs*, to the Pex release. With this feature, Pex can leverage already existing manually written unit tests that call parameterized unit tests. So that developers who get used to writing traditional unit tests can keep their existing habit while seamlessly enjoying the benefits provided by parameterized unit tests and Pex.

On the other hand, in August 2010, the Pex team added a new feature, *Test Promotion*, to allow developers to easily turn an automatically generated test into part of the traditional unit tests manually written by the developers. In particular, when the developers like a generated unit test, they can click on the "Promote" button to turn it into a "manually written unit test". Pex then will move it to the main test class and remove its attribute that indicates being an automatically generated unit test.

6.6 Listening to Practitioners

The success of Pex and its relatives largely attributed to that the Pex team closely interacted with and gathered feedback from the target tool users directly (e.g., via the MSDN Pex forum, tech support, outreaching to Microsoft engineers and outside-company .NET user groups) and indirectly (e.g., via interactions with the Microsoft Visual Studio product team, being a tool vendor to its huge user base). As stated in Section 4.1, from the period of October 2008 till November 2010, the MSDN forum dedicated to Pex and Moles included more than 1,400 forum posts made by the user community.

Besides interacting directly and indirectly with the target tool users, the Pex team also proactively attended venues oriented for software practitioners or with significant audience as software practitioners. Attending one such venue helped shape or reinforce the founding of the Moles project, which eventually resulted in the Microsoft Fakes framework shipped with Visual Studio, achieving substantial industry impacts. We next describe the brief background on the birth of the Moles project.

Environment dependency (which the units under test commonly have) has been a well-known problem for unit testing, including automatic unit-test generation, as aimed by Pex. If environment dependency involves native code, e.g., x86 instructions called through the P/Invoke mechanism of .NET, then Pex cannot instrument or monitor such native code. Even when environment dependency involves managed code and Pex can instrument the code, instrumenting and monitoring such managed code incur significant performance overhead including the huge path-exploration space to be explored by Pex.

Initially, to apply Pex on real-world code with environment dependency, developers were suggested to refactor their code by introducing explicit interface boundaries and allowing different interface implementations, i.e., by employing the design pattern of dependency injection [9]. However, often the time, developers hesitate to refactor their production code for only the testing purpose. In addition, when the code is legacy code, refactoring it is more difficult (or even infeasible when the legacy code is from a third party). Instead of further educating developers to refactor their code for dependency injection, the Pex team decided to take a different route to solve the problem after observing much resistance from the target tool users on such refactoring.

In August 2008, the Pex team members attended the Agile 2008 conference, a conference in agile development with significant audience from industry. There, the Pex team members observed that mock objects caught a lot of attentions from practitioners there, being a hot topic, and there was a strong need of providing effective tool support for mocking. Then the observation from broad practitioner communities at the conference reinforced the Pex team's determination and efforts on starting the Moles project, which was viewed as critical in enabling the success and adoption of Pex in practice. After months of tool development efforts, Moles was released to the public in September 2009, was released as a standalone tool in the Visual Studio Gallery in May 2010, and was shipped as Fakes with Visual Studio 2012 in August 2012, achieving high industry impacts (considering the huge user base of Visual Studio).

6.7 Collaborating with Academia

Collaborations between the Pex team and the academia have been very fruitful, playing important factors in developing a number of key technologies to enable the success of Pex and its relatives. Such collaborations are win-win for both the Pex team and the collaborating academic researchers. For example, the Pex team could extend its man power (with only two full-time employees) to a larger scope in order to feasibly explore both short-term and long-term research goals. On the other hand, the academic researchers could (1) focus on important research problems (in practice) determined with the help of the Pex team, and (2) leverage powerful tool infrastructures developed by the Pex team so that the academic researchers can focus main efforts on research innovations other than infrastructure-building engineering efforts, which are also critical to enable applying proposed tool features (developed for the research innovations) on real-world code bases but which themselves may not be research innovations.

We classify the collaboration forms as industry-located and academia-located collaborations as discussed below. Note that below we mention primarily concrete examples of collaborations that already directly resulted in or indirectly contributed to technologies as part of releases of Pex or its relatives.

Industry-located Collaborations. In the form of industry-located collaborations, collaborative research is conducted at the collaborating industrial research lab (e.g., Microsoft Research) with academic researchers as consulting visiting faculty or student interns paid by the research lab. This form can ease transferring the collaborative research outcomes (once demonstrated to be valuable to be transferred) to be parts of the released industrial tools, without complications of intellectual property (IP) issues. We illustrate concrete examples of such form as below:

- *Faculty visits*. The collaborating industrial research lab hosts a faculty member as a consulting researcher to visit the re-

search lab for conducting collaborative research. For example, Xie, the third author of this paper, spent some time visiting the Pex team during summers in the past several years. Xie's visit during 2008 Summer resulted in the Fitnex search strategy [75], being integrated as part of the Pex releases since September 2008 (note that the resulting Fitnex source code was also released as open source [19] to engage the research community to develop future extensions of Pex). Based on a technique of regression test generation [57], Xie's visit during 2010 summer resulted in the game type of coding duels [66,74], being integrated as the major game type of Pex4Fun since July 2010. Voronkov's visit resulted in string analysis [25] integrated in Pex releases.

- *Student internships*. The internship of Lakhotia [44] resulted in an approach called FloPSy for handling constraints over floating point variables, being part of Pex releases (note that the resulting FloPSy source code was also released as open source [6] to engage the research community to develop further improvement on solving floating point constraints). The internship of Thummalapenta [59] resulted in an approach called DyGen for generating tests via mining dynamic traces recorded during program executions; this approach was integrated in internal releases of a tool extended from Pex. The internship of Vanoverberghe [69] resulted in the definition and measurement of state coverage; measuring state coverage was integrated in releases of Pex since September 2010, but it has not been enabled by default due to high runtime overhead.

Academia-located Collaborations. In the form of academia-located collaborations, collaborative research is conducted at the university where the collaborating academic researchers are (e.g., North Carolina State University previously and University of Illinois at Urbana-Champaign currently where the third author was/is affiliated, respectively). Although the collaborative research outcomes produced in this form cannot be directly incorporated to be parts of the released industrial tools (e.g., due to IP issues), such collaborations can still produce substantial indirect impacts on releases of industrial tools:

- *Immediate indirect impacts*. In collaboration with the Pex team, Li et al. [45] identified the challenge of complex regular expressions faced by Pex, and developed an approach, called Reggae, to reduce the exploration space of DSE when dealing with complex regular expressions. The Reggae approach provided an initial step to address the challenge of complex regular expressions and inspired researchers at Microsoft Research to develop a more advanced Rex approach [71], being part of the Pex releases since June 2010. In collaboration with the Pex team, Thummalapenta et al. [60] developed an approach, called MSeqGen, that statically mines code bases and extracts sequences related to receiver or argument object types of a method under test for generating method sequences in unit-test generation. The MSeqGen approach directly inspired its dynamic counterpart, called the DyGen approach, resulted from the internship of Thummalapenta [59], being part of internal tool releases, as described above. In collaboration with the Pex team, Pandita et al. [53] developed an approach to guide DSE to achieve boundary-value and logical coverage. Such approach provided an initial step to address a request frequently made by Pex users in demanding more and higher-quality test inputs being generated and reported beyond just those achieving new branch or block coverage. Along with the UnitPlus approach by Song et al. [55], such approach inspired the definition and measurement of state coverage, produced by the internship of Vanoverberghe [69], as described above, along with the work on augmented DSE by Jamrozik et al. [41,42] (which was part of Jamrozik's master thesis awarded for the Poland's best MSc thesis in computer science).
- *Long-term indirect impacts*. In collaboration with the Pex team, Csallner et al. [30] developed the DySy approach, an invariant inference tool based on DSE, showcasing that Pex is an extensible platform of dynamic program analysis and being the first open source Pex extension. In collaboration with the Pex team, Thummalapenta et al. [61] developed the Seeker approach, being a state-of-the-art approach for addressing one of the most critical challenges faced by Pex in practice: generating desirable method sequences for object-type method arguments. In collaboration with the Pex team, Xiao et al. [72] proposed the cooperative testing methodology to engage Pex users to cooperate with Pex to address those challenges faced by Pex. Such methodology has inspired and aligned future directions for improving Pex to better serve and engage practitioners.

7. CONCLUSION

In this paper, we have reported the technology background, tool overview, impacts, project timeline, lessons learned from Microsoft Research's Pex project, which has been going on for more than eight years. The Pex project has resulted in Pex, Moles as Fakes, Code Digger, and Pex4Fun/Code Hunt, which have accomplished substantial impacts. For example, Moles has been shipped as Fakes with Visual Studio since August 2012, benefiting a huge user base of Visual Studio around the world. The number of download counts of Pex and its lightweight version Code Digger has reached tens of thousands within one or two years. Pex4Fun, released in June 2010, has achieved high educational impacts, reflected by the number of clicks of the "Ask Pex!" button as over 1.5 million till July 2014. Code Hunt, evolved from Pex4Fun, has been quickly gaining popularity. The Pex project at Microsoft Research is still ongoing for further technology transfer. We hope that our reported experiences can inspire more high-impact technology-transfer research from the research community.

In future work, we plan to collect quantitative data for further corroborating the learned lessons reported in this paper. For example, we plan to manually study Pex-related questions being asked by Pex users in StackOverflow [21] and previously in a MSDN Forum [16]. In addition, we plan to conduct a user survey among Pex users or conduct a field study among Pex users to gather their feedback on using Pex along with learning about their testing practices assisted by Pex.

Acknowledgments

We thank people from Microsoft Research and product teams along with academic collaborators for their assistance on the Pex project. Tao Xie's work is supported in part by a Microsoft Research Award, NSF grants CCF-1349666, CNS-1434582, CCF-1434596, CCF-1434590, CNS-1439481, and NSF of China No. 61228203.

8. REFERENCES

[1] Blog post: Fun with the ResourceReader. `http://blogs.msdn.com/b/nikolait/archive/2008/06/04/fun-with-the-resourcereader.aspx`.

[2] Blog post: Pex, dynamic analysis and test generation for .NET. http://blog.dotnetwiki.org/2007/03/08/PexDynamicAnalysisAndTestGenerationForNet.aspx.

[3] Blog post: What if coding were a game? http://blogs.msdn.com/b/msr_er/archive/2014/05/15/what-if-coding-were-a-game.aspx.

[4] Facebook Page on Code Hunt Game. https://www.facebook.com/codehuntgame.

[5] Facebook Page on Pex and Moles. https://www.facebook.com/PexMoles.

[6] Flopsy - search-based floating point constraint solving for symbolic execution. http://pexarithmeticsolver.codeplex.com/.

[7] ICFP Programming Contest 2013. http://research.microsoft.com/en-us/events/icfpcontest2013/.

[8] ICSE 2011 Pex4Fun Contest. http://research.microsoft.com/ICSE2011Contest.

[9] Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html, January 2004.

[10] Microsoft Customer Experience Improvement Program. http://www.microsoft.com/products/ceip/.

[11] Microsoft Devlabs Extensions. http://msdn.microsoft.com/DevLabs.

[12] Microsoft Patterns & Practices SharePoint Guidance. http://spg.codeplex.com/.

[13] Microsoft Visual Studio 2010 Moles x86 - isolation framework for .NET. http://visualstudiogallery.msdn.microsoft.com/b3b41648-1c21-471f-a2b0-f76d8fb932ee/.

[14] Microsoft Visual Studio Gallery. http://visualstudiogallery.msdn.microsoft.com/.

[15] Microsoft Visual Studio Gallery: Microsoft Code Digger. http://visualstudiogallery.msdn.microsoft.com/fb5badda-4ea3-4314-a723-a1975cbdabb4.

[16] MSDN Forum on Pex and Moles PowerTool. http://social.msdn.microsoft.com/Forums/en-US/home?forum=pex.

[17] MSDN: Isolating code under test with Microsoft Fakes. http://msdn.microsoft.com/en-us/library/hh549175(v=vs.110).aspx.

[18] Open source Pex extension: Fitnex. http://pexase.codeplex.com/wikipage?title=Fitnex.

[19] Open source Pex extensions by the Automated Software Engineering Group at Illinois. http://pexase.codeplex.com/.

[20] Publications from the Microsoft Research Pex project. http://research.microsoft.com/projects/pex/publications.aspx.

[21] Stackoverflow questions tagged with Pex. http://stackoverflow.com/questions/tagged/pex.

[22] T. Akiba, K. Imajo, H. Iwami, Y. Iwata, T. Kataoka, N. Takahashi, M. Moskal, , and N. Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time. Technical report, Microsoft Research, December 2013.

[23] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[24] J. Bishop, J. de Halleux, N. Tillmann, N. Horspool, D. Syme, and T. Xie. Browser-based software for technology transfer. In *Proc. SAICSIT, Industry Oriented Paper*, pages 338–340, 2011.

[25] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proc. TACAS*, pages 307–321, 2009.

[26] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ISSTA*, pages 169–180, 2006.

[27] L. C. Briand. Embracing the engineering side of software engineering. *IEEE Software*, 29(4):96, 2012.

[28] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446, 2008.

[29] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[30] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. ICSE*, pages 281–290, 2008.

[31] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. CRANE: Failure prediction, change analysis and test prioritization in practice - experiences from Windows. In *Proc. ICST*, pages 357–366, 2011.

[32] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. XIAO: Tuning code clones at hands of engineers in practice. In *Proc. ACSAC*, pages 369–378, 2012.

[33] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proc. ISSTA*, pages 207–218, 2010.

[34] J. de Halleux and N. Tillmann. Moles: tool-assisted environment isolation with closures. In *Proc. TOOLS*, pages 253–270, 2010.

[35] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.

[36] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proc. ISSTA*, pages 291–301, 2013.

[37] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proc. ICSE*, pages 225–234, 2010.

[38] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[39] W. Grieskamp. Microsoft's protocol documentation program: A success story for model-based testing. In *Proc. TAIC PART*, pages 7–7, 2010.

[40] W. Grieskamp, N. Tillmann, and W. Schulte. XRT–exploring runtime for .NET architecture and applications. *Electron. Notes Theor. Comput. Sci.*, 144(3):3–26, Feb. 2006.

[41] K. Jamrozik, G. Fraser, N. Tillmann, and J. de Halleux. Augmented dynamic symbolic execution. In *Proc. ASE*, pages 254–257, 2012.

[42] K. Jamrozik, G. Fraser, N. Tillmann, and J. de Halleux. Generating test suites with augmented dynamic symbolic execution. In *Proc. TAP*, pages 152–167, 2013.

[43] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[44] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. Flopsy: Search-based floating point constraint solving for symbolic execution. In *Proc. ICTSS*, pages 142–157, 2010.

[45] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Proc. ASE*, pages 515–519, 2009.

[46] J.-G. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie. Software analytics for incident management of online services: An experience report. In *Proc. ASE*, pages 475–485, 2013.

[47] P. McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[48] L. J. Osterweil, C. Ghezzi, J. Kramer, and A. L. Wolf. Determining the impact of software engineering research on practice. *IEEE Computer*, 41(3):39–49, 2008.

[49] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proc. DBTest*, pages 4–9, 2011.

[50] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *Proc. ASE*, pages 73–82, 2011.

[51] K. Pan, X. Wu, and T. Xie. Automatic test generation for mutation testing on database applications. In *Proc. AST*, pages 111–117, 2013.

[52] K. Pan, X. Wu, and T. Xie. Guided test generation for database applications via synthesized database interactions. *ACM Trans. Softw. Eng. Methodol.*, 23(2):12:1–12:27, Apr. 2014.

[53] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proc. ICSM*, pages 1–10, 2010.

[54] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[55] Y. Song, S. Thummalapenta, and T. Xie. UnitPlus: Assisting developer testing in Eclipse. In *Proc. ETX*, pages 26–30, 2007.

[56] J. Strejček and M. Trtík. Abstracting path conditions. In *Proc. ISSTA*, pages 155–165, 2012.

[57] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.

[58] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: Guided path exploration for efficient regression test generation. In *Proc. ISSTA*, pages 1–11, 2011.

[59] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proc. TAP*, pages 77–93, 2010.

[60] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, 2009.

[61] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proc. OOPSLA*, pages 189–206, 2011.

[62] N. Tillmann, J. Bishop, N. Horspool, D. Perelman, and T. Xie. Code Hunt – searching for secret code for fun. In *Proc. SBST*, pages 23–26, 2014.

[63] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[64] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Code Hunt: Gamifying teaching and learning of computer science at scale. In *Proc. Learning at Scale*, pages 221–222, 2014.

[65] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Constructing coding duels in Pex4Fun and Code Hunt. In *Proc. ISSTA, Tool Demo*, pages 445–448, 2014.

[66] N. Tillmann, J. de Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *Proc. ICSE, Software Engineering Education (SEE)*, pages 1117–1126, 2013.

[67] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[68] N. Tillmann and W. Schulte. Parameterized unit tests with Unit Meister. In *Proc. ESEC/FSE*, pages 241–244, 2005.

[69] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *Proc. SOFSEM*, pages 542–553, 2012.

[70] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Formal methods and testing. chapter Model-based Testing of Object-oriented Reactive Systems with Spec Explorer, pages 39–76. Springer-Verlag, 2008.

[71] M. Veanes, J. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *In Proc. ICST*, pages 498–507, 2010.

[72] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proc. ICSE*, pages 611–620, 2011.

[73] T. Xie, J. de Halleux, N. Tillmann, and W. Schulte. Teaching and training developer-testing techniques and tool support. In *Proc. SPLASH, Educators' and Trainers' Symposium*, pages 175–182, 2010.

[74] T. Xie, N. Tillmann, and J. de Halleux. Educational software engineering: Where software engineering, education, and gaming meet. In *Proc. GAS*, pages 36–39, 2013.

[75] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, pages 359–368, 2009.

[76] D. Zhang, Y. Dang, J.-G. Lou, S. Han, H. Zhang, and T. Xie. Software analytics as a learning case in practice: Approaches and experiences. In *Proc. MALETS*, pages 55–58, 2011.

[77] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie. Software analytics in practice. *IEEE Software*, 30(5):30–37, 2013.

[78] D. Zhang and T. Xie. Pathways to technology transfer and adoption: Achievements and challenges. In *Proc. ICSE, Software Engineering in Practice (SEIP), Mini-Tutorial*, pages 951–952, 2013.

[79] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, pages 1–10, 2010.

[80] H. Zhong, S. Thummalapenta, and T. Xie. Exposing behavioral differences in cross-language api mapping relations. In *Proc. FASE*, pages 130–145, 2013.

[81] Y. Zhou. Connecting technology with real-world problems - from copy-paste detection to detecting known bugs (keynote abstract). In *Proc. MSR*, pages 2–2, 2011.