Cooperative Testing and Analysis: Human-Tool, Tool-Tool, and Human-Human Cooperations to Get Work Done

(Keynote Paper)

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
xie@csc.ncsu.edu

***Abstract*—Tool automation to reduce manual effort has been an active research area in various subfields of software engineering such as software testing and analysis. To maximize the value of software testing and analysis, effective support for cooperation between engineers and tools is greatly needed and yet lacking in state-of-the-art research and practice. In particular, testing and analysis are in a great need of (1) effective ways for engineers to communicate their testing or analysis goals and guidance to tools and (2) tools with strong enough capabilities to accomplish the given testing or analysis goals and with effective ways to communicate challenges faced by them to engineers — enabling a feedback loop between engineers and tools to refine and accomplish the testing or analysis goals. In addition, different tools have their respective strengths and weaknesses, and there is also a great need of allowing these tools to cooperate with each other. Similarly, there is a great need of allowing engineers (or even users) to cooperate to help tools such as in the form of crowdsourcing. A new research frontier on synergistic cooperations between humans and tools, tools and tools, and humans and humans is yet to be explored. This paper presents recent example advances on cooperative testing and analysis.**

***Keywords*-cooperative testing and analysis; human-assisted computing; human-centric computing; tool integration; crowdsourcing**

I. Introduction

Manual software testing and analysis are known to be labor intensive and insufficient. To reduce manual effort in software testing and analysis, testing and analysis tools can be applied to automate activities in software testing and analysis (such as test-input generation), enabling economical use of resources. To maximize the value of testing and analysis, effective and efficient support for cooperation between engineers and tools is greatly needed. In particular, both research and practice are in a great need of (1) effective ways for engineers to communicate their testing or analysis goals and guidance to tools and (2) tools with strong enough capabilities to accomplish the given testing or analysis goals. To meet this need, recent research starts to explore a new research frontier on synergistic cooperation between engineers and tools; such human-tool cooperation is a type of cooperative testing and analysis [15].

Generally there are various types of cooperative testing and analysis. This paper summarizes three types: human-tool cooperation, tool-tool cooperation, and human-human cooperation. Human-tool cooperation further consists of two sub-types, depending on who are on the "driver" seat to conduct major work: human-assisted computing and human-centric computing. In human-assisted computing, tools are on the "driver" seat and engineers provide guidance to the tools so that the tools could better carry out the work. In contrast, in human-centric computing, engineers are on the "driver" seat and tools provide guidance to the engineers so that the engineers could better carry out the work. Tool-tool cooperation is often in the form of tool integration. Human-human cooperation is often in the form of crowdsourcing. The rest of the paper illustrates examples of these five types of cooperative testing and analysis: human-assisted computing (human-tool cooperation) (Section II), human-centric computing (human-tool cooperation) (Section III), tool integration (tool-tool cooperation) (Section IV), and crowdsourcing (human-human cooperation) (Section V).

II. Human-Assisted Computing: Human-Tool Cooperation

Human-assisted computing in the context of software testing and analysis consists of three phases: (1) *Setup* phase: engineers set up and apply tools to conduct initial testing and analysis; (2) *Feedback* phase: the tools provide feedback to the engineers; (3) *Action* phase: the engineers provide guidance to the tools based on the feedback. The feedback phase and the action phase form a feedback-action loop that enables engineers and tools to refine and accomplish specific goals of testing and analysis.

For example, producing high-covering test inputs is an important goal of software testing, since high code coverage can help identify the insufficiency of test inputs, e.g., showing which parts of the program under test are not tested by the test inputs. To reduce the manual burden of manually producing test inputs, engineers can apply tools built based on automated test-generation approaches to generate test inputs automatically, such as Dynamic Symbolic Execution (DSE) [6]. DSE executes the program under test symbolically with arbitrary or default inputs. Along the execution

path, DSE collects the constraints in the branch statements to form a path condition and negates part of the path condition to obtain a new path condition that leads to a new path. The new path condition is then fed to a constraint solver, which computes new test inputs for exploring new paths.

Although these automated test-generation tools can easily achieve high code coverage on simple programs, they face different kinds of challenges to achieve high code coverage on complex programs in practice. Based on recent studies [15], the top two major problems that prevent these tools from achieving high code coverage of object-oriented programs are (1) the object-creation problem (OCP), where tools fail to generate sequences of method calls to construct desired object states for covering certain branches; (2) the external-method-call problem (EMCP), where tools cannot deal with method calls to external libraries, such as native system libraries or pre-compiled third-party libraries. The main reason for OCPs is that certain branches of the program under test require desired object states that cannot be generated by the tools. The main reason for EMCPs is that external-method calls cannot be precisely analyzed by the tools or can throw exceptions to hinder the test generation.

Since tools are imperfect in dealing with various challenges in achieving high code coverage, cooperative testing identifies problems faced by tools during test generation (with the focus on OCPs and EMCPs), enabling engineers and tools to generate test inputs cooperatively as follows. The engineers first apply the tools to automatically generate test inputs until the tools cannot achieve higher code coverage or run out of pre-defined resources. Then the tools report the achieved coverage and problems that prevent them from achieving higher coverage. By looking into the problems, the engineers provide guidance to the tools, helping the tools address these problems. As an example of providing guidance to the tools, the engineers can write factory methods that encode sequences of method calls to produce desired object states to deal with OCPs [11]. To deal with EMCPs, the engineers can instruct tools to instrument and explore the external libraries or write mock objects [13] to simulate the dependency. After providing guidance to the tools based on the reported problems, the engineers can reapply the tools to generate test inputs for achieving better coverage. Such iterations of applying the tools and providing the guidance can continue until satisfied coverage is achieved.

To achieve this cooperation between engineers and tools, the tools need to precisely report problems for reducing effort from the engineers. Straightforward approaches such as locating all non-primitive object types and external method calls produce too many irrelevant problem candidates that do not prevent tools from achieving higher code coverage. To address the needs of precisely identifying problems, our previous Covana approach [15] prunes the irrelevant problem candidates using the data dependencies of partially-covered branch statements on problem candidates.

III. Human-Centric Computing: Human-Tool Cooperation

In collaboration with Microsoft Research, our previous work [12] has proposed the game type of coding duels for a web-based serious gaming platform, called Pex for Fun (in short as Pex4Fun) (http://www.pexforfun.com/). Any one around the world could create coding duels for others to play besides playing existing coding duels themselves. In a coding duel, the player is given a working implementation, being an empty or faulty implementation of a method (with optional comments to give the player hints on reducing the difficulty level of gaming). Then the player is asked to modify the working implementation to make its behavior (in terms of the method inputs and return) to be the same as the secret (golden) implementation (which is supplied by the game creator but is not visible to the player). Over the game-playing process, the player has the opportunity to request the gaming platform to provide the following feedback to the player (by clicking the "Ask Pex!" button on the user interface): (1) under what method input(s) the working implementation and the secret implementation have different method returns; (2) under what method input(s) the working implementation and the secret implementation have the same method return. The gaming platform leverages a DSE engine called Pex [11] to provide such feedback.

Pex4Fun has been increasingly gaining popularity in the community. Since it was released to the public in 2010 summer, the number of clicks of the "Ask Pex!" button (indicating the attempts made by players to solve games at Pex4Fun) has reached more than 938,000 as of early July 2012. In May 2011, Microsoft Research hosted a contest on solving coding duels at the 2011 International Conference on Software Engineering (ICSE 2011). The ICSE 2011 contest received 7,000 Pex4Fun attempts, 450 duels completed, and 28 participants (though likely more, since some did not actually enter the official contest portal to play the coding duels designed for the contest).

IV. Tool Integration: Tool-Tool Cooperation

Integration of analyses or tools has been pursued by various researchers [3]. Our previous work has integrated static analysis and dynamic analysis [4], integrated dynamic analysis and dynamic analysis [16], integrating dynamic analysis and static analysis [14], integrated dynamic analysis, static analysis, and dynamic analysis [2], [10].

When integrating stand-alone tools to be used in combination instead of choosing only one of them to be used, assessing these tools needs to take into account of the complementary effect of multiple tools. For example, for test-generation tools that aim to achieve high code coverage such as branch coverage, comparing just the percentages of branch coverage achieved by each tool is an existing common way of assessing and comparing the effectiveness of the tools. Such assessment and comparison would be

desirable when only one of the tools under comparison is selected to be used but undesirable when multiple tools are selected to be used in combination. To address this issue, our previous work [8] has proposed *branch ranking* to characterize which branches are more difficult to be covered by n tools under consideration for being selected to be used in combination. In particular, we rank all the branches in the code under test based on the number of tools that can cover them. A rank-1 branch is covered by only one of the n tools while a rank-2 branch is covered by only two of the n tools, and so on. If a tool can cover more top-ranked (e.g., rank 1 or 2) branches, this tool demonstrates better effectiveness in covering branches that are difficult to be covered by other tools. Then such tool is more desirable to be selected when multiple tools are selected to be used in combination.

V. Crowdsourcing: Human-Human Cooperation

Crowdsourcing could be leveraged for software testing and analysis. For example, crowdsourcing has been used to engage human crowds to solve manageable subproblems decomposed from the task of writing verifiable specifications [9]. Crowdsourcing has been used to engage human crowds to solve puzzles decomposed from object-creation problems and complex constraint-solving problems that are encountered by test-generation tools [1].

Pex4Fun [12] presented in Section III can also be viewed as a form of crowdsourcing. A coding-duel creator can construct a secret implementation in order to collect its alternative implementations for various purposes. For example, the coding-duel creator may seek a *better* alternative implementation in terms of design quality, performance, energy, etc. Indeed, when a coding duel is created by using a real-world method implementation as the secret implementation, it is very challenging for human crowds to win such coding duel.

In recent years, *debugging in the large* [5], [7] has been realized with industrial solutions from companies such as Microsoft. Such industrial solutions are deployed for collecting and leveraging a high volume of deployment-site usage data from human crowds to improve debugging with postmortem analysis. For example, the Microsoft Windows Error Reporting (WER) system [5] focuses on crash/hang debugging and the Microsoft PerfTrack and StackMine systems focus on performance debugging. Using a long-time period in the post-release stage and a huge number of information sources from real-world users, such debugging-in-the-large systems allow engineers to obtain distribution information of crashing/hanging/performance bugs to guide their debugging prioritization.

Acknowledgment. This work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, ARO grant W911NF-08-1-0443, an NSA Science of Security Lablet grant, a NIST grant, and a 2011 Microsoft Research Software Engineering Innovation Foundation Award. The author would like to thank his collaborating students at North Carolina State University and collaborators from Microsoft Research and other universities.

References

[1] N. Chen and S. Kim. Puzzle-based automatic testing: bringing humans into the loop by solving puzzles. In *Proc. ASE*, 2012.

[2] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 17(2):345–371, April 2008.

[3] M. B. Dwyer and S. G. Elbaum. Unifying verification and validation techniques: relating behavior and properties through partial evidence. In *Proc. FoSER*, pages 93–98, 2010.

[4] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: Dynamic symbolic execution guided with static verification results. In *Proc. ICSE, Demonstration*, pages 992–994, 2011.

[5] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proc. SOSP*, pages 103–116, 2009.

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

[7] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proc. ICSE*, pages 145–155, 2012.

[8] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

[9] T. W. Schiller and M. D. Ernst. Reducing the barriers to formal methods. In *Proc. OOPSLA*, 2012.

[10] 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.

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

[12] N. Tillmann, J. de Halleux, and T. Xie. Pex4Fun: Teaching and learning computer science via social gaming. In *Proc. CSEET, Practice and Methods Presentations, & Tutorials (PMP&T)*, pages 546–548, 2011.

[13] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *Proc. ASE*, pages 365–368, 2006.

[14] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proc. FSE*, 2012.

[15] 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.

[16] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.