A Framework for Incentivizing Deep Fixes

Malvika Rao
SEAS, Harvard University
malvika@eecs.harvard.edu

David C. Parkes
SEAS, Harvard University
parkes@eecs.harvard.edu

Margo Seltzer
SEAS, Harvard University
margo@eecs.harvard.edu

David F. Bacon
IBM Research
bacon@us.ibm.com

Abstract

We study the problem of how to incentivize deep fixes to software bugs, where a deep fix attempts to correct the root cause of the bug instead of just suppressing it superficially. To this end we introduce a dynamic model of the software engineering ecosystem. We then solve this problem by proposing *subsumption mechanisms*. In a subsumption mechanism, deeper fixes can replace or subsume shallower fixes and a worker's payoff increases if his fix subsumes other fixes. We use a solution concept known as mean field equilibrium, an approximation methodology suited to large market settings. Taking a computational approach, we simulate the dynamic model of the ecosystem with subsumption mechanisms. Our algorithm achieves convergence and thus estimates a mean field equilibrium. We further compare our mechanism to baseline mechanisms using metrics, such as percentage of bugs receiving deep fixes, rate of bugs fixed, and cost to the user. Simulation results indicate that the subsumption mechanism performs favourably versus the baseline mechanisms.

Introduction

The size and complexity of software systems have increased to such an extent that it is beyond our ability to effectively manage them. A study commissioned by the U.S. National Institute of Standards and Technology concluded that software errors alone cost the U.S. economy approximately $\$59.5$ billion annually (NIST 2002). Software often ships with discovered as well as undiscovered bugs because there are simply not enough resources to address all issues (Liblit et al. 2003), (Anvik, Hiew, and Murphy 2006). Through an empirical study of 277 coding projects in 15 companies, Wright and Zia (Wright and Zia 2011) determine that software maintenance actually introduces more bugs: each subsequent iteration of fixes has a $20 - 50\%$ chance of creating new bugs.

In fact software systems have come to resemble systems where behaviour is decentralized, interdependent, and dynamic – rather like economies. This suggests that the principles of market and mechanism design might be more effective in managing software than traditional software engineering techniques. In previous work (Bacon et al. 2010) we described our vision for a software development process, inspired by economies, where supply and demand drive the allocation of work and the evolution of the system. *Software economies* consist of a private and a public component. A private software economy deals with the internal incentives of managers and their employees, such as allocating resources and predicting completion times. On the other hand a public software economy refers to a market mechanism where users bid for coveted bug fixes and features. Over time the software reaches an equilibrium where all fixes and features for which there is enough market value have been implemented. However designing such a market mechanism is fraught with challenges. This work addresses the public software economy and is a response to a challenge raised in (Bacon et al. 2009).

Specifically the problem we are interested in is how to design incentives to obtain "deep" rather than "shallow" fixes to bugs. A deep fix attempts to correct the root cause of the problem so that another bug with the same root cause is found only after a long time or not at all. In contrast a shallow fix suppresses the bug at a superficial level so that other bugs with the same root cause appear soon after. We solve this problem by proposing *subsumption mechanisms*. In a subsumption mechanism, deeper fixes can replace or *subsume* shallower fixes and a worker's payoff increases if his fix subsumes other fixes. The mechanism employs an instalment-based payment rule that stops paying the worker when his fix is subsumed and transfers the remaining reward to the competitor responsible for the deeper fix.

To this end we design a dynamic model of the software engineering ecosystem comprising workers, users, root causes of bugs, bugs, fixes, time periods, rewards, and payments. The user base reports bugs and offers reward money for fixes. A worker assigned a bug decides which fix to submit subject to cost and payment. The paradigm we adopt is of restricting our attention to those features of the software development process that are *externally observable*. In other words we use only the information that can be observed by the market, and refrain from delving into the internal workings of the software such as the structure and composition of the code. Some externally observable features are: time taken for the next bug to appear, number of bugs fixed, and the amount of reward money. Our framework is novel and can be viewed as building blocks that can be reconfigured to

explore different questions in this ecosystem.

Our technical approach is to frame our problem in the context of an approximation methodology, useful to analyze the behaviour of large systems, called *mean field equilibrium* (MFE). In large markets, it is intractable and implausible for individuals to best-respond to competitors' exact play (Adlakha, Johari, and Weintraub 2011). Instead it is assumed that in the long run fluctuations in agents' actions "average out". Hence agents optimize with respect to long run estimates of the distribution of other agents' actions. Using this methodology we simulate the dynamic model of the software engineering ecosystem. In the case of subsumption mechanisms, our algorithm achieves convergence and is thus able to estimate an MFE. We then compare the subsumption mechanism to two other mechanisms that do not involve subsumption. For this purpose we consider metrics such as the percentage of bugs receiving deep fixes, *coverage rate*, and user cost. Coverage rate refers to how many bugs are fixed per time period on average, and user cost refers to the total monetary amount spent by users towards these fixes. Preliminary results show that the subsumption mechanism produces deeper fixes and consequently achieves higher coverage rate for lower user cost than the other mechanisms.

Related work

Research into vulnerability reporting systems has explored a market-based approach. Schechter (Schechter 2002) describes a vulnerability market where a reward is offered to the first tester that reports a specific security flaw. The reward grows if no one claims it. At any point in time the product can be considered secure enough to protect information worth the total value of all such rewards being offered. Ozment (Ozment 2004) likens this type of vulnerability market to an open first price ascending auction. While the vulnerability market as well as existing bug bounty programs (e.g. Mozilla security bug bounty) motivate testers to report flaws, such systems do not capture users' valuations for fixes.

Le Goues *et al.* (Goues, Forrest, and Weimer 2010) share our view of software as an evolving, dynamic process. However where we approach software engineering from a mechanism design perspective, they are influenced by biological systems. They apply genetic programming for automated code repair (Goues et al. 2012), (Weimer et al. 2010).

Methodologically our work is most closely related to a series of recent papers that analyze MFE in dynamic settings. Iyer *et al.* (Iyer, Johari, and Sundararajan 2011) consider a sequence of single-item second price auctions where bidders are still learning their private valuations. Gummadi *et al.* (Gummadi, Key, and Proutiere 2011) examine both repeated second price as well as generalized second price (GSP) auctions when bidders are budget constrained. Both papers establish existence of MFE and characterize the optimal bidding strategy. Other settings have also been analyzed in the mean field context (Adlakha and Johari 2010), (Gummadi, Johari, and Yu 2012).

Organizations that crowdsource software development exist [1]. A notable example is TopCoder (TopCoder Inc.)

[1]A survey of crowdsourced software platforms is presented in (Bacon et al. 2009).

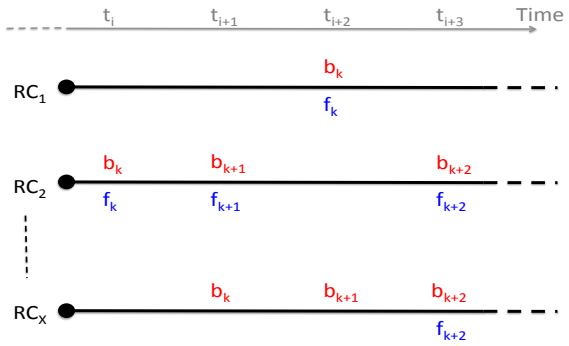

Figure 1: Root causes generate bugs which receive fixes.

where programmers compete in a contest with cash awards to submit the best solution to a software project. DiPalantino and Vojnovic (DiPalantino and Vojnovic 2009) and Chawla *et al.* (Chawla, Hartline, and Sivan 2012) study crowdsourcing contests like TopCoder and model them as all-pay auctions. However crowdsourcing contests are an altogether different scenario to ours.

BountySource (BountySource Inc.) is a funding platform for open-source software, where users post rewards on issues they want solved while developers devise solutions and claim rewards. GitHub (GitHub Inc.) is a code repository that allows programmers to work on portions or versions of the code by providing operations like forking, merging, and syncing. These seem like natural precursors to the externally observable market-based system proposed in this paper.

A system of bugs and fixes

We present an abstract model of the software as a set of independent *root causes*, where each root cause generates a series of related bugs. To draw an analogy with a real world scenario, a root cause may be thought of as a specific component or functionality of a software; for example, one root cause might be the user interface component, while another might be the graphics component.

Bit string representation A bit string model is used to capture how a particular root cause can generate several bugs. Each root cause is associated with a particular bit string length l. The set of bugs belonging to this root cause comprises the $2^l - 1$ non-zero bit strings of length l. The set of fixes that can address this set of bugs is represented by the same language – it consists of the set of 2^l bit strings, including the null string (i.e. the string with all 0's). The inclusion of the null string is to allow for the possibility that a bug may not receive a fix at all. Next we describe rules and relationships between bugs and fixes in our system.

Properties of bugs and fixes

Armed with the bit string language we proceed to define concrete properties relating bugs and fixes. First we note that, in our model, fixes pertaining to a particular root cause cannot be used to fix bugs generated by other root causes.

Thus all relationships and properties are relevant for only those bugs and fixes that belong to the same root cause. In what follows, we refer to a bit whose value is 1 as an ON-bit.

Definition 1. *A fix f fixes a bug b if it includes all the ON-bits in b. Thus an AND operation between f and b must result in b.* [2]

We refer to the set of fixes for a bug plus the null fix as the set of *feasible* fixes. Different bugs can have different numbers of feasible fixes. Bug 1110 has only 3 feasible fixes. In contrast bug 0001 has $2^3 + 1$ feasible fixes.

Example 1. *A root cause with $l = 4$ can generate the set of bugs $\{0001, 0010, \ldots, 1111\}$, where each bit string represents a single bug. Consider bug $b_i = 1110$. b_i is fixed by two fixes: $f_{i1} = 1110$ and $f_{i2} = 1111$. The entire set of feasible fixes for b_i is $\{1110, 1111, 0000\}$. However fix 0111 cannot fix b_i as they do not have the same ON-bits.*

Definition 2. *[Fix depth]*
The fix depth of f refers to the number of ON-bits in the bit string of f and is denoted $|f|$.

Continuing with the above example, f_{i1} and f_{i2} have fix depths equal to 3 and 4 respectively. We can now define, in the context of the bit string representation, what constitutes a shallow or deep fix with respect to a given bug.

Definition 3. *[Shallow fix]*
Given a bug b, a shallow fix f is one whose bit string is exactly identical to b, $|f| = |b|$.

Definition 4. *[Deep fix]*
Given a bug b of bit string length l, a deep fix f is a fix with $|f| > |b|$, for a maximum of l ON-bits.

In other words, a shallow fix is the fix that meets the essential requirement of having the same ON-bits as the bug being fixed and no more. The deepest fix not only fixes the bug in question but all bugs of that root cause.

Example 2. *Consider bug $b_j = 1001$ generated by a root cause with bit string length 4. A shallow fix for b_j is $f_{j1} = 1001$. A deeper fix for b_j would be $f_{j2} = 1011$ or $f_{j3} = 1101$. Notice that even f_{j1} could fix more than just b_j; for example it could fix bugs 0001 and 1000. The deepest possible fix would be $f_{j4} = 1111$, which fixes all possible bugs on the root cause.*

Externally Observable So far we have described how bugs and fixes relate to one another in the context of the bit string representation. Moving on, we consider those properties that are externally observable, and to that end introduce a key concept: subsumption.

Definition 5. *[Subsumption relation]*
A fix f_z subsumes another fix f_y ($f_z \succ f_y$) if the set of bugs fixed by f_z contains the set fixed by f_y.

Using example 2, suppose the root cause generates another bug $b_k = 1111$ The only possible non-null fix for b_k is $f_{k1} = 1111$. f_{k1} subsumes f_{j1} since f_{k1} fixes all the bugs fixed by f_{j1} as well as fixing b_j. In example 2, the subsumption relation is as follows: $f_{j1} \prec \{f_{j2}, f_{j3}\} \prec f_{j4}$, where fixes f_{j2} and f_{j3} do not subsume each other. A sequence $f_1 \prec f_2 \prec \cdots \prec f_x$ implies a partial order on the set of fixes pertaining to a root cause.

From an externally observable viewpoint, if f_z subsumes f_y, then f_z is considered to be a deeper fix than f_y.

[2] The null fix is the exception.

The model of the ecosystem

We are now ready to present the model of the entire ecosystem. We study a setting with discrete time periods $\{t_1, t_2, \ldots\}$ and an infinite population of workers who submit fixes. The software is used by a large, anonymous user base who discover bugs and offer reward money for fixes. The software consists of a fixed number X of root causes of bugs, $RC = \{RC_1, RC_2, \ldots, RC_X\}$, where all root causes are associated with the same bit string length l. The X root causes regenerate with probabilities $\{\beta_1, \beta_2, \ldots, \beta_X\}$. Consider a bug b_k generated by root cause RC_x. Reward money for a fix for b_k is denoted r_k and drawn according to a distribution. Suppose a worker w_j submits fix f_{ki} for b_k. To produce the fix w_j incurs cost c_j per ON-bit in f_{ki} (the total cost being $c_j|f_{ki}|$), where c_j is drawn from a distribution. The utility w_j derives from submitting fix f_{ki} for b_k is denoted y_{jki} and comes from the payments w_j receives.

Bug generation In each time period we sample uniformly at random from all $2^l - 1$ bit strings or bugs associated with a root cause, regardless of whether some of those bugs might be already fixed (i.e. sampling with replacement). A new bug enters the system only if we choose an unfixed and as yet unreported bug. This reflects a natural situation where a root cause may generate lots of bugs at the start but fewer and fewer as fixes accumulate.

Root causes that have not generated a new bug in a while (say the last $\bar{t}$ time periods, for some $\bar{t}$) are considered inactive. Let D_{RC} be the set of inactive root causes in the current time period. Each $RC_i \in D_{RC}$ is regenerated with probability β_i. This models the root cause as having received deep enough fixes that it is now hard to generate bugs. As a result the user base shifts its attention to a new set of bugs. When a root cause is regenerated it is removed and replaced with a new root cause (one with all bugs yet to be generated) of the same bit string length. The presence of multiple regenerating root causes ensures an infinite stream of bugs and provides the necessary conditions to model stationarity (needed for equilibrium analysis later in the paper).

Model dynamics

We consider a dynamic setting that consists of a sequence of *fix-verify cycles* that occur over time. Each fix-verify cycle takes place in a single time period t_i at a given root cause. Thus at time period t_i we "round-robin" around all X root causes, executing fix-verify cycles at each root cause. This process is repeated at time period t_{i+1}. A fix-verify cycle proceeds as follows.

1. The root cause is queried once to see if it generates a new bug, which is associated with a user reward.
2. A worker is assigned to a bug, where the set of bugs includes prior unfixed bugs as well as newly generated ones.

3. The worker submits a feasible fix that maximizes his expected utility.
4. The fix is verified by the market infrastructure.
5. The total reward to be paid to the worker is calculated.
6. The worker is paid if he submits a non-null fix. Other payments may be made depending on the payment rule.

We make three assumptions. First, user rewards are collected once when a new bug is generated and do not continue to accumulate over time. Second, the worker incurs a cost to produce a fix but that fix is submitted in the same time period the bug is assigned. Third, the likelihood that the same worker is repeatedly assigned bugs belonging to the same root cause is low.

Each worker works on one bug at a time and submits a fix before starting work on the next bug. Hence an individual worker works *sequentially*. Moreover, in any given time period, each root cause generates at most one new bug, at most one worker is assigned to an unfixed bug, and at most one bug is fixed [3]. While work *within* a particular root proceeds in sequential order, work *across* different roots may happen in parallel.

Subsumption mechanism

Consider an instantiation of the model where Step 4 of the fix-verify cycle involves a check for whether the current fix subsumes any previous fixes. If so, the subsumed fixes are discarded and replaced by the current fix. Suppose as well that the worker's payoff increases if his fix subsumes previous fixes. We refer to this instantiation as a *subsumption mechanism*. Here the worker faces his (indirect) competitors, who fix subsequent bugs on the same root cause, in sequential order. Thus the model captures *indirect* competition in the following way: another worker w_z might produce a deeper fix for a subsequent bug b_k, that not only fixes b_k but also subsumes w_y's fix for b_j. Subsumption mechanisms are the focus of the rest of the paper.

Payment scheme We concentrate on instalment-based payment schemes. Specifically in this instance, we use a simple payment rule that pays out equal instalments of the total reward over a fixed number of time periods, h^*. The total reward for a fix f_k for bug b_k is,

$$r_{total} = \sum_i \hat{r}_i^{prev} I_{\{f_k \succ f_i{}^{prev}\}} + \sum_q \bar{r}_q^{open} + r_k \qquad (1)$$

where $\hat{r}_i^{prev}$ is the remaining unpaid reward money of a previous fix $f_i{}^{prev}$ subsumed by f_k, and $\bar{r}_q^{open}$ is the reward money of an unfixed bug b_q that f_k has fixed. The worker is paid an instalment every time period until all instalments are exhausted or until the worker's fix is subsumed by a new fix, whichever occurs sooner. In the latter case, the remainder of the worker's reward is transferred to the subsuming fix. Hence if the worker's fix is only subsumed after h^* time periods have passed, or not subsumed at all during the lifetime of the root cause, the worker is paid r_{total} in its entirety. Note that an instalment-based payment rule requires any ongoing instalments to other workers to be paid in Step 6 of the fix-verify cycle.

[3]This process ensures that at most one fix is submitted per time period, per root cause. This is done to avoid situations where two or more overlapping or equivalent fixes might be submitted simultaneously.

The worker

In the current model the worker does not choose which bug to work on, rather he is assigned to a bug at random. Therefore the worker's decision problem is to determine which fix to submit, including the null fix to model a decision to do no work. In addition the worker's decision to work now does not impact his decision in the future, since fixes are produced instantaneously and the set of indirect competitors faced by the worker does not include himself.

We define the worker's utility function in the context of the subsumption mechanism. Worker w_j's expected utility for submitting f_k to fix b_k is equal to the sum of the payments he will receive, starting from the time period when he submits the fix, minus his cost for f_k:

$$E[y_{jk}] = E_h[\sum_{t=0}^{h} \frac{r_{total}}{h^*}\delta^t - c_j|f_k|] \qquad (2)$$

where δ is a discount factor applied to payment instalments received at later time periods, and $h = min(h^*, H)$ such that H is a random variable representing *subsumption time* (number of time periods before f_k gets subsumed by a later fix). w_j chooses the fix that maximizes his expected utility. Subsumption times vary amongst fixes. This is because each fix precludes a different set of bugs, which affects the time till the next bug (and its fix) appears. Moreover each fix permits a different set of future fixes to subsume it.

Mean field equilibrium

In order to understand the long-term behaviour of workers, we need to consider the equilibrium of the present system. Once equilibrium is established, performance metrics can be meaningfully measured, and a market designer may be better informed on conditions for desirable market behaviour.

Although our system is dynamic and complex, we can simplify equilibrium analysis by opting for a solution concept known as mean field equilibrium (MFE). MFE is an approximation methodology suited to large market settings, where keeping track of the strategies of individual agents becomes prohibitive. As the number of agents grows large, it is reasonable to assume that any individual agent has negligible effect on overall outcomes. Accordingly, MFE assumes that agents optimize with respect to long run estimates of the distribution of other agents' actions. MFE requires a consistency check: the latter distribution must itself arise as a result of agents' optimal strategies.

Applying MFE to our setting, we assume that each worker models the future as facing an i.i.d. distribution D of fix depths submitted by opponents, where the set of possible fix depths is $\{0, \ldots, l\}$. The worker assumes that all fixes associated with a particular fix depth occur with equal probability. This induces a probability distribution over the set of

all 2^l possible fixes for a root cause. Now that the worker knows the distribution over all fixes, he can infer the conditional distribution of feasible fixes for a specific bug. Given this, the worker chooses the fix that maximizes his expected utility by estimating subsumption time when opponents play according to D. Consequently a realization of fixes is generated every time period, which yields an empirical distribution on fix depths.

Let $\Phi(D)$ be the long-term empirical distribution that results when workers facing D apply their optimal strategy.

Definition 6. *We define an MFE as a distribution D of fix depths such that $\Phi(D) = D$.*

Simulation

We simulate the subsumption mechanism, with the goal of estimating an MFE by converging to it.

Algorithm to estimate MFE

1. Initial distribution on fix depths D_{n-1}, $n = 1$.
2. A worker is assigned bug b. For each feasible fix f_i, the worker samples possible future trajectories assuming that all other workers are playing according to the current distribution D_{n-1} and arrives at an estimate for h_{f_i}. He submits the fix ${f^*}_i$ whose expected subsumption time $h_{{f^*}_i}$ maximizes his total expected utility.
3. Let ${fd^*}_i$ be the fix depth of ${f^*}_i$. Update distribution D_{n-1} to D_n.
4. Stop when convergence criterion is met (i.e. the distribution no longer changes with updates).

Sampling Given a fix f_i submitted in the current time period, bugs not (preemptively) fixed by f_i might appear in future time periods. In order to compute the expected utility of submitting f_i, we simulate the future. This is done by running the mechanism repeatedly assuming f_i has been submitted, where each run is a possible future scenario. To simulate how future competitors will play when assigned a bug, the current distribution D_{n-1} is used. Several trajectories are sampled in order to realize subsumption time and arrive at an estimate of the utility the worker can expect if he submits f_i.

To determine convergence we compare the distribution of fixes in equilibrium in different epochs of time and use a likelihood ratio test.

Comparing mechanisms

We compare the subsumption mechanism with two alternate mechanisms that do not involve subsumption.

- In the "instalment mechanism", when the worker submits a fix, Step 4 of the fix-verify cycle simply makes a basic check that the fix is feasible. The worker is still paid in instalments, however the instalments stop as soon as a new bug of the same root cause appears. Here workers do not best-respond to competitors' play. They simply optimize with respect to the state of the environment.
- The "myopic mechanism" can be viewed as a baseline. The worker is paid in full as soon as he submits a non-null feasible fix.

	Subsumption Mechanism	Instalment Mechanism	Myopic Mechanism
Coverage	15	15	15
Coverage Rate	7.68	4.27	2.16
User Cost	32.37	38	59.71
# of Fixes	1	2	4
% Bugs with Deep Fixes	50	36.11	0

Figure 2: Experimental results with $l = 4$.

Metrics In evaluating the mechanisms, we examine the percentage of bugs that receive deep fixes, coverage rate, and user cost. All metrics are computed once the system has stabilized to an equilibrium. In particular, data is collected from a root cause just before it regenerates. The final metric values are calculated by averaging over all the individual metric values computed for each root cause during a certain length of time or epoch.

Coverage refers to how many bugs of a particular root cause, during its lifetime before regeneration, have been fixed by the current set of fixes. For a root cause RC with bit string length l, coverage lies in the range $\{0 \dots 2^l - 1\}$. Let F_{RC} denote the set of fixes submitted to RC. Let $B(f_i^{RC})$ denote the set of bugs $\in RC$ fixed by f_i^{RC}, where $f_i^{RC} \in F_{RC}$. Then $coverage(RC) = |\bigcup_i B(f_i^{RC})|$. Coverage rate is the average coverage achieved per time period. Given a root cause, it is calculated by normalizing coverage with the number of time periods elapsed during the root cause's life.

Let B_{RC} denote the set of bugs generated by RC. Let r_k denote the user reward for a fix for $b_k \in B_{RC}$. Then $user\ cost(RC) = \sum_{b_k \in B_{RC}} r_k$. In words, user cost refers to the total user reward collected for all the bugs reported in a root cause's lifetime before regeneration. The percentage of bugs with deep fixes is a straightforward measure of the mechanism's performance. We also consider the final quantity of fixes that a root cause ends up with. Recall that subsumption discards fixes as they become redundant whereas the other two mechanisms retain all fixes. Overall a good mechanism has high coverage rate, low user cost, high percentage of deep fixes, and small final quantity of fixes.

Experimental results

The simulation is implemented in Matlab and run on the Odyssey cluster supported by the Research Computing Group at Harvard University. Unless otherwise specified, we use the following settings: number of root causes $X = 10$, bit string length $l = 4$, number of instalments $h^* = 10$, worker's discount factor $\delta = 0.6$, and user reward $r_k \in [1, 100]$. With probability p worker cost is low ($c_j = 1$) and with probability $1 - p$ it is high ($c_j = 2$), $p \in [0, 1]$.

Under these settings we find that the algorithm for computing MFE under the subsumption mechanism converges in accordance with the test criterion. Next we obtain preliminary results on the performance of the mechanisms. We run the simulation with parameters as mentioned above but with $r_k \in \{10, 20\}$. Figure 2 summarizes the findings. In another run we change to the following settings, keeping all else the same: $l = 3$, $h^* = 5$, $\delta = 0.8$, and $r_k \in \{3, 7\}$ (Figure 3).

	Subsumption Mechanism	Instalment Mechanism	Myopic Mechanism
Coverage	7	6	7
Coverage Rate	3.39	2.44	1.49
User Cost	9.53	9.33	15.79
# of Fixes	1	1.3	2.89
% Bugs with Deep Fixes	41.67	31.84	0

Figure 3: Experimental results with $l = 3$.

What is happening here is that, at low to moderate reward values, the instalment mechanism can not afford deep fixes early in a root cause's lifetime. Therefore it either produces shallow fixes or allows unfixed bugs to accumulate. This has the effect of aggregating user reward for these unfixed bugs. After a time, the instalment mechanism is able to produce a deep fix thanks to the unclaimed user reward that has collected from prior unfixed bugs. On the other hand the subsumption mechanism can avail itself of additional rewards by subsuming fixes. This allows the subsumption mechanism to "subsidize" deeper fixes to earlier bugs, and therefore provide higher coverage faster. The user cost results are consistent with the fact that subsumption produces deeper fixes and hence the user has to spend less to get most bugs of a root cause fixed.

The myopic mechanism always submits the shallowest possible fix to a bug. Since the myopic mechanism pays the worker immediately in full, the user reward always exceeds the worker's cost for certain settings. At first glance the myopic mechanism seems to perform comparably to the instalment mechanism (e.g. coverage). However as user reward is increased the instalment mechanism produces deeper fixes whereas the myopic mechanism remains unresponsive. In the extreme case where user reward far exceeds worker cost, the instalment and subsumption mechanisms produce similar results with respect to the above metrics. Even without subsumption, instalment-based payment rules can still help.

Discussion and future work

In ongoing work we seek to unify the themes discussed in the previous section into a formal model of user utility that captures the user's monetary cost, wait time for a fix, and side-effects of fixes submitted. We also consider variations of the subsumption mechanism and the worker cost model introduced here. As a next step we are interested in relaxing the following present constraints: 1) Instead of workers being assigned to bugs, suppose workers choose which bug to work on. 2) How does the system behave when user rewards are allowed to increase over time? Further on, we would like to explore additional phenomena such as fixes that introduce new bugs and overlapping root causes.

Acknowledgements We thank Edo Airoldi, Yiling Chen, Alice Gao, John Lai, Greg Stoddard, Ming Yin, and the anonymous reviewers for helpful feedback.

References

Adlakha, S., and Johari, R. 2010. Mean field equilibrium in dynamic games with complementarities. In *Proceedings of the IEEE Conference on Decision and Control*, CDC'10.

Adlakha, S.; Johari, R.; and Weintraub, G. Y. 2011. Equilibria of dynamic games with many players: Existence, approximation, and market structure. In *Working Paper*.

Anvik, J.; Hiew, L.; and Murphy, G. C. 2006. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06.

Bacon, D. F.; Chen, Y.; Parkes, D. C.; and Rao, M. 2009. A market-based approach to software evolution. In *Proc. 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09.

Bacon, D. F.; Bokelberg, E.; Chen, Y.; Kash, I. A.; Parkes, D. C.; Rao, M.; and Sridharan, M. 2010. Software economies. In *Proc. FSE/SDP Workshop on Future of Software Engineering Research*.

Bountysource inc. website. *https://www.bountysource.com*.

Chawla, S.; Hartline, J. D.; and Sivan, B. 2012. Optimal crowdsourcing contests. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12.

DiPalantino, D., and Vojnovic, M. 2009. Crowdsourcing and all-pay auctions. In *Proceedings of the 10th ACM Conference on Electronic Commerce*, EC '09.

Github inc. website. *https://github.com*.

Goues, C. L.; Dewey-Vogt, M.; Forrest, S.; and Weimer, W. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12.

Goues, C. L.; Forrest, S.; and Weimer, W. 2010. The case for software evolution. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10.

Gummadi, R.; Johari, R.; and Yu, J. Y. 2012. Mean field equilibria of multiarmed bandit games. In *Proceedings of the ACM Conference on Electronic Commerce*, EC'12.

Gummadi, R.; Key, P.; and Proutiere, A. 2011. Optimal bidding strategies and equilibria in repeated auctions with budget constraints. In *Proceedings of the Allerton Annual Conference on Communications, Control and Computing*.

Iyer, K.; Johari, R.; and Sundararajan, M. 2011. Mean field equilibria of dynamic auctions with learning. In *Proceedings of the ACM Conference on Electronic Commerce*, EC'11.

Liblit, B.; Aiken, A.; Zheng, A. X.; and Jordan, M. I. 2003. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03.

Mas-Colell, A.; Whinston, M. D.; and Green, J. R. 1995. *Microeconomic Theory*. Oxford University Press, Inc.

NIST. 2002. *The economic impacts of inadequate infrastructure for software testing. Planning report 02-3*. http://www.nist.gov/director/planning/upload/report02-3.pdf.

Ozment, A. 2004. Bug auctions: Vulnerability markets reconsidered. In *Third Workshop on the Economics of Information Security*.

Schechter, S. E. 2002. How to buy better testing: using competition to get the most security and robustness for your dollar. In *In Infrastructure Security Conference*.

Topcoder inc. website. *https://www.topcoder.com*.

Weimer, W.; Forrest, S.; Goues, C. L.; and Nguyen, T. 2010. Automatic program repair with evolutionary computation. *Commun. ACM* 53(5).

Wright, C. S., and Zia, T. A. 2011. A quantitative analysis into the economics of correcting software bugs. In *Proceedings of the 4th International Conference on Computational Intelligence in Security for Information Systems*, CISIS'11.