Rethinking the Economics of Software Engineering

Todd W. Schiller
Computer Science and Engineering
University of Washington
Seattle, Washington
tws@cs.washington.edu

Michael D. Ernst
Computer Science and Engineering
University of Washington
Seattle, Washington
mernst@cs.washington.edu

Abstract

Reliance on skilled developers reduces the return on investment for important software engineering tasks such as establishing program correctness. This position paper introduces *adaptive semi-automated* (ASA) tools as a means to enable less-skilled workers to perform aspects of software engineering tasks. In an ASA tool, a task is decomposed and the computationally difficult subtasks are performed by less-skilled workers using an adaptive user interface, reducing or eliminating the skilled developer's effort.

We describe strategies for decomposing a software engineering task and propose design principles to maximize the cost effectiveness of ASA tools in the presence of imperfect decomposition. Though the approach can be applied to many different types of tasks, this paper focuses on and provides examples for the software correctness tasks of test generation, program verification, and program synthesis. Additionally, we address the auxiliary challenges of latency, intellectual property risk, and worker error.

Categories and Subject Descriptors
K.6.0 [**Management of Computing and information systems**]: General—*Economics*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms
Economics

Keywords
adaptive semi-automation, less-skilled, test generation, program synthesis, program verification

1. Introduction

Software bugs cost the United States economy an estimated $59.5 billion annually; it is estimated that improved testing practices could reduce this cost by $22.5 billion [13]. The high cost of software bugs does not imply a lack of attention to testing, however. Beizer found that 50% of the labor when developing software is spent on testing [4].

The labor-intensive nature of testing can be explained in large part by the fact that, despite the existence of software engineering tools, skilled developers must still manually write test cases, write program annotations, refactor software, etc. Since software developers in the United States earn a mean annual salary of $90,170 [17], the expected return on investment (ROI) for exhaustively performing tasks like test generation is often unattractive.

This paper presents *adaptive semi-automation* (ASA) as a strategy for reducing the cost of software engineering by shifting the labor burden to less-skilled, less-expensive, workers. Depending on the context, a less-skilled worker might have only a bachelor's degree or might have never attended college. ASA tools are distinct from traditional software engineering tools in that they have both semi-automated and adaptive aspects.

Semi-automation, the mixture of computation and effort by at least one non-primary user, is well-motivated by research into utilizing human knowledge to solve computationally difficult problems. Examples include reCAPTCHA for character recognition [19], the ESP Game for image labeling [18], and FoldIt for protein folding [6]. Paid services, such as Amazon's Mechanical Turk [2], have also emerged to provide a marketplace for people to perform small human intelligence tasks (HITs) at competitive rates.

In the software engineering domain, however, it may be impossible to decompose the tasks so cleanly as to decisively leverage human strengths. Therefore, the interaction with ASA tools is necessarily adaptive — the user input and tool feedback (outputs) are serially dependent — to enable less-skilled workers to efficiently perform subtasks. Section 3 establishes what we see as the fundamental design principles guiding the development of ASA tools.

Ultimately, the major challenge is how to decompose high-level tasks in order to make the best use of less-skilled labor. For example, an ASA test generator might use less-skilled workers to guide procedure call ordering. Whether the tool should also solicit inputs from the workers is ultimately a question of economics. In Section 2, we describe the microeconomic principles of opportunity cost and comparative advantage that serve as a basis for answering this question. Later, Section 3.1 prescribes specific research questions that can be used to analyze the economics of ASA tools.

The use of ASA tools also introduces auxiliary challenges such as latency, intellectual property protection, and user error. We address these challenges in Section 5. Determining the risk these challenges pose under the aforementioned economic framework allows tool designers and researchers to determine the financial feasibility of their tools and make a strong business case for their tool, if one exists.

The ASA approach, together with the low cost of on-demand computing and globalized labor markets, has the potential to increase the return on investment for software engineering tasks. Therefore, ASA tools have the potential to change the economics of software engineering, whether the goal is improved software reliability, usability, or correctness.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

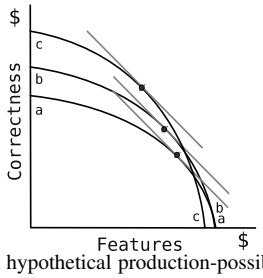

Figure 1: Curve (a) is a hypothetical production-possibility frontier (PPF) given fixed resources. Each axis is the present value of the associated cash flow measured in dollars (e.g., expected sales minus bug fix cost). The total value of an allocation at a particular level of correctness and features is the sum of the value of the established correctness (e.g., future bug fix costs avoided) and the value of the features. Maximum value is achieved where the slope is -1, as marked. Frontiers (b) and (c) show how introducing a new correctness tool changes the PPF. In (b), the developers use the tool. In (c), management re-balances the workforce, reducing the number of skilled workers and hiring less-skilled, less expensive, workers to use the new tool.

2. Opportunity Cost and Comparative Advantage

Opportunity cost is the cost associated with a limited resource, such as time. For example, if a company's best software developer is assigned to write unit tests, she is unable to write new high-value features. Because not all individuals are equally proficient in all endeavors, opportunity costs differ across individuals. An individual who has the lowest opportunity cost for a particular task has a comparative advantage. It is important to note that having a comparative advantage does not imply that the individual is the most proficient at the task (has an absolute advantage). For example, a summer intern may provide a comparative advantage in testing over a senior software developer.

Aggregate opportunity costs can be expressed as a production-possibility frontier (PPF). Figure 1, curve (a), shows a hypothetical production-possibility frontier for a project where a fixed set of resources — time, money, employees, etc. — can either be used for feature development or to establish correctness. The total value of an allocation is the sum of the dollar value of the established correctness (e.g., future bug fix costs avoided) and the value of the features.

Though Figure 1 grossly simplifies the trade-offs when managing a project, it captures the fundamental notion of the marginal rate of transformation (MRT) between correctness and feature development. At any given point on the frontier, the MRT is the opportunity cost associated with choosing an extra unit of features over an extra unit of correctness and vice versa. Assuming risk-neutrality — that the firm doesn't take into account the probabilities of the cash flows, only their expected values — the firm will choose a combination such that the MRT $= -1$ (the enlarged points in Figure 1), where the ROI trade-off is equal for features and correctness. In the figure, the lines with slope -1 are indifference lines (also known as indifference curves): the expected revenue is equivalent anywhere on the curve. An allocation on one indifference curve is preferable to an allocation on an indifference curve that is closer to the origin.

Technological innovations shift the production-possibility frontier outward because more value can be created. When a new testing tool is introduced, for example, the value of establishing software correctness increases relative to developing new features, shifting the curve outward as in Figure 1, curve (b), and making a higher total value achievable. For technologies that encourage a shift to non-developer employees, the increase in absolute testing and verification ability is accompanied by a decrease in the ability to develop features, as in Figure 1, curve (c).

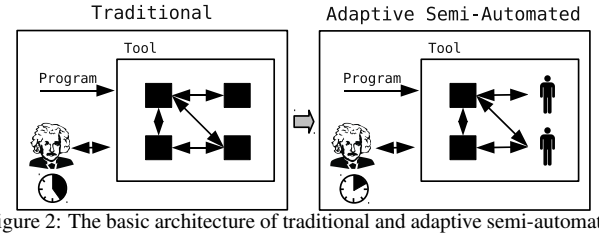

Figure 2: The basic architecture of traditional and adaptive semi-automated (ASA) tools. The square black boxes represent tool modules and the stick figures represent less-skilled workers. ASA tools use less-skilled workers to perform subtasks, reducing or eliminating the Einstein's interaction with the tool.

By increasing the ability for non-developers to establish correctness, adaptive semi-automated tools will cause a shift like that shown in Figure 1, curve (c). The idea is to increase the dollar value of correctness without significantly cannibalizing the dollar value of feature development, thus creating more value overall.

3. Adaptive Semi-Automation

This section describes the principles guiding task decomposition and the construction of adaptive user interfaces to enable less-skilled workers to perform the subtasks. In addition, it introduces three design principles for effectively integrating less-skilled labor. For ease of exposition, skilled developers will be referred to as "Einstein"s and less-skilled workers as "John"s (in reference to the American idiom "John Q. Public").

The software engineering task should be decomposed into automated and human subtasks as depicted in Figure 2. Each subtask that a John performs should (1) be easier to perform than the larger task, (2) require human insight, and (3) be limited in scope. We provide example decompositions for software correctness tasks in the next section.

Once the task has been decomposed, an adaptive feedback loop needs to be constructed for a John to interact with. In general, the loop should introduce subtasks that build on the subtasks that the John has already completed. Additionally, the loop might help the John avoid retracing his own steps or direct him to a part of the task-space that is computationally more difficult (and therefore less likely for the tool to automatically solve in a reasonable amount of time). Regardless of the primary purpose of the loop, clear feedback is necessary for the John to understand the consequences of his actions and to complete actions with minimal insight or understanding of the tool.

To improve the cost-effectiveness of the less-skilled labor, we propose the following three design principles:

Principle #1: Target a well-defined skill set User-performed subtasks should target a specific, well-defined skill set. By targeting a specific skill set, the cost of finding a qualified John is reduced. In addition, the use of a consistent skill set helps to ensure that continued use of the tool will make the John more efficient.

Principle #2: Exploit parallelism An hour of on-demand computing on Amazon's EC2 cloud service costs as little as $0.085 [1]. It's clear that a task should be automated when it can. However, if the latency of the automated portion of the feedback loop is large, a John could waste time waiting for a response. Similarly, if the latency of the semi-automated tool is large, an Einstein could waste time waiting for a response. Both result in an unfavorable ROI. Therefore, it is important to exploit parallelism in two ways: (1) have multiple Johns work on independent subtasks at the same time, and (2) have each John perform similar subtasks when waiting for a response. Essentially, each John is acting as a processor and therefore many common parallelization tactics can be

employed. However, humans are far worse at shifting contexts, so careful thought must be given to how tasks are distributed.

Principle #3: Establish a market Schechter theorized that markets in which companies purchase defect reports (for previously unknown defects) from testers could be used to measure and improve software quality [15]. An efficient market would ensure that (1) the price paid for each defect is minimized, (2) defects are reported as quickly as possible, and (3) easier, cheaper-to-fix, defects are reported earlier. Bacon et al. propose a similar marketplace in which developers bid to fix bugs [3]. When the market clears, the remaining bugs are not important enough for users to pay developers to fix them.

By utilizing similar markets (e.g., Amazon's Mechanical Turk [2]), tasks are naturally performed by individuals with the largest comparative advantage. Depending on the needs of the tool and the company, a John might perform correctness tasks for a company over the course of five years or over five minutes; a John might work in-house or halfway around the globe. Since ASA tools allow for less-skilled labor, the market can be much more efficient than one composed solely of skilled developers.

3.1 Tool Analysis

The set of research questions that should be considered when building adaptive semi-automated tools is similar to that used when evaluating traditional tools:

- What skill sets are required for the Einstein and John to use the tool?
- What is the learning curve for the Einstein and John to use the tool?
- Given a task and information such as the salaries of the Einstein and John, how much time and money is saved by using the tool?
- Given a fixed resource budget, can the tool provide benefits (e.g., preventing bugs) beyond those provided by other methods?

Small changes to the tool and interface may result in large changes to the efficacy of the tool when used by different groups (sensitivity). In addition, answering such questions becomes more difficult when multiple roles are introduced — i.e., a semi-automated tool, a software developer, and a less-skilled worker. For example, if the Einstein spends an extra 10 minutes, will the John and the computer finish three hours faster? By and large, analyzing an ASA tool means characterizing a multi-factor optimization problem, the solution to which is the most cost effective strategy for using the tool given a fixed resource budget.

4. Examples

This section sketches three possible adaptive semi-automated tools for testing, verification, and program synthesis. For each example tool, we address why the humans are well-suited to perform the task and why it is likely that using less-skilled labor would be more cost-effective than using skilled developers.

4.1 Testing

Random test generators, such as Randoop [11], perform poorly because they are unlikely to put complex objects into interesting states. For example, when testing a graph ADT, many of the generated tests will be performed on simple, uninteresting graphs because not enough edge and node creation calls come before the procedure being tested. An ASA testing tool programs might leverage procedure call ordering suggestions from a John.

Design Principles The subtasks can be easily parallelized by running multiple instances of the test generator. A market could be created for achieving various targets, such as branch coverage.

The John's Edge The user's understanding of the concept of a graph should give the ASA tool an advantage over automated tools. More generally, ASA tools stand to be the most effective for object-oriented programs where the objects have a natural analog (e.g., graphs, trees, cars).

The John's edge lies in the simplicity of the task presented to the user. Since the tool is taking care of the bulk of the detail work (e.g., input generation), the John only needs to have a high-level understanding of the code. This makes the skill requirement, and therefore the cost, much lower. Use of a John also avoids any costs associated with maintaining morale or employee retention when the skilled developers are assigned to such insipid tasks.

4.2 Verification

Consider the task of verifying a program with ESC/Java [9]. An adaptive semi-automated solution to the problem might involve the Johns iteratively refining a set of invariants inferred with a static- and dynamic-inference tool (such as Houdini [8] and Daikon [7], as in [10]).

Design Principles To limit the required skill set for a particular John, the different verification tasks, e.g., verifying the absence of null pointer or division by zero exceptions, should be kept as separate as possible. Parallelization can then be realized based on module and verification type. A marketplace could be established for potential Johns to bid on procedures and modules to verify; the specialization enabled by the splitting of verification job types will drive down costs.

The John's Edge The level of abstraction is key. Clearly, an automated system would outperform a John at solving an instance of SAT. However, a John with even a basic mathematical background can provide higher-level guidance such as suggesting relevant theories (e.g., quadratic arithmetic) or positing intermediary invariants. In addition to the John's own mathematical insight, the John can take advantage of comments in the program and the lexical structure of the program.

There is anecdotal evidence that suggests individuals with relatively basic computer science knowledge can successfully complete certain verification tasks. In our spring 2010 Software Design and Implementation course, the first-year computer science students used the Java Checker Framework [12] to verify the absence of null pointer exceptions in their individual class projects. In addition to the majority of the students completing the assignment without erroneously bypassing the checker, we observed that, in general, the students became more efficient at verifying program modules over the course of the weeklong assignment.

4.3 Synthesis

Testing and verification are reactive — they verify the correctness of existing code. An alternative approach is to create software that is correct by construction using program synthesis tools. While recent advances in program synthesis are exciting, general program synthesis is beyond the state of the art.

Consider Armando Solar-Lezama's program sketching work in which an Einstein provides a program sketch and a reference implementation or specification [16]. There are numerous ways to allocate the Einstein's and John's labor throughout the process of: (1) creating a rough specification, (2) programming a reference implementation, (3) writing a program sketch, (4) refining the program sketch and generating the program, (5) selecting which implementation to use if the sketch allowed multiple implementations. One

reasonable labor allocation would be to have the Einstein perform steps 1, 2, and 5 and provide an initial set of hints to the Johns for steps 3 and 4. The existence of a reference implementation after step 2 allows the Einstein to work on other engineering tasks that depend on a working implementation.

Design Principles One way to control the target skill set of the Johns would be to differentiate functions based on function characteristics (e.g., iterative versus recursive). While parallelization is easily achieved by having different Johns work on different functions at the same time, the parallel relationship of the Johns to the automated modules of the tool are more interesting. Ideally, the tool should heuristically search the sketch space while the John is interacting with the tool. The insight gained from the parallel heuristic search can then be used to guide the John via feedback in the user interface.

The John's Edge Significant human interaction is required for current program synthesis tools to be tractable. The Johns should be more cost effective than the Einstein because once the primary structure of the program has been determined, the bulk of the effort comes from fighting with the tool. Given that the Einstein can give the John hints as to primary structure of the program, the challenge lies in becoming familiar with the particular tool being used. The price of Johns familiar with the tool should be less than the cost of the skilled developer. In any case, the correctness of the resulting code is necessarily verifiable, limiting the risk of mistakes by a John.

5. Auxiliary Challenges

Latency, intellectual property risk, and worker error are three major challenges facing adaptive semi-automation.

5.1 Latency

Speed is important in software engineering — e.g., a type-checker should not take two days to run. It is less clear how long it should take to automatically generate a test suite or verify the absence of run-time exceptions. The software engineering community needs to define what it means for a tool to be "tractable." This will enable engineers to create a tool that companies want to use or to understand why a tool isn't being used.

Two approaches for mitigating latency are scheduling and exploiting time-zone differences. For example, in the case of scheduling, companies typically schedule long-running automated processes such as builds at night when most employees aren't working. American companies leverage time-zone differences when out-sourcing to countries such as India.

5.2 Intellectual Property Risk

Granting workers access to source code can pose a significant intellectual property (IP) risk, especially if employees are located in different countries.

Two approaches for mitigating intellectual property risk are employing workers in-house (or in a subsidiary when offshoring [14]) and restricting worker access. The former is straightforward; the latter is not — the problem must be decomposed in such a way that a John (or a group of Johns) cannot reassemble important aspects of the system.

Limiting a John's access, in general, will undermine the effectiveness of the tool by limiting the availability of requisite information. Therefore, it is important to find the equilibrium point where the IP risk and benefit of broadening the John's access are balanced. As this point is different for each company and each tool, the negative effects of limiting a John's view are not necessarily a deal breaker.

5.3 Worker Error

When there is reason to mistrust the abilities or the intentions of the human component, special care must be taken. ASA tools that utilize unfiltered and ad hoc labor from global labor markets are especially at risk.

In the extreme, the domain should be decomposed into subtasks that can be checked automatically. This notion is similar to the NP complexity class, for which there are efficient algorithms to verify certificates (solutions), but not to create them. For example, intermediary assert statements that the John supplies for use in program verification can be checked automatically. When the John's work cannot be checked, it may be necessary to utilize a fusion mechanism (e.g., a majority vote) that combines his answer with that of other Johns. More advanced fusion mechanisms can automatically adapt to changes in reliability [5]. The need for redundancy may increase costs, however.

6. Conclusion

This paper has introduced adaptive semi-automated tools as a method of decreasing software engineering labor costs. ASA tools differ from traditional tools in that they decompose tasks into computer and human elements that allow less-skilled workers to perform tasks via adaptive feedback loop. The software engineering workforce is already somewhat specialized with specific roles — e.g., developers and testers — but we propose special tools to permit utilization of a much wider range of human talent.

Work on ASA tools should proceed in parallel with research on fully automated tools. The practice of decomposing domains may shed light on new strategies for reaching full automation.

Establishing correctness via ASA tools and less-skilled users is not a perfect strategy. In particular, the reliance on less-skilled workers may result in increased latencies, intellectual property risk, and user error. These concerns can be controlled by means similar those used in other domains.

Overall, adaptive semi-automated tools expand the set of tasks that can be effectively performed by less-skilled workers, freeing skilled developers to create value in ways that only they can.

Acknowledgments We thank David Notkin, Colin Gordon, and the anonymous reviewers for their feedback. This work was supported in part by IBM through a John Backus Award.

7. References

[1] Amazon EC2 pricing, June 2010. http://aws.amazon.com/ec2/pricing/ .

[2] Amazon Mechanical Turk, May 2010. https://www.mturk.com/ .

[3] D. F. Bacon, Y. Chen, D. Parkes, and M. Rao. A market-based approach to software evolution. In *OOPSLA Companion*, pages 973–980, Oct. 2009.

[4] B. Beizer. *Software Testing Techniques*. Boston: International Thomson Computer Press, 1990.

[5] Y. Brun. *Self-Assembly for Discreet, Fault-Tolerant, and Scalable Computation on Internet-Sized Distributed Networks*. PhD thesis, University of Southern California, 2008.

[6] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popovic, and F. Players. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, August 2010.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon

system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1–3):35–45, Dec. 2007.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, pages 500–517, Mar. 2001.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, June 2002.

[10] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE*, pages 11–20, Nov. 2002.

[11] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, May 2007.

[12] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[13] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. NIST Planning Report 02-3, National Institute of Standards and Technology, May 2002.

[14] S. Sakthivel. Managing risk in offshore systems development. *CACM*, 50(4):69–75, Apr. 2007.

[15] S. Schechter. How to buy better testing: Using competition to get the most security and robustness for your dollar. In *Infrastructure Security Conference*, pages 73–87, Bristol, UK, Oct. 2002. Springer.

[16] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[17] U.S. Bureau of Labor Statistics. Computer software engineers, applications, May 2010. `http://www.bls.gov/oes/current/oes151031.htm` .

[18] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *CHI*, pages 319–326, Apr. 2004.

[19] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, Sep. 2008.