Xie T, Zhang L, Xiao X *et al.* Cooperative software testing and analysis: Advances and challenges. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(4): 713–723 July 2014. DOI 10.1007/s11390-014-1461-6

Cooperative Software Testing and Analysis: Advances and Challenges

Tao Xie[1] (谢 涛), *Senior Member*, *ACM*, *IEEE*, Lu Zhang[2,3] (张 路), *Senior Member*, *CCF*, *Member*, *ACM*
Xusheng Xiao[4] (肖旭生), Ying-Fei Xiong[2,3] (熊英飞), *Member*, *CCF*, *ACM*, *IEEE*
and Dan Hao[2,3] (郝 丹), *Member*, *CCF*, *ACM*, *IEEE*

[1]*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.*
[2]*Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China*
[3]*Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*
[4]*NEC Laboratories America, Inc., Princeton, NJ 08520, U.S.A.*

E-mail: taoxie@illinois.edu; zhanglu@sei.pku.edu.cn; xusheng.xiao@gmail.com; {xiongyf04, haod}@sei.pku.edu.cn

Received April 3, 2014; revised May 31, 2014.

Abstract In recent years, to maximize the value of software testing and analysis, we have proposed the methodology of cooperative software testing and analysis (in short as cooperative testing and analysis) to enable testing and analysis tools to cooperate with their users (in the form of tool-human cooperation), and enable one tool to cooperate with another tool (in the form of tool-tool cooperation). Such cooperations are motivated by the observation that a tool is typically not powerful enough to address complications in testing or analysis of complex real-world software, and the tool user or another tool may be able to help out some problems faced by the tool. To enable tool-human or tool-tool cooperation, effective mechanisms need to be developed 1) for a tool to communicate problems faced by the tool to the tool user or another tool, and 2) for the tool user or another tool to assist the tool to address the problems. Such methodology of cooperative testing and analysis forms a new research frontier on synergistic cooperations between humans and tools along with cooperations between tools and tools. This article presents recent example advances and challenges on cooperative testing and analysis.

Keywords software verification, testing and debugging, software quality

1 Introduction

Applying software testing and analysis tools is the most widely used way for improving software quality in practice. Typically, software testing[1-2] includes the following four major tasks: 1) generating test inputs, 2) creating expected outputs (also referred to as test oracles), 3) running test inputs, and 4) verifying actual test outputs. Software analysis (also referred to as program analysis) includes static analysis (i.e., statically analyzing software without executing the software) and dynamic analysis (i.e., dynamically analyzing software by executing the software). Software testing is considered as a type of dynamic analysis while static analysis is commonly used in static verification[3]: statically analyzing software to detect software faults or prove their absence, typically against given properties that the software is expected to satisfy.

However, in practice, software testing and analysis tools are typically not powerful enough to address complications in testing or analysis of real-world software, which is growing increasingly larger and more complex over time. Although ongoing and future research advances in tool automation can help improve the capability of such tools, such real-world complications would be still difficult for the tools to tackle in the foreseeable future. For example, our previous study[4] showed that even state-of-the-art test-generation tools may achieve no more than 65% code coverage, and recent studies[5] conducted by others also show that static analysis tools cannot prove various properties in complex software. For some of these problems faced by the tools, tool users, often being software engineers (e.g., developers and testers), may provide guidance to help the tools tackle these problems. In addition, different tools (along with their underlying analyses) have diffe-

Survey

This work is supported in part by the National Natural Science Foundation of China under Grant Nos. 61228203, 61225007, and 61272157, the National Science Foundation of USA under Grant Nos. CCF-1349666, CNS-1434582, CCF-1434596, CCF-1434590, CNS-1439481, and a Microsoft Research award.

A preliminary version of the paper was published as a keynote paper in the Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM 2012).

rent strengths and weaknesses, and thus they may complement each other. In other words, problems faced by a tool may be tackled by another tool.

To maximize the value of software testing and analysis, in recent years, we have proposed the methodology of cooperative software testing and analysis[6] (in short as cooperative testing and analysis) to enable testing and analysis tools to cooperate with their users (in the form of tool-human cooperation), and enable one tool to cooperate with another tool (in the form of tool-tool cooperation). To enable such tool-human or tool-tool cooperation, cooperative testing and analysis need to provide effective mechanisms 1) for a tool to communicate problems faced by the tool to the tool user or another tool, and 2) for the tool user or another tool to assist the tool to address the problems. The basic rationale of cooperative testing and analysis is that tools and tool users typically have their respective strengths and weaknesses, and similarly different tools typically have their respective strengths and weaknesses; enabling cooperation among these entities can provide opportunities to enhance their strengths and alleviate their weaknesses, respectively.

In particular, tool-human cooperation 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 or analysis integration. Note that here we use the terms of human-assisted computing and human-centric computing to intuitively denote the two sub-types of tool-human cooperation, instead of denoting an existing model of computing or defining a new model of computing.

Indeed, the topics of tool/analysis integration, human-computer interactions, or human decision making based on tool outputs have been long investigated in the research community. However, complementing with such existing efforts, our methodology of cooperative testing and analysis explicitly treats *cooperation* as a first-order entity, and often realizes cooperation via a *feedback loop* between the cooperating parties. In this methodology, testing or analysis algorithms need to address problems that emerge in such cooperation. For example, how to design or improve the tool algorithm to produce high-quality information (e.g., succinct yet sufficient information for decision making) for the tool user to consume and guide the tool?

Exploring effective mechanisms for tool-human cooperation and tool-tool cooperation, our methodology of cooperative testing and analysis forms a new research frontier on synergistic cooperations among humans and tools. This article presents recent example advances and challenges on cooperative testing and analysis. The rest of the paper illustrates examples of four types of cooperative testing and analysis: human-assisted computing (human-tool cooperation) (Section 2), human-centric computing (human-tool cooperation) (Section 3), and tool/analysis integration (tool-tool/analysis-analysis cooperation) (Section 4), as well as challenges in cooperative and analysis (Section 5).

2 Human-Assisted Computing: Human-Tool Cooperation

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. 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[7] that enables engineers and tools to refine and accomplish specific goals of testing and analysis. Note that in the action phase, some tool support can be provided to facilitate the engineers in providing guidance to the tools.

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 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)[8]. 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 often achieve high code coverage on simple programs, they face various problems to achieve high code coverage on complex programs in practice. Based on recent studies[4], 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 problems 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 predefined 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[9]. To deal with EMCPs, the engineers can instruct tools to instrument and explore the external libraries or write mock objects[10-12] 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 Covana approach[4] prunes the irrelevant problem candidates using the data dependencies of partially-covered branch statements on problem candidates. We next describe our Covana approach (Subsection 2.1) and then discuss how engineers can help DSE-based tools to address loop problems, another common type of problems faced by DSE-based tools (Subsection 2.2).

Note that although the rest of this section presents examples of cooperative testing and analysis only in the domain of test generation, such cooperative testing and analysis can be also seen in the domain of compiler optimization or static verification. For example, St-Amour *et al.*[13] proposed an approach to report optimizations that a compiler could perform on a program if the compiler has additional information, enabling users to provide further help for the compiler optimization in the program. Dillig *et al.*[5] proposed an approach to compute small and relevant queries that capture the facts that the analysis is missing when automated static analysis fails to verify a program. These queries are presented to users who decide whether the answers to queries are yes or no, and the answers are then used to either verify the program or prove the existence of a real fault. Dincklage and Diwan[14] proposed an analysis language and built a system to produce reasons when program analyses fail to produce desirable results. The objective of their approach is to express arbitrary data flow analyses using their analysis language and compute reasons for the failures.

2.1 Precise Identification of Problems for Structural Test Generation

To precisely identify problems faced by test-generation tools, our approach, Covana[4], is built based on the insight that partially-covered branch statements have data dependency on real problem candidates.

Covana consists of three main steps: problem-candidate identification, forward symbolic execution, and data dependence analysis. Covana accepts as input a program under test, and generates test inputs from automated test-generation tools (such as a DSE-based tool). Covana then treats the generated test inputs as program inputs, and leverages the DSE engine to perform forward symbolic execution on the program (program inputs are assigned with symbolic values). During execution, Covana monitors runtime events triggered by the DSE engine for identifying different types of problem candidates. After identifying problem candidates, Covana assigns symbolic values to elements of these problem candidates (such as return values of external-method calls and fields of program inputs), performs forward symbolic execution on these symbolic values, and collects runtime information, such as symbolic expressions and exceptions. Covana then uses the collected structural coverage and runtime information to compute the data dependencies of partially-covered branch statements on problem candidates. In Covana, symbolic execution is used to compute data dependencies for pruning irrelevant problem candidates, and target states for solving problems.

EMCP Identification. An external method call is considered as an EMCP candidate if its arguments have data dependencies on program inputs. The reason is that those other external-method calls (whose arguments have no data dependencies on program inputs) usually print a constant string or put a thread to sleep for some time. Such external-method calls typically do not compromise coverage achieved by test-generation tools and can be pruned. For each EMCP candidate, Covana assigns symbolic values to the return value of the identified external-method call for data dependence analysis. Covana prunes EMCP candidates if none of partially-covered branch statements have data dependencies on the candidates for their return values.

Uncaught exceptions thrown by external-method calls abort test executions, preventing structural test-generation tools from exploring remaining parts of the program under test. To identify such external-method calls, Covana collects exceptions during test executions and analyzes the stack traces to extract external-method calls. If the remaining parts of the program after the call sites of the external-method calls are not covered, Covana reports these external-method calls as EMCPs.

OCP Identification. Since an OCP requires objects of a non-primitive type as program inputs, Covana considers as problem candidates only program inputs whose type is a non-primitive type, such as a user-defined type. Covana assigns symbolic values to such program inputs and all their fields for computing data dependencies. If a partially-covered branch statement has data dependencies on only program inputs, Covana directly reports the program inputs as OCPs. However, if a partially-covered branch statement has data dependencies on a field of a program input, Covana performs further analysis to identify which field of the program input is the cause to the OCP.

2.2 Loop Problems for Structural Test Generation

Besides OCPs and EMCPs, DSE-based tools still face another significant problem: how to handle loops. As a special type of branches, loops can cause the number of paths under exploration to grow exponentially. Even worse, the number of paths becomes infinite due to the presence of input-dependent loops (IDLs)[①], causing DSE to run out of resources (e.g., the allocated time or number of explored paths) before achieving satisfactory code coverage. For example, a recent study[15] shows that DSE may keep unfolding an IDL without achieving coverage of any new branches. Recent research has tried to address this problem by using a bound to constrain loop unrolling[16], using search-guiding heuristics to guide path exploration[9,17], or using loop summarization to summarize loops based on inferred loop invariants[18-20] (referred to as *loop summarization*).

Our previous work[21] provides two-phase characteristic studies on loop problems for DSE-based tools. Our proposed study methodology starts with conducting a literature-survey study to investigate how technical problems such as loop problems compromise automated test generation, and which existing techniques are proposed to deal with such technical problems. Then the study methodology continues with conducting an empirical study of applying the existing techniques on real-world applications sampled based on the literature-survey results and major open-source project hostings. This empirical study investigates the pervasiveness of the technical problems and how well existing techniques can address such problems among real-world applications.

Based on the literature survey of 159 published articles on symbolic execution, our study finds that loop problems are one of the major problems for software testing via symbolic execution, and bounded iteration and search-guiding heuristics are two most widely used techniques for dealing with loops. We then conducted an empirical study of applying these two techniques on 16 open-source applications, and results show that bounded iteration and search-guiding heuristics can address 65% of unbounded loops whose side effects may compromise coverage of subsequent branches.

For remaining unbounded loops that cannot be handled by bounded iteration and search-guiding heuristics, we provide three guidelines with the last one engaging human guidance in the form of cooperative testing: 1) data validation: for applications that impose heavy validation on input values, a separate data generator that generates valid objects may be employed and only the constraints that lead to valid objects should be collected; 2) weighted heuristics: search-guiding heuristics should assign lower probabilities on loop guards than other branches, and branches collected in later iterations should be given lower probabilities than branches collected in earlier iterations; 3) cooperative analysis: for complex IDLs that interleave nested loops and branches, a test-generation tool may identify such loops and report to developers; developers can provide manually specified loop invariants or summaries to assist the tool in addressing the loop problems.

[①]Input-dependent loops are loops whose iteration numbers depend on some unbounded input.

3 Human-Centric Computing: Human-Tool Cooperation

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. Human-centric computing in the context of software testing and analysis consists of three phases in order to assist engineers to accomplish a goal (which can be beyond a typical goal of testing and analysis) such as debugging or fixing code. These three phases are as below: 1) *setup* phase: engineers set up and apply tools to conduct initial testing and analysis; 2) *action* phase: the engineers act on the results of the initial testing and analysis towards the goal; 3) *feedback* phase: the tools provide feedback to the engineers. The feedback phase and the action phase form a feedback-action loop that enables engineers and tools to refine and accomplish the goal. Although these three phases are similar to the ones in human-assisted computing, in human-centric computing, engineers are the ones who are mainly responsible for leading the efforts for accomplishing the goal while tools just provide assistance to the engineers.

For example, our previous work[22-24] 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/). In a coding duel, the goal of the game player (which can be a software engineer or new learner of programming) is to modify the given working implementation (initially empty or wrong) to match the behavior of the secret implementation. In the setup phase, the game player initially requests to apply a DSE-based tool to generate test inputs to show the different behaviors of the working and secret implementations (along with their same behaviors). In the action phase, based on the feedback given by the DSE-based tool, the game player modifies the working implementation, attempting to accomplish the goal by making the reported different behaviors disappear while retaining the reported same behaviors. In the feedback phase, the game player requests the DSE-based tool to give further feedback on the revised working implementation, and then moves to the action phase again.

As another example, our previous work[25-26] has proposed an interactive testing-based fault localization approach (whose tool is named VIDA). By using the VIDA tool, the goal of a developer is to identify the faulty statements in the program under debugging. In the setup phase, the developer initially runs VIDA to identify a set of suspicious statements based on the statistical analysis of test cases. In the action phase, from the reported set of suspicious statements, the developer selects one of these suspicious statements, checks the correctness of the selected statement and the status of the program after executing the selected statement: if the selected statement is faulty according to the developer's check, the fault-localization process ends; otherwise, based on the status of the program after executing the selected statement, VIDA further reduces the scope of suspicious statements, and then focuses on the no-innocent scope. In the feedback phase, based on the developer's check, VIDA modifies its analysis of test cases to recommend another set of suspicious statements, and then move to the action phase again.

We next give more details on Pex4Fun and its extension Code Hunt (Subsection 3.1) and interactive debugging (Subsection 3.2) as two concrete examples of human-assisted computing.

3.1 Pex4Fun and Code Hunt

In recent years, educational software engineering[27] (i.e., software engineering for education) has emerged as a research subarea of software engineering, for broadening the impact of software engineering research to the education domain. In this subarea, researchers develop software engineering technologies (e.g., software testing and analysis, software analytics) for general educational tasks, including but not limited to software engineering education or computer science education. This subsection illustrates an example of educational software engineering that leverages cooperative testing and analysis, especially human-centric computing, for teaching and learning programming and software engineering.

Teaching and learning programming and software engineering have received a lot of attention from researchers and educators. There exist various programming environments[28-29] for instilling fun into students' programming-learning experiences, especially for beginner learners. Although these programming environments help teach and learn programming concepts for beginner learners, these environments typically target at some specialized programming languages other than mainstream programming languages. In addition, these environments primarily target at teaching and learning programming without focusing on software engineering.

To address such issues, in collaboration with Microsoft Research, our previous work[22-24] 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 can 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[9] 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 1 479 000 as of early April 2014. 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).

In early 2014, Microsoft Research released Code Hunt[30-31] (http://www.codehunt.com/), a dramatic evolution of the Pex4Fun web platform. In Code Hunt, the game consists of a series of worlds and levels, which get increasingly challenging. In each level, the player has to discover a secret code fragment and write code for it. The game has sounds and a leaderboard to keep the player engaged. The game works in any modern browser, and currently supports C# or Java programs. Both Pex4Fun and Code Hunt are representative examples of educational software engineering[27] (i.e., software engineering for education), which develops software engineering technologies (e.g., software testing and analysis) for general educational tasks, going beyond educational tasks for software engineering.

3.2 Interactive Debugging

Software debugging consists of fault localization and fault fixing: the former aims to localize faulty statements in the program whereas the latter aims to correct the program by replacing the faulty statements. Without identifying the faulty statements correctly, it is hard to fix faults. To reduce the manual burden of localizing faulty statements, various automatic fault-localization approaches have been proposed to reduce the scope of suspicious statements, which may be faulty statements. However, in software development, fault localization is an intelligent work of developers filled with developers' development experience and understanding of the program under debugging. Without such knowledge, such automatic fault-localization approaches can hardly identify faulty statements correctly. Furthermore, as these automatic approaches cannot precisely point out where the faulty statements are, these approaches are rarely used in practice.

To address this issue, our previous work[25-26] has proposed an interactive testing-based fault localization approach (whose tool is named as VIDA), which combines developers' debugging experience and statistical analysis on the execution information of the program under test. With VIDA, developers may identify the faulty statements as follows. First, VIDA identifies a set of suspicious statements based on the statistical analysis of test cases. Second, VIDA asks developers to select one of these suspicious statements, and to check the correctness of the selected statement and the status of the program after executing the selected statement. If the selected statement is faulty according to developers' check, the fault-localization process ends. Otherwise, based on the status of the program after executing the selected statement, VIDA further reduces the scope of suspicious statements, and then focuses on the not-innocent scope. Based on the developers' check, VIDA modifies its analysis of test cases so as to recommend another set of suspicious statements. Repeating the preceding process, VIDA can aid developers to identify where the faulty statements are correctly.

Different from existing automated fault-localization approaches, VIDA combines developers' feedback to improve the efficiency of fault-localization. Based on the execution information, existing approaches recommend a list of suspicious statements, which are ranked based on their probability of containing faults, but the statement with the largest suspicion may not contain faults at all. To help find out where the fault is, VIDA improves the existing approaches by modifying the suspicions of statements following developers' check. In particular, if the status of the program after executing the selected statement is correct, the fault typically would lie in the statements executed after the selected statements; otherwise, the fault would lie in the statements executed before the selected statements. In other words, based on developers' check on such a status, the set of suspicious statements are reduced.

Furthermore, VIDA improves the efficiency of fault localization by following developers' debugging habit.

As existing approaches usually recommend a list of suspicious statements for developers, developers have to manually check one by one although these statements may spread around anywhere in the program. Developers' check on one statement cannot be used to aid their check on other statements. To fully leverage developers' feedback, VIDA combines existing fault-localization approaches with developers' debugging habit. Moreover, as VIDA follows developers' debugging habit, VIDA may be more applicable than existing fault-localization approaches.

4 Tool/Analysis Integration: Tool-Tool/ Analysis-Analysis Cooperation

Integration of analyses or tools has been pursued by various researchers[32]. Our previous work has integrated static analysis and dynamic analysis[33], integrated dynamic analysis and dynamic analysis[34], integrated dynamic analysis and static analysis[35], integrated dynamic analysis, static analysis, and dynamic analysis[36-37]. We next discuss how tool assessment needs to be adjusted when integrating the use of multiple tools on the program under analysis (Subsection 4.1) and then present a concrete example of integrating static and dynamic analyses (Subsection 4.2).

4.1 Tool-Tool Integration

When integrating stand-alone tools to be used in combination (either tools within the same integrated environment or tools from different integrated environments) 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[38] 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 a tool is more desirable to be selected when multiple tools are selected to be used in combination.

4.2 Analysis-Analysis Integration

Program analysis has been approached in two different directions. One is static analysis, where some properties about a program are analyzed by considering all possible inputs. The other one is dynamic analysis, where a program is executed with a specific input, and the properties are examined along the execution of this input. The result of static analysis is generic to all inputs, but is not always a deterministic result. For example, given a property P, a static analysis usually reports three kinds of results for a program: P holds, P does not hold, and P may hold. On the other hand, the result of a dynamic analysis is specific to one input, but is always deterministic, i.e., informing whether P holds or does not hold for a specific input. Furthermore, the result of a dynamic analysis is also companied with a specific input, enabling further analysis. In other words, we not only know that P holds, but also know on which specific input P holds.

Since static and dynamic analyses are complementary to each other, a new trend in program analysis is to mix static and dynamic analyses, forming a hybrid analysis. An example of this mixture can be seen in the area of PHP program maintenance. A common task in web application development is to change the presentation style of a web page. For static HTML pages, this task can be done easily in visual editors. However, for HTML pages generated from PHP programs, we have to directly modify the PHP source, and this task is not easy for two main reasons. First, the developer has to be familiar with the structure of the PHP code, which is usually different from that of the HTML page. In addition, the developer writing the PHP code and the developer changing the presentation style are often not the same person. Second, the developer needs to ensure that the presentation changes are correctly propagated back without any unexpected behavior. It is very common that different parts of the generated web pages depend on one location in the PHP source, and the developer has to be very careful to ensure that the change affects only the desired locations.

To address such issues, our previous work[35] proposes an approach, named as collaborative hybrid analysis, that automatically propagates the changes on the generated HTML pages back to their PHP sources. In this way, the developer needs to change only the generated web page, in visual editors or any other ways, and the changes can be automatically propagated back to the source code, and our approach ensures that the source code generates exactly the same HTML page.

This approach is based on the idea of SyncATL[39]. Given a model transformation program in ALT, SyncATL uses dynamic analysis to propagate the changes on the output model back to the input model. First, SyncATL runs the transformation in a special virtual machine, which does not only generate the output, but also builds precise links between the input and the output. When an element on the output is changed, SynATL finds the corresponding element in the source model, and changes the source element accordingly. SyncATL also checks whether multiple locations depend on the source element to ensure that no unexpected change is introduced.

SyncATL propagates the changes only from the output to the input. However, if we include the transformation program also in the links, it can also propagate the changes to the transformation program. A PHP program is essentially a transformation from some source data such as a database to an HTML page. As a result, the same approach can be used to propagate the changes on HTML pages to PHP programs.

However, there is a key difference: while a transformation input is specific to one execution of the transformation, a PHP code snippet may be shared in the generation of many web pages. As a result, when we propagate changes to a transformation program, using only dynamic analysis may result in unexpected results when the program is used in the generation of a different web page. As a result, our approach uses a hybrid analysis: after the changes are propagated back using the dynamic analysis, we further perform a static analysis to determine if the changes are safe in all possible executions.

The safety of a change is defined by its effect on the executions of the program. If a change affects only the statement in the execution that generates the modified HTML page, then the effect of this change has already been seen on the modified output, and thus the change is safe. Inversely, a change is potentially unsafe if a change may affect some code that is not included in the execution generating the modified HTML page. Our static analysis checks in all PHP source programs whether there exists any code that is data-dependent or control-dependent on the changed code, and is not included in the execution trace. This check can be easily performed by program slicing[40]. If such code is identified, we highlight the code and inform the programmers that the change cannot be directly performed to the source code.

5 Challenges

Economic Analysis for Cooperative Testing. In cooperative testing, to test complex programs, often a long list of problems can be presented to developers, and developers have limited time to solve all the problems. Thus, it is desirable to enable developers to maximize their testing goals within the given time of solving problems, such as achieving as high branch coverage as possible or covering critical target branches (e.g., assertion-violating branches) by solving as few problems as possible. To enable such economical cooperation of developers and tools, there is a strong need of economic analysis that estimates the benefit of solving a problem and the cost of covering a branch. With such analysis, we can answer questions such as "what if developers solve a specific problem, what benefits can such solving induce?" (i.e., the benefit of solving a problem), "how many problems need to be solved in order to cover a specific branch?" (i.e., the cost of covering a branch).

Visualization to Improve User Understanding of Problems. In cooperative testing, by looking at the problems faced by test-generation tools, developers provide guidance to help DSE-based tools address the problems and help the tools achieve higher code coverage. Although Covana[4] identifies problems faced by test-generation tools, the problems are shown to developers in the format of textual output. Such textual output is not readily understandable, and provides limited assistance to developers in investigating problems. Given an identified problem in the textual format, developers still need to locate the problem in the program and investigate the program to decide how to provide guidance. To help developers understand problems faced by DSE-based tools and improve the usability of DSE-based tools, there is a strong need of a visualization approach that visualizes the coverage achieved by DSE-based tools and the problem-analysis results for the not-covered areas, allowing developers to effectively and efficiently investigate the problems related to the not-covered areas.

Tool Support for Human Guidance. Even if developers understand what problems are faced by the tools, developers may have difficulties in implementing their guidance from scratch, and it could be quite time-consuming if they are not familiar with the program under test. There is a strong need of program synthesis techniques to generate a partial solution to better assist developers in providing guidance. For example, to provide a factory method, such techniques can synthesize a factory method that produces an object state that is close to the target state based on the generated object states, and can suggest methods for developers to modify the object state towards the target state.

Interactive Debugging for Fault Fixing. Existing interactive debugging approaches[25-26,41] typically focus on fault localization rather than fault fixing. Although automatic fault fixing[42] has been pursued, there is an open challenge on how to conduct interactive fault fixing by combining developers' experience with existing automatic techniques. As fault fixing is also a human-centric task, there is a strong need of future research of improving the efficiency of fault fixing in an interactive way.

Integration of Program Analyses. While more and more approaches are proposed by integrating different kinds of analyses, there still exists no systematic mechanism that guides the integration of analyses. There is a strong need of a theoretical foundation to explain when and how different analyses can be integrated, and how to ensure or, for the least, to determine the safety of the integrated analyses.

6 Conclusions

In this article, we have summarized recent advances and challenges in cooperative testing and analysis, a new research frontier on synergistic cooperations between humans and tools along with cooperations between tools and tools. Such cooperations are motivated by the observation that a tool is typically not powerful enough to address complications in testing or analysis of complex real-world software, and the tool user or another tool may be able to help out some problems faced by the tool. In cooperative testing analysis, tool-human cooperation 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. An example approach of human-assisted computing is Covana[4], an approach that precisely identifies problems faced by test-generation tools to reduce human efforts in providing guidance to the tools. 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. An example approach of human-centric computing is Pex4Fun[22-24], a web-based serious gaming platform for teaching and learning programming and software engineering. Tool-tool or analysis-analysis cooperation is often in the form of tool or analysis integration. An example approach of analysis-analysis cooperation is collaborative hybrid analysis[35] that automates presentation changes in dynamic web applications.

References

[1] Xie T, Tillmann N, de Halleux J, Schulte W. Future of developer testing: Building quality in code. In *Proc. the 18th FSE/SDP Workshop on the Future of Software Engineering Research*, Nov. 2010, pp.415-420.

[2] Xiao X, Thummalapenta S, Xie T. Advances on improving automation in developer testing. In *Advances in Computers*, volume 85, Memon A (ed.), Burlington: Academic Press, 2012, pp.165-212.

[3] D'Silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2008, 27(7): 1165-1178.

[4] Xiao X, Xie T, Tillmann N, de Halleux J. Precise identification of problems for structural test generation. In *Proc. the 33rd Int. Conf. Software Engineering*, May 2011, pp.611-620.

[5] Dillig I, Dillig T, Aiken A. Automated error diagnosis using abductive inference. In *Proc. the 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 2012, pp.181-192.

[6] Xie T. Cooperative testing and analysis: Human-tool, tool-tool, and human-human cooperations to get work done. In *Proc. the 12th Int. Working Conf. Source Code Analysis and Manipulation*, Sept. 2012, pp.1-3.

[7] Hellerstein J L, Diao Y, Parekh S, Tilbury D M. Feedback Control of Computing Systems. John Wiley & Sons, 2004.

[8] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 2005, pp.213-223.

[9] Tillmann N, de Halleux J. Pex-white box test generation for .NET. In *Proc. the 2nd Int. Conf. Tests and Proofs*, April 2008, pp.134-153.

[10] Tillmann N, Schulte W. Mock-object generation with behavior. In *Proc. the 21st Int. Conf. Automated Software Engineering*, Sept. 2006, pp.365-368.

[11] Marri M R, Xie T, Tillmann N, de Halleux J, Schulte W. An empirical study of testing file-system-dependent software with mock objects. In *Proc. the 4th Int. Workshop Automation of Software Test*, May 2009, pp.149-153.

[12] Taneja K, Zhang Y, Xie T. MODA: Automated test generation for database applications via mock objects. In *Proc. the 25th Int. Conf. Automated Software Engineering*, Sept. 2010, pp.289-292.

[13] St-Amour V, Tobin-Hochstadt S, Felleisen M. Optimization coaching: Optimizers learn to communicate with programmers. In *Proc. the 3rd ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2012, pp.163-178.

[14] Dincklage D, Diwan A. Explaining failures of program analyses. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 2008, pp.260-269.

[15] Lakhotia K, McMinn P, Harman M. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 2010, 83(12): 2379-2391.

[16] Godefroid P, Levin M Y, Molnar D A. Automated whitebox fuzz testing. In *Proc. Network and Distributed System Security Symp.*, Feb. 2008.

[17] Xie T, Tillmann N, de Halleux P, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. the 39th IEEE/IFIP Int. Conf. Dependable Systems and Networks*, July 2009, pp.359-368.

[18] Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test generation. In *Proc. Int. Symp. Software Testing and Analysis*, July 2011, pp.23-33.

[19] Saxena P, Poosankam P, McCamant S, Song D. Loop-extended symbolic execution on binary programs. In *Proc. the 18th Int. Symp. Software Testing and Analysis*, July 2009, pp.225-236.

[20] Strejček J, Trtík M. Abstracting path conditions. In *Proc. Int. Symp. Software Testing and Analysis*, July 2012, pp.155-165.

[21] Xiao X, Li S, Xie T, Tillmann N. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. the 28th Int. Conf. Automated Software Engineering*, Nov. 2013, pp.246-256.

[22] Tillmann N, de Halleux J, Xie T. Pex4Fun: Teaching and learning computer science via social gaming. In *Proc. the 24th IEEE-CS Conf. Software Engineering Education and Training*, May 2011, pp.546-548.

[23] Tillmann N, de Halleux J, Xie T, Gulwani S, Bishop J. Teaching and learning programming and software engineering via interactive gaming. In *Proc. the 35th Int. Conf. Software Engineering, Software Engineering Education*, May 2013, pp.1117-1126.

[24] Tillmann N, de Halleux J, Xie T, Bishop J. Pex4Fun: A web-based environment for educational gaming via automated test generation. In *Proc. the 28th Int. Conf. Automated Software Engineering*, Nov. 2013, pp.730-733.

[25] Hao D, Zhang L, Xie T, Mei H, Sun J. Interactive fault localization using test information. *Journal of Computer Science and Technology*, 2009, 24(5): 962-974.

[26] Hao D, Zhang L, Zhang L, Sun J, Mei H. VIDA: Visual interactive debugging. In *Proc. the 31st Int. Conf. Software Engineering*, May 2009, pp.583-586.

[27] Xie T, Tillmann N, de Halleux J. Educational software engineering: Where software engineering, education, and gaming meet. In *Proc. the 3rd International Workshop on Games and Software Engineering*, May 2013, pp.36-39.

[28] Fincher S, Cooper S, Kölling M, Maloney J. Comparing Alice, Greenfoot & Scratch. In *Proc. the 41st ACM Technical Symp. Computer Science Education*, March 2010, pp.192-193.

[29] Utting I, Cooper S, Kölling M, Maloney J, Resnick M. Alice, Greenfoot, and Scratch – A discussion. *ACM Trans. Computing Education*, 2010, 10(4): 17:1-17:11.

[30] Tillmann N, de Halleux J, Xie T, Bishop J. Code Hunt: Gamifying teaching and learning of computer science at scale. In *Proc. the 1st ACM Conf. Learning at Scale*, March 2014, pp.221-222.

[31] Tillmann N, Bishop J, Horspool N, Perelman D, Xie T. Code Hunt - Searching for secret code for fun. In *Proc. the 7th Int. Workshop Search-Based Software Testing*, June 2014.

[32] Dwyer M B, Elbaum S G. Unifying verification and validation techniques: Relating behavior and properties through partial evidence. In *Proc. the 18th FSE/SDP Workshop on the Future of Software Engineering Research*, Nov. 2010, pp.93-98.

[33] Ge X, Taneja K, Xie T, Tillmann N. DyTa: Dynamic symbolic execution guided with static verification results. In *Proc. the 33rd Int. Conf. Software Engineering*, May 2011, pp.992-994.

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

[35] Wang X, Zhang L, Xie T, Xiong Y, Mei H. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proc. the 20th ACM SIGSOFT Symp. Foundations of Software Engineering*, Nov. 2012, pp.16:1-16:11.

[36] Csallner C, Smaragdakis Y, Xie T. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Software Engineering and Methodology*, 2008, 17(2): 8:1-8:37.

[37] Thummalapenta S, Xie T, Tillmann N, de Halleux J, Su Z. Synthesizing method sequences for high-coverage testing. In *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2011, pp.189-206.

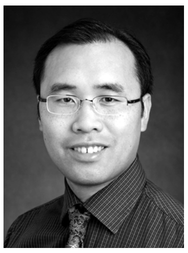
[38] Inkumsah K, Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. the 23rd Int. Conf. Automated Software Engineering*, Sept. 2008, pp.297-306.

[39] Xiong Y, Liu D, Hu Z, Zhao H, Takeichi M, Mei H. Towards automatic model synchronization from model transformations. In *Proc. the 22nd Int. Conf. Automated Software Engineering*, Nov. 2007, pp.164-173.

[40] Weiser M. Program slicing. In *Proc. the 5th Int. Conf. Software Engineering*, March 1981, pp.439-449.

[41] Gong L, Lo D, Jiang L, Zhang H. Interactive fault localization leveraging simple user feedback. In *Proc. the 28th IEEE Int. Conf. Software Maintenance*, Sept. 2012, pp.67-76.

[42] Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. In *Proc. the 31st Int. Conf. Software Engineering*, May 2009, pp.364-374.



Tao Xie is an associate professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, USA. He received his Ph.D. degree in computer science from the University of Washington in 2005. Before that, he received an M.S. degree in computer science from the University of Washington in 2002 and an M.S. degree in computer science from Peking University in 2000. His research interests are in software engineering, with an emphasis on software testing, program analysis, and software analytics.

Lu Zhang is a professor at the School of Electronics Engineering and Computer Science, Peking University, Beijing. He received his B.S. and Ph.D. degrees in computer science from Peking University in 1995 and 2000, respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. His research interests include software testing, software analysis, program comprehension, software maintenance, software reuse, and service computing.

Xusheng Xiao is a researcher at NEC Laboratories America. He received his Ph.D. degree in computer science from North Carolina State University, and was a visiting student in computer science at the University of Illinois at Urbana-Champaign, USA. Before starting his Ph.D. program, he was a consultant/developer who specialized in agile software development in ThoughtWorks. His research in software engineering focuses on improving cooperation between tool users and software testing and analysis tools.

Ying-Fei Xiong is an assistant professor under the Young Talents Plan at the School of Electronics Engineering and Computer Science, Peking University, Beijing. He got his Ph.D. degree in computer science from the University of Tokyo in 2009 and worked as a postdoctoral fellow at University of Waterloo between 2009 and 2011. His research interest is software engineering and programming languages.

Dan Hao is an associate professor at the School of Electronics Engineering and Computer Science, Peking University, Beijing. She received her Ph.D. in computer science from Peking University in 2008. Her current research interests include software testing, debugging, and program comprehension.