Design of a Question and Answer Approach to Crowd Debugging

Christian M. Adriano
University of California Irvine
adrianoc@uci.edu

Namrata Puri
University of California Irvine
nspuri@uci.edu

ABSTRACT

Software debugging comprises most of the software maintenance time and is notorious for requiring high-level skills and application specific knowledge. Crowdsourcing software debugging could lower those barriers by having each programmer perform small, self-contained and parallelizable tasks, hence accommodating different levels of availability and expertise. Therefore, such new approach might enable society to tackle massive software development efforts, as for instance, setting a task force of hundreds of programmers to debug and adapt the existing software to be used in an emergency response to a natural catastrophe. This type of effort is unimaginable nowadays due to the high latency in mobilizing the right programmers and organizing their work. Crowdsourcing assists in overcoming these challenges due to the availability of a large base of contributors working towards a common goal. Debugging process is not a sequential task and this leads to the primary issue of dividing the debugging task into microtasks and asking the appropriate questions based on the microtasks for analysis of the software by the crowd. Our paper attempts to provide the solution of dividing the main task into several microtasks by leveraging the structure of the task followed by associating template questions with each of the microtasks. This can assist in reducing the overhead of the individual developer during the debugging process and make crowd debugging a reality.

Categories and Subject Descriptors

D.2 SOFTWARE ENGINEERING

General Terms

Design, Experimentation

Keywords

Debugging, Crowdsourcing, Fault localization, Questions and Answers, Templates.

1. INTRODUCTION

Crowdsourcing enables non-professionals to participate in labor intensive activities and produce useful results. For instance, the game Foldit [1] enabled a crowd of people to collectively contribute to find the most stable structure of folded proteins. This brings forth the concept of applying the knowledge of the crowd to contribute to the software development effort. Besides being labor intensive, software development has a large base of contributors to possibly draw on (an estimated 13 million people in US self-report as "doing programs" [2]). Crowdsourcing software development aims at enabling thousands of volunteer programmers to make micro contributions. These micro contributions are further aggregated to accomplish a large software product. For instance, imagine that a resource allocation system (e.g., for food, water, personnel) requires the adaptation of existing software as part of the emergency response for a natural catastrophe. How could we quickly benefit from the thousands of volunteers eager to contribute to adapt the software? How to fast uncover faults in failing functionalities that are subject to unanticipated uses by the emergency response team? Therefore, adaptation boils down to a debugging process that encompasses identifying each failure, relating it to faults in the code (fault localization) and fixing it, but in record times.

Crowdsourcing software debugging is challenging because crowdsourcing usually requires a discrete sequence of steps, whereas software debugging is very hard to break down into discrete steps for many reasons. First, debugging consists of iterations of trial and error, so people don't follow predictable sequences of steps. Second, debugging the same fault with multiple people is even more difficult, because debugging as an individual activity relies on tacit knowledge that is acquired as the programmer searches for the fault, understands it and experiments with possible fixes. Therefore, the design solution for crowd debugging either contemplates a way to transfer the tacit knowledge among programmers or the solution completely precludes the need of it.

The research approach comprises a mechanism of question and answers. A lot of developers rely on questions during debugging and maintenance tasks. The solution we propose is that if the questions presented to the developers are answered within a suspicion range then is possible to identify the fault within a range of certainty. Thus, the approach primarily constructs on encapsulating questions in microtasks, the results of which are combined to either suggest a new microtask to be worked on or to reach the final stage of fault identification. Developers need to provide a positive or negative answer to the questions generated by the microtasks. We hypothesize that a continuous chain of answer ultimately produces a root-case that links the failure to the fault. The challenge of the approach is effectively dividing the debugging task into microtasks and designing questions that can be answered by considering the context of only one microtask. Our insight consists of developing questions for which positive answers imply code behaving correctly and negative answers for code behaving suspiciously. The former positive answers indicate

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

that the part of the code can be safely ignored, whereas the negative ones highlight the code for further questioning. Additionally, the answers of multiple developers can be combined and compared to check the validity of the results.

The paper is structured by explaining the research approach in section 2, which is followed by proposing the solution for the problem in section 3. Section 4 explains the methodology and evaluation of the study and followed by section 5 with the discussion of the evaluation. Section 6 highlights the limitations of the study which proceeds to the related work in section 7 and is concluded by future work in section 8 and conclusion in section 9.

2. APPROACH

2.1 Questions can be captured as templates

We are interested in questions that can be answered objectively in the context of one microtask. The solution is to ask questions about possible known problems in the code structure. Since the structures are finite and the list of problems can be made explicit, we are able to create template questions.

We currently have three types of template questions. First, questions about entry points for program execution, which comprise expected values for GUI (graphic user interface) output, code responsible to draw an output, or execution stack trace pointing to methods involved in an exception. Second are questions about function interface, which comprise return values, parameters passed, and the control flow of the program (i.e., which methods were called and when). Third are questions about function body, which consist of how values are computed, which datastructure fields are read, how parameters are mutated, and how conditionals and loops are executed.

2.2 Answers can be combined to point to faults

Answers to individual questions can provide individual points of evidence. Positive answer shows that one smallest bit of code supposedly behaved correctly, while negative answers show that a bit of code is suspicious. By collecting a set of positive and negative answers we can provide chains of evidence in twofold. First, sequences of positive answers indicate code that can safely be ignored, so we can reduce the search space and focus the crowd on the more mistrustful area of the code. Second, tracing the root of a sequence of negative answers presumably leads to the root cause of the problem.

We also aim at combining multiple answers to similar questions in order to confirm suspicion raised by different works. Thereby, we can reduce the impact of false assumptions in the debugging process. Such is essential not only to improve quality of the process, but to promote diversity of outcomes. For the same failure there might be multiple viable fixes, but with different design compromises, thereby making such diversity important in evaluating among competing sets of possible fixes [7].

3. SOLUTION

3.1 Process

The process is similar to a partition-map-reduce, in which the source code has to be divide in snippets (partition) about which questions can be asked (map) and later answers be collected to indicate where the fault is located(reduce). In order to enable that, we divide crowd debugging in four steps.

- Select and make source code available
- Generate microtasks and questions
- Analyze microtasks
- Aggregate and visualize results

Next we detail each of the steps.

Select and make source code available
An original developer facing a failure in the code decides to crowd source the localization of the fault. That is done by selecting which source files should be part of the debugging process. Some files might be deemed not pertinent for many reasons such as developer knowledge, configurations files, third party code, etc.

Generate microtasks and questions
This step entails the partitioning of the work so the crowd workers can perform small and isolated tasks in parallel. The first concern is the size of the task, which maps to the size of the source code the worker would be requested to read in order to answer the associated questions. Our design choice was for one method per microtask, which is represented in our solution by the concept of Code Snippet.

Each code snippet has different sets of internal structures (e.g., loops, parameters, conditional clauses, method calls, and return statements) and for each of those, different questions will be asked. In our process, we instantiate questions from a template list which comprises of replacing tags in the template question with the names of methods and line position of the internal structure. This is important to provide a precise context for the worker to understand and answer the questions.

A microtask contains a code snippet and a question. Multiple copies of the exact same microtask will be produced and the answers by each of the workers for the same question will be composed and presented to the original developer.

Analyze microtasks
Workers analyze microtasks by answering a question after reading the bug report, the source code, and the question prompt. Answers can be selected from one of the five following options: “Yes”, “Probably yes”, “I can’t say”, “Probably no”, “No”. The first option (Yes) points to a high level of suspicious that the structure pointed by the question has a fault. The other extreme of the spectrum (No) points to code that is considered safe and can be ignored.

Aggregate and visualize results
In this approach, two levels of aggregation take place. First, different answers for the same questions are combined together to showcase the number of answers obtained as well as the variance in them. The original developer can compare different answers and try to infer some level of confidence, or even decide to wait for more answers to be obtained. Second, all questions for the same snippet are combined together to determine which questions were effectively asked and answered. For code snippets with few answers for a particular question, the original developer might decide to wait for more rigorous analysis by the crowd. It can also be useful to comprehend the number of questions asked for a code snippet in comparison to other code snippets.

3.2 Architecture

The architecture reifies the process through a set of components, which are described as follows:

Source Code Uploader (Figure 1): This component enables the original developer to upload the source code related to the failure.

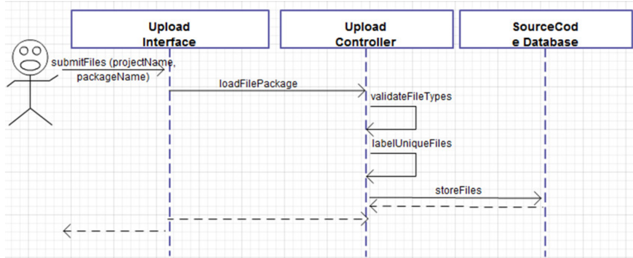

Figure 1. Source code uploader

The steps for generation of microtasks are displayed in Figure 2.

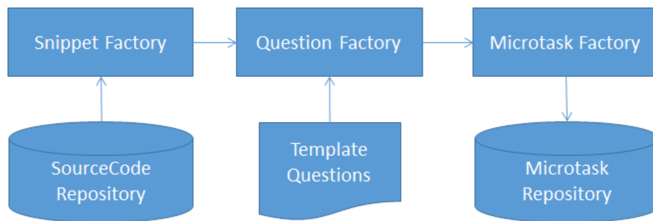

Figure 2. Generating microtasks

Snippet Factory: This component partitions the source code in code snippets according to the methods. After partitioning each file in the repository is retrieved and marked as processed.

Question Factory: For each code snippet, this component instantiates the appropriate template questions, which are then indexed to the code snippet in order to track future answers.

Microtask Factory: This component generates microtasks by receiving an instantiated template question and the associated code snippet.

Microtask Interface: The web page for a crowd worker to perform a microtask is provided by this component (Figure 3). Workers provides login credentials and are redirected to a web page consisting of a randomly selected microtask. If desired, the worker can request new microtasks. The answer for the microtask provided by the worker is collected and associated with the worker, the respective code snippet, and the template question. This is important to prevent the same worker from answering the same question multiple times.

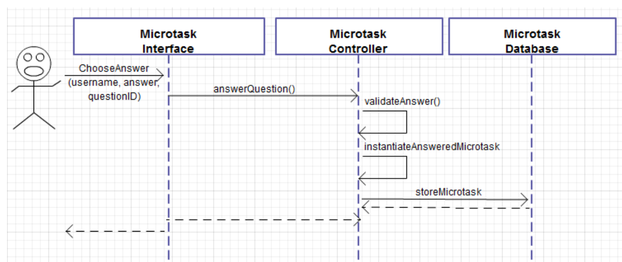

Figure 3. Microtask interface to collect answers

Results Panel: The functionality of this component is to keep track of the results to be presented to the original developer aggregated according to the answers for the same question and answers for the same code snippet. (Figure 4).

code Snippet (method name)	# of template questions instantiated	% of questions answered	suspiciousness (1=low, 5= high)	detail
inscribeFigure()	7	57%	3.5	
computeSideBySide()	4	75%	3.5	
computeConcentric()	3	67%	3.7	

Question	Answers
Is it possible that the conditional clause at line 10 has any typical faults (e.g., wrong boolean operator, wrong comparison, misplaced parentheses)?	2;3;3;2
Is it possible that the conditional clause at line 10 causes the wrong code to be executed, or does not protect the code from a call to a null pointer, or from a	1;1;3;2
Is there perhaps something wrong with the parameters received by function computeConcentric (e.g., wrong order, missing parameter, wrong type of parameter, parameters that are not checked)?	no answer

Figure 4. Results panel with two levels of microtask aggregation

4. METHODOLOGY AND EVALUATION

The initial concern was related to obtaining the first set of questions which could be used to identify bugs by only considering the source code. To identify faults in the code, our approach was to understand existing bugs by looking at bug reports and respective bug fixes. In order to guide the mining of bugs, we categorized bugs in four types: static, dynamic, behavioral, and crosscutting. The static type requires only the source code to be identified. The dynamic type requires the values of the variables or the execution stack trace. The behavioral type requires the expected values for each method (i.e., unit tests). The crosscutting type requires the expected values for global states (i.e., data that is globally shared among methods). According to the four types of bugs, we adopted four criteria for mining them. First, bugs should all be from the same system and in the same programming language to enable the comparison of their fixes. Second, bug reports should have "fixed" in the resolution state. Third, bugs should include explanations of the fixes in the bug reports. Four, fixes should be limited to a single files. We selected the Mylyn application because it is implemented in Java and is in a very active stage with a large number of recent bug reports along with a detailed discussion of fixes for them. We mined bug reports from Eclipse Bugzilla and respective bug fixes from GIT. It is worth mentioning for validity concerns that we had to manually match the bug identification number code in Bugzilla with commit messages in GIT. Only bugs that could be matched were considered. We mined 253 bug reports and found 26 bugs that fit in the static type. After analyzing these 26 instances, we subsume them in three subcategories: missing data, wrong data, or exception. We combined these subcategories with the notion of asking questions about the code structures and this resulted in an initial set of 10 questions.

The next step was the design of the prototype for the crowd sourcing system. This was carried out by building the front end of the system in HTML. Separate access levels were provided for the developers uploading their code snippet for verification by the crowd and the developers downloading and analyzing the code snippet to assist in the debugging activity. The back end of the system was designed in Microsoft Excel which contained the initial set of template questions, the final set of questions adapted

for the particular code snippet as well as the results of the debugging task as provided by the developers.

This was followed by carrying out an evaluation of the initial set of questions. The code snippet to be analyzed by the crowd was manually generated to incorporate faults. The manual generation of the code snippet assisted in evaluating the number of faults located by the crowd as we were aware of the exact position of each fault. This code snippet was matched with the template questions and 10 questions adapted to the code snippet were generated. Along with the set of 10 questions, additional 3 questions were generated to compare the questions belonging to the same category. This helped in determining the questions in the same category which were more challenging to answer thereby assisting in the future design of the questions. A bug report was created for the code snippet which consisted of the problem statement, the expected output and the actual output as well as the code which was to be analyzed. The questions along with the bug report were sent to crowd of developers and responses were collected from the 17 developers. Each response was in the form of a Likert scale of 5 ranging from “I am sure there is a fault in this line” to “I am sure there is no fault in this line” with the intermediate three response leaving room for slight uncertainty in bug detection. This was followed by collecting feedback from the developers on the microtasks generated and the questions asked.

According to the results collected from the developers as well as their post-task completion feedback, the initial set of questions were modified to incorporate their feedback and the new set of 8 questions were defined. Among the 8 template questions, 6 questions were found to be relevant to the code snippet and this new set of 6 questions were circulated among 5 developers using the same code snippet and bug report. The results of the debugging task were analyzed. This was also followed by collecting feedback on the revised set of questions.

All data from experiments are available for download [24].

5. DISCUSSION

The analysis of the initial set of 10 questions revealed the fact that all the developers were able to successfully locate the faults in the code snippet. The developers were in agreement of the faulty lines of code and thereby guaranteed the soundness of our questions. This was followed by the analysis of the 3 comparison question as well as the feedback from the developers to guide the redesign of the new set of questions. It revealed that some of the questions were very trivial to answer, with quotes for the developers such as “The first (was harder to answer). To answer the second we just need to count.” and “The second question was harder to answer (as compared to the first and third) because I needed to link the requirements with the code.” One of the questions in our study received the feedback of being vague with the developer commenting “Ambiguous question. The function "can" return a value, but whether it is relevant to the requirement or not is questionable.” According to the feedback, the questions which were ambiguous were clarified to make them more specific to the task at hand as well as the questions which were trivial to answer were either replaced by more substantial questions or were omitted. Accordingly, the new set of 8 template questions was defined. 6 of these questions were found to be relevant to the code snippet according to the structure and content of the code snippet and the crowd debugging was carried out with this new set of questions and associated code snippet and bug report.

The results were analyzed and it was revealed that the developers were able to aptly uncover the fault location in the code snippet that was assigned to them. In addition to differentiating the faulty lines from the lines which were not faulty, the developers were able to identify the lines which may or may not have faults depending on the input provided by the user. This was clearly a progress from our first evaluation wherein the developers had issues with identifying the faults in the lines which were dependent on the user input. The feedback from the developers on the quality and the accuracy of the questions showcased that the revised set of questions were a lot less vague with one of the developers commenting that “The questions in particular were easy and clear to understand”. The developers considered the questions to be of the right amount of difficulty with one of the developers commenting that “The questions were challenging that the original uploader required assistance from the crowd, yet not too challenging such that they could not be understood and thereby not answered by the crowd.” The developers also appreciated the fact that the questions were limited in number and yet were able to discover the faults in the program. However, one of the issues that the developers had with the questions was the presence of multiple ORs which required them to disintegrate the question into several individual questions to provide an efficient answer. Another issue that the developer had in the debugging task was the requirements for the task and the bug report was challenging to comprehend and the names of the variables which were present in the uploaded code were not intuitive. Another developer highlighted the issue that the functionality of each method was subject to personal assumption by the developers carrying out the debugging task and this could be misleading in identifying the faults.

6. LIMITATIONS

Three threats to validity [23] were analyzed; external validity, construct validity and, internal validity.

The primary limitation of this paper is related to the evaluation of the template questions. The evaluation was carried out with a small number of developers. This brings about the issue of generalizing the findings of the study to the larger crowd as the debugging approach of a large number of people may differ significantly thereby resulting in the issue of conclusion validity, also known as external validity. Additionally, the code which was tested for the evaluation was a Java code snippet. Having such a language specific code snippet could have had an influence on the template questions developed in the study. Although through the evaluation, the answers to the debugging questions provided were to be the result of primarily the design of the template questions, there could be a possibility that the perception of the developers towards the questions was a factor in answering the question. This would have a negative effect on the redesigned new set of questions resulting in an issue known as construct validity. Additionally, the new set of questions in the second evaluation were a result of redesigning the questions from the first evaluation based on the responses by the developers and this could be a subject to the issue of internal validity if the mapping rationale, which was selecting the trivial questions as well as the ambiguous questions for refinement was subject to interpretation bias in the analysis of the evaluation.

7. RELATED WORK

Questions have a key role in software programming. Questions and answers have been a recent focus of research in the field of

program comprehension, system maintenance, and software debugging. Research has shown that programmers ask why and why-not questions during debugging [3]. Programmers rely on categories of questions while changing code as part of software evolution tasks [4]. Also in this context of software adaptation, programmers use questions to navigate through their code [5]. A successful instance of questions for debugging is the website Stackoverflow (www.stackoverflow.com), where programmers post their faulty code and receive advice from the community by means of concrete code examples [6]. As we see, questions have been part of multiple and varied successful approaches to debug software.

Our approach builds on top of it by encapsulating questions in microtasks. The results of microtasks are aggregated to either suggest new microtasks or to identify a fault. Each microtask contains an automatically generated question, for which a worker has to provide an answer. The chain of answers represents the root-cause that links the failure to the fault.

One concern is the way our approach handles the technical problems that have prevented progress and success in this area in the past. Recent research has been exploring crowdsourcing to help in finding and fixing bugs. Regarding crowdsourcing for recommending bug fixes, in the systems "Help Me Out" [13] and "BlueFix" [14], crowd workers fix compilation errors based on suggestions from previous workers. Similarly, the prototypes named "Crowd::Debug" [15] and "Crowd Oracles" [16] harvest fixes to unit tests, which can later be queried to help an individual developer facing a test failure. This differs from our approach in that these systems use the crowd to build a database of fixes, which are later queried to compare against one specific situation. Counter wise, in our approach the crowd is actively working on a specific situation. While the previous examples involve specialists, there is also past work on systems that enable non-specialists to help in finding errors. For instance, using a gamification paradigm [22] to verify software models for correctness [20] or for security vulnerabilities (Pipe Jam [21]). An approach that is similar to ours, but focused on a different type of problem, is the crowdsourcing of GUI usability as for example in the commercial systems "Five Seconds Test" (www.fivesecondtest.com) and "Voyant" [17]. Their approach consists of asking template questions to a crowd of designers in order to collect opinions and suggestions about the usability of a specific GUI. The Whyline [3] solution can be seen as a modular debugging approach for a single user. Whyline enables a single user to debug by relating program output with program statements, by posing questions, and by navigating program dependencies. Enacting debugging with multiple concurrent users and enforcing a specific granularity for each debugging step are issues out of the scope of the Whyline tool.

Automatic debugging techniques were criticized for assuming that people can explain a bug by looking only at isolated pieces of code [18]. We mitigate that by enabling the user to set executable expectations for code dependencies (stubbing functions and modifying inputs dynamically). Crowd debugging could thereby be instrumental to complement large scale challenges for which automatic program repair was shown to be promising [19].

Ultimately our approach is innovative because it studies debugging in a new context of concurrency, fragmentation, and lack of ownership. In other words, instead of tens of people resolving tens of bugs, our approach is to have tens of people answering a multiplicity of different questions in order to resolve a single bug at a time. Previous work tended to surrender to the facts that debugging relies heavily on tacit knowledge and unpredictable steps, so deeming it not amenable to be systematized. We believe there is a design that can strike a collaborative dynamic between work performed by a crowd and decisions made by tools and expert teams.

8. FUTURE WORK

The primary future work of this paper is to improve the generalizability our findings, for which we plan a study with a larger crowd of developers, recruiting them through a crowd sourcing platform such as Amazon's Mechanical Turk. This would provide results which would be more accurate as they would match with the real world setting. The evaluation would also be carried out using real code snippets in multiple programming languages and the questions would be redefined into a generalizable language independent set of questions.

Currently, according to our approach, questions for the debugging tasks are asked on the entire code base. In the future, there would be an automated mechanism set up to determine which section of the code base has more faults and to which sections should the efforts of the developers be concentrated. This automatic way to intelligently analyze the answers from a crowd can be too hard for many reasons. First, we do not know the level of precision and accuracy of the answers. Second, the selection of the next set of questions is not obvious, because it is sense making activity, which combines objective inputs (e.g., answers gathered) with subjective ones (e.g., experience, knowledge, intuition). It resembles the concept of information foraging, which was also recently explored for debugging [8][9]. Third, we are faced with the halting problem, which means that we do not have an algorithm to tell us when to stop asking questions.

Therefore, solution that contemplates an expert team interacting with the crowd is a learning step to discover what intelligence is amenable to automation. In order to be more concrete, the following is a list of possible activities that the team could perform with the crowd:

- Select which parts of the code deserve more or less investigation based levels of suspiciousness, density of answers, or some other measure of code quality (e.g., code churn, history of defects).
- Add new questions on-the-fly or rephrase any for clarity or for double checking. The new questions could be contributed by the expert team as well as the crowd. Some of those new questions could become new templates, whereas others not, because they could be distinct in nature (e.g., domain or context specific). Such mechanisms would enable the generation of questions by and with the crowd, which can be understood as an unfolding and emerging phenomenon. This is similar to the process of the individual programmer when discovering new knowledge by raising and eliminating hypotheses [10][11].
- Adapt the divide-and-conquer strategy by deciding where to concentrate next crowd efforts. For instance, by visualizing a map of the code and where the crowd is currently working on (or has been).
- Finally, decide when to stop asking questions on some parts of the system, because the team might judge that the knowledge gathered is sufficient, that is, it is enough

to perform a bug triage, to scope where the fault is, or in the best scenario, to suggest a fix.

- In terms of experimental setting, the team could comprise regular developers who receive a basic training on the architecture and the domain of the source code.

Improvement will also be made in the future to augment the developer experience of crowdsourcing the debugging task by allowing them to select multiple files to be crowd sourced at the same time thereby reducing the overhead of selecting each file individually. Additionally, in the debugging session, if the files are predicted by the developer to be related to a particular bug, then the files would be labeled accordingly and linked to the bug. In our current approach, the microtasks are formed by considering all of the method in the code snippet uploaded by the developer. In the future, automatic mechanism will be employed to unselect particular methods by their specific patterns such as unselecting the getter method, setter method, abstract methods and constructors. One concern is to be able to update files as the debugging process progresses, because changes or fixes will be more relevant by taking into consideration the current bug. Therefore, it is useful to introduce the idea of debugging session, which consists of a set of file versions and a failure description. Every time a file is update, a new debugging session will be created, because answers obtained for a previous file version would have become invalid.

9. CONCLUSION

The quality of the software depends on the quality of each individual phase in the software development and one of the longest and most critical phases in software development is the debugging phase in addition to being most challenging phase due to its inherent complicated nature. Through this paper we have presented a novel approach to software debugging by using the knowledge of the crowd to make the debugging process more efficient and manageable. The major challenge in this approach is of defining the microtasks from the code and the associated questions related to the microtasks for analysis by the crowd. These two issues are addressed through this paper by leveraging the structure of the code for generating microtasks and developing a set of template questions which can be adapted for each of the microtasks. The initial set of template questions were developed and were refined according to the results obtained from the preliminary evaluation. The testing carried out on these new set of templates questions showcased that the crowd was able to successfully locate and detect the fault in code, thereby providing evidence of the success of this approach. Our approach will be further investigated, so as to improve the fault localization capability of the crowd.

10. ACKNOWLEDGMENTS

Christian M. Adriano is supported by the CNPq and the Department of Informatics of the Donald Bren School of Information and Computer Science at the University of California Irvine.

11. REFERENCES

[1] Khatib, F., Cooper, S., Tyka, M. D., Xu, K., Makedon, I., Popović, Z., & Foldit Players (2011). Algorithm discovery by protein folding game players. In *proc. of the National Academy of Sciences*, *108*(47), pp. 18949-18953.

[2] Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. *In proc. of Visual Languages and Human-Centric Computing*, pp. 207-214.

[3] Ko, A., J., & Myers, B., A. (2009). Finding causes of program output with the Java Whyline. *In proc. of SIGCHI Conference on Human Factors in Computing Systems*, pp. 1569-1578.

[4] Sillito, J., Murphy, G., C., & De Volder, K. (2006). Questions programmers ask during software evolution tasks. *In proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 23-34.

[5] LaToza, T., D., & Myers, B., A. (2010). Developers ask reachability questions. *In proc. of the ACM/IEEE International Conference on Software Engineering,* pp. 185-194.

[6] Nasehi, S., M., Sillito, J., Maurer, F., & Burns, C. (2012). What makes a good code example? A study of programming Q&A in StackOverflow. In *proc. of the IEEE International Conference on* Software Maintenance, pp. 25-34

[7] Murphy-Hill, E., Zimmermann, T., Bird, C., & Nagappan, N. (2013). The design of bug fixes. In *proc. of the ACM/IEEE International Conference on Software Engineering International Conference on Software Engineering*, pp. 332-341.

[8] Lawrance, J., Bogart, C., Burnett M., Bellamy, R., Rector, K., & Fleming, S. (2013). How programmers debug, revisited: An information foraging theory perspective. In *IEEE Transactions on Software Engineering*, 39(2), pp. 197-215.

[9] Kuttal, S., Sarma, A., & Rothermel, G. (2013). Predator behavior in the wild web world of bugs: information foraging theory perspective. In *proc. Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 59-66

[10] Marie Vans, A., Mayrhauser, A., v., & Somlo, G. (1999). Program understanding behavior during corrective maintenance of large-scale software. In International Journal of Human-Computer Studies, 51(1), pp. 31-70.

[11] Ko, A., Myers, B., Coblenz, M., & Aung, H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. In *IEEE Transactions on Software Engineering*, 32(12), pp. 971-987.

[12] Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., & Felleisen, M. (1996). Catching bugs in the web of program invariants. In *ACM SIGPLAN Notices*, 31(5), pp. 23-32

[13] Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do: suggesting solutions to error messages? In *proc. of the ACM/IEEE SIGCHI Conference on Human Factors in Computing Systems* pp. 1019-1028.

[14] Watson, C., Li, F. W., & Godwin, J. L. (2012). BlueFix: using crowd-sourced feedback to support programming students in error diagnosis and repair. In proc. of the *Advances in Web-Based Learning-ICWL 2012*, pp. 228-239.

[15] Mujumdar, D., Kallenbach, M., Liu, B., & Hartmann, B. (2011). Crowdsourcing suggestions to programming problems for dynamic web development languages. In *proc*

of ACM/IEEE Extended Abstracts on Human Factors in Computing Systems, pp. 1525-1530

[16] Pastore, F., Mariani, L., & Fraser, G. (2013). CrowdOracles: Can the Crowd Solve the Oracle Problem. In *proc. of the IEEE International Conference on Software Testing, Verification and Validation,* pp. 342-351

[17] Xu, A., Huang, S. W., & Bailey, B. P. (2014). Voyant: Generating Structured Feedback on Visual Designs Using a Crowd of Non-Experts. In *Urbana*, 51, 61801.

[18] Parnin, C., & Orso, A. (2011). Are automated debugging techniques actually helping programmers*? In proc. of International Symposium on Software Testing and Analysis*, pp. 199-209.

[19] Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. *In proc. of ACM/IEEE International Conference on Software Engineering*, pp. 3-13.

[20] Li, W., Seshia, S.A., and Jha, S. CrowdMine: Towards Crowdsourced Human-Assisted Verification. *Proceedings of the 49th Annual Design Automation Conference*, ACM (2012), 1254–1255.

[21] Dietl, W., Dietzel, S., Ernst, M.D., et al. Verification Games: Making Verification Fun. *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, ACM (2012), 42–49.

[22] Deterding, S., Dixon, D., Khaled, R., and Nacke, L.From game design elements to gamefulness: defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments* (MindTrek '11). ACM (2011), pp.9-15.

[23] Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285-311). Springer London..

[24] Adriano C., Puri, N, (2014). Pilots Run during Spring 2014. DOI: 10.5281/zenodo.10055