GenProg Meets Cluster Computing

Junnosuke Matsumoto, Yoshiki Higo, Hiroyuki Matsuo,
Ryo Arima, Shinsukue Matsumoto and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{j-matumt, higo, h-matuo, r-arima, shinsuke, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—GenProg is an automated program repair tool that leverages genetic algorithm. In the repairing process of GenProg, a larger number of mutated programs are generated, built, and tested. If none of mutated programs passes all the test cases, GenProg redos the loop of generation, build and test. The build and test occupy over 90% of GenProg's execution. In this paper, we introduce our cluster-based GenProg, which builds and tests in parallel with many computers. We have implemented the cluster-based GenProg with Kubernetes environment and applied it to real bugs in Defects4J dataset. As a result, we confirmed that the time required for program repair is reduced according to the number of computers in the cluster.**

***Index Terms*—automated program repair, GenProg, cluster computing**

I. Introduction

Software developers spend long time for debugging [1], [2]. A variety of research has been conducted to support debugging. GenProg [3], which is an automated program repair (in short, APR) tool, made a breakthough. It succeeded in removing real bugs from OSS without any developer intervention [4]. GenProg leverage genetic algorithm to generate a repaired program from a given faulty program. GenProg generates some mutated programs and executes all given test cases for each of them. If a mutated program passes all test cases, the mutated program is returned as a repaired program. If not, GenProg selects some of the mutated programs and then, generates new mutated programs and executes test cases. This loop (selection, generation, and execution) is repeated until finding a solution or reaching time limit or maximum number of generations.

A serious issue in GenProg is its long execution time. GenProg naturally takes long time to find a repaired program if the loop is repeated many times. We run kGenProg[1] [5], which is a Java edition of GenProg, for bugs of Apache Commons Math, so that we found that compiling mutated programs and executing test cases occupy over 90% of execution time. Long execution time is a serious issue for practical use for both practioners who want to use GenProg on their projects and researchers who want to improve GenProg.

In this paper, we propose to parallel compilation and test execution to shorten GenProg's execution time. However, a simple parallelization in a single computer has a limited capability for shortening execution time. Thus, we develop a framework[2] to execute compilation and test execution with a cluster computing. In our proposed framework, unlimited number of computers can be joined to the cluster as a node of compiling mutated programs and executing test cases for them. We implemented our cluster computing in our IaaS environment and we confirmed that our cluster computing dramatically shortened the time required to find repaired programs for real bugs in open source software.

[1]https://github.com/kusumotolab/kGenProg

[2]https://github.com/kusumotolab/clustered-kGenProg

II. Terminology

Herein, we introduce several terminologies used in this paper.

A. Fault Localization

Fault localization is a technique to infer which lines of a given program include the bug. APR techniques require fault localization because they need to decide which lines to change. In the context of APR, spectrum-based fault localization technqiues are used. Several studies reported that Ochiai [6] outperforms other spectrum-based techniques [7].

B. GenProg

GenProg is an APR technqiue that leverages genetic algorithm [3]. Firstly, GenProg identifies suspicious lines by using a fault localization technique. Secondly, GenProg generates multiple mutated programs by changing the suspicious lines. The program mutations include three operations: insertion, deletion, and replacement. Thirdly, GenProg builds and executes given test cases for all the mutated programs. If a mutated program succeeded in passing all the test cases, GenProg outputs it as a repaired program. If not, GenProg picks up some better programs and generate new mutated programs of the next generation. This processing is repeated until finding a repared program or reaching time limit or maximum number of generations.

In the operations of insertion and replacement, GenProg uses a program statement in a given program. Selecting an operation and selecting a program statement in case of insertion and repalcement are randomly decided.

C. kGenProg

kGenProg [5] is a Java implementation of GenProg. kGenProg has a high portability, only a single JAR file of kGenProg is required to repair target programs. kGenProg equips interfaces for many operations such as fault localization and program mutation. Thus, kGenProg users easily add new features for such operations. kGenProg builds mutated programs

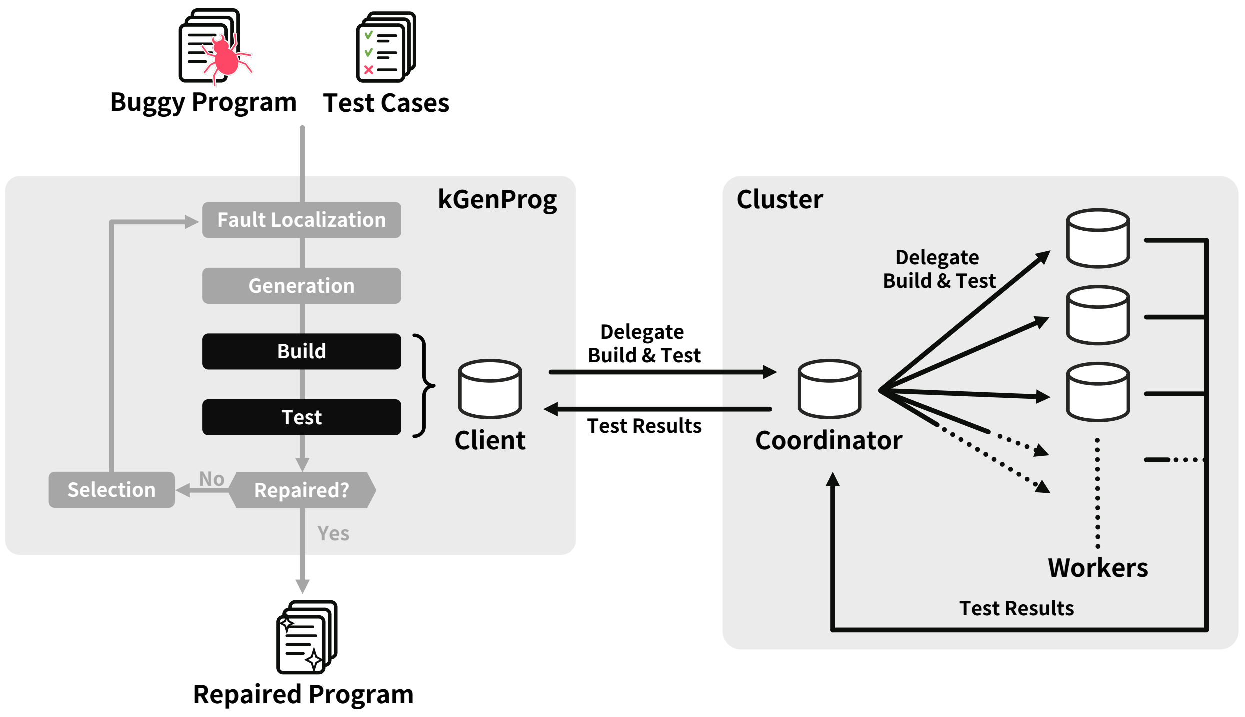


Fig. 1. Overview

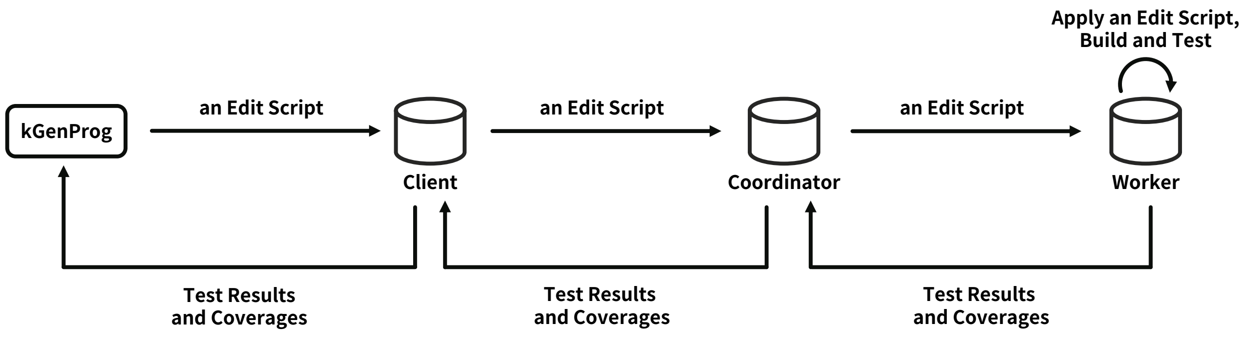

Apply an Edit Script, Build and Test
kGenProg
an Edit Script
Client
an Edit Script
Coordinator
an Edit Script
Worker
Test Results and Coverages
Test Results and Coverages
Test Results and Coverages

Fig. 2. The flow of a mutated program

and execute test cases on memory to avoid file IO as much as possible. kGenProg's compilation is a differential one, which is very effective for large programs.

D. Kubernetes

Kubernetes[3] is a container orchestration system for automating application deployment, scaling, and management. It works with a range of container tools, including Docker. To utilize Kubernetes, we prepare a master node firstly, and then we register multiple computers to the master node. Kubernetes distributes a Docker image container to each of the registered computers and we can execute a program on them with only a single command. Kubernetes monitors each of the registered computers during program execution. If some errors happen in a container of a registerd computer, Kubernetes dispose it and launch a new container automatically.

[3]http://kubernetes.io

III. Proposed technique

To improve the performance of GenProg, we focus on the build phase and the test phase that mainly impact the performance. While multiple mutated programs can be built and tested in parallel, if these phases are simply executed in multiple threads in only a machine, the performance is limited by the specification of the machine.

Thus, we implemented a tool that improves the performance of GenProg by distributing its build phase and test phase. Our tool runs on many computers that execute build and test. Our tool is implemented as an extension of kGenProg [5] because it is easy to extract the build phase and test phase from kGenProg.

Figure 1 shows our tool's overview. It consists of three nodes.

Client: the node is the implementation of the interface in kGenProg that it is responsible for executing build and test.

Worker: the node actually executes build and test.

Coordinator: the node has responsibility for relaying between the client and workers and balancing loads.

These nodes communicate with TCP/IP network.

Figure 2 shows the flow that kGenProg generates an edit script (a set of editing actions), the client sends the edit script to a worker via the coordinator and it receives the test results and the coverages. The coverages are used in kGenProg to determine which program statements are the targets of next editing actions. kGenProg generates edit scripts from the input program and sends each of them to a different worker. Then, each worker mutates the input program based on the edit script that it received. Herein, the mutated program means the program that the worker executes build and test. After executing build and test, each worker sends the test results and the coverages.

A. Communication between nodes

Figure 3 shows the communication flow.

(1) Registering a worker: when a worker is launched, it notifies the coordinator of the IP and the port of the worker, and the coordinator recognizes them.

(2), (3) Sending files to the coordinator and the workers: when kGenProg is launched, the client sends files to the registerd workers via the coordinator for build and test.

(4), (5) Notifying build completion: the worker builds the received files and notifies the coordinator of build completion. After receiving the notification from all workers, the coordinator notifies the client of build completion.

(6), (7) Sending an edit script to the worker: after kGenProg calculates an edit script, the client sends it to the coordinator. The coordinator relays the received edit script to an idle worker.

(8), (9) Returning the test results and the coverages: the worker mutates the program based on the received edit script, and then, it builds and tests the mutated program. The worker returns the test results and the coverages to the coordinator, and then, the coordinator relays the received test results and the coverages to the client.

The flow of (6)$\sim$(9) is executed in parallel. kGenProg calculates edit scripts and iterates the loop of (6)$\sim$(9) until finding a solution.

B. Client

A client is imported to kGenProg as a strategy which executes build and test. kGenProg delegates build and test to the client. The client uses the following information: (a) files required to build and test target projects, and (b) edit scripts. The client compresses the files, which are necessary for build and test, to a zip file, and the client sends the binary to the coordinator.

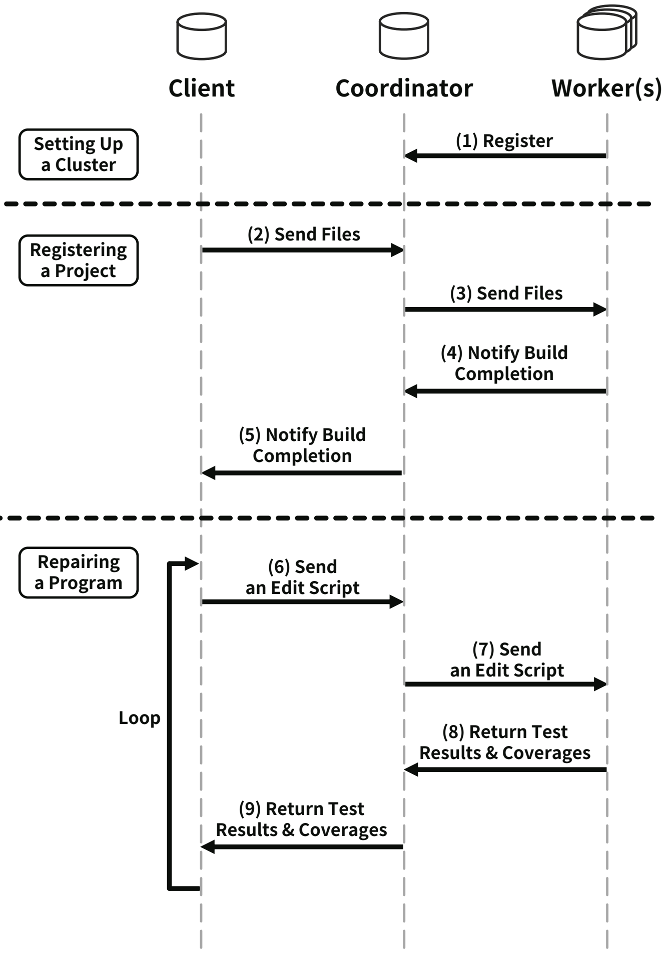


Fig. 3. The communication flow

C. Coordinator

A coordinator is a node which relays from the client to workers which execute build and test. The coordinator does the followings:

- registering workers ((1) in Figure 3),
- registering a project ((2)$\sim$(5) in Figure 3), and
- receiving a delegation for build and test ((6)$\sim$(9) in Figure 3).

1) Registering workers: The coordinator always receives a request for registering a worker. After receiving a notification which a worker has been launched, the coordinator connects to the worker.

2) Registering a project: The coordinator always receives a request for registering a project. The coordinator provides an ID to a given project, and the coordinator sends the binary and ID to all registered workers. After receiving responses from all workers, the coordinator returns the project ID to the client.

The coordinator keeps the pair of the binary and the ID, because a new worker may be registered after registering projects.

3) Receiving a delegation for build and test: The coordinator receives a delegation for build and test from the client.

The coordinator receives a request which contains a project ID and an edit script from the client and sends the request to an idle worker.

The requests from the client are kept in the coordinator. The coordinator manages which worker is delegated build and test, and sends the next request to a worker which has finished build and test. If a worker halts unexpectedly, another worker is delegated.

D. Worker

A worker is a node which actually executes build and test. The coordinator sends a project ID and an edit script to a worker. The worker mutates the project based on the edit script and executes build and test. All files in the project are built at first build, after this, only edited files are built.

After finishing the build and test, the worker sends the test results and the coverages to the coordinator. The reason why the worker sends the coverages is that the coverages are necessary for the fault localization in kGenProg.

E. Deployment

While we can launch each node one by one, the more workers is launched the more difficult it is to manage all nodes. Thus, our tool adopts k8s for the deployment. We prepared the Docker images for the coordinator and the workers. By editing the configuration file of k8s, a user can easily change the configuration of the coordinator and the workers (e.g., the number of workers, the specification of each node). k8s checks the status of each worker. If a worker unexpectedly exits (e.g., out of memory, network error), the container is restarted by k8s automatically, and the worker connects to the coordinator again soon. Our tool can always keep the number of workers which a user wants.

F. How to Use

Figure 4 (a) shows an example of the command which launches our tool manually without k8s. At first, a user launches a coordinator. The user launches a worker, and then, the user registers the worker to the coordinator. The user has to execute the command which launches a worker as many as the user wants.

By using k8s, the user can more easily launch our tool. Figure 4 (b) shows the commands. After setting up k8s, the user can set up a cluster only by executing one command. deploy.yml has information about the cluster (e.g., the number of workers, the specification of each node, the port of the coordinator). If the user wants to increase the number of workers, all the user has to do are only modifying the files and executing the command.

TABLE I
THE SPECIFICATIONS OF EACH NODE

Node	CPU	Memory
kGenProg (including Client)	2	32GB
Coordinator	2	16GB
Worker	1	6GB

```
# launch a coordinator
$ coordinator −−port 50051

# launch a worker (as many as you want)
$ worker −−host <Coordinator IP> −−port 50051

# launch kGenProg with a client
$ client  −−host <Coordinator IP> −−port 50051 \
          −−kgp−args '−−config kGP.toml'
```

(a) launching nodes manually

```
# in  k8s master node
$ kubectl apply −f kubernetes/deploy.yml

# launch kGenProg with a client
$ client  −−host <k8s master IP> −−port 50051 \
          −−kgp−args '−−config kGP.toml'
```

(b) launching nodes with k8s

Fig. 4. The command that launches our tool

IV. EVALUATION

We evaluate the performance of our tool by applying it to OSS. We expect that the performance of our tool can be improved by adding workers. Thus, the purpose of this experiment is that we confirm how the performance is improved by adding the workers.

A. Dataset

We use the Apache Commons Math (in short, Math) included in Defects4J [8] for the evaluation. Defects4J is a dataset which contains information of real bugs occurred in the development of some projects. Math has 106 bugs. The main reason why we use Math is that it is used as benchmarks in many papers of APR [9], [10].

B. Environment

In the experiment, six IaaS servers were used. Each IaaS server has 12 CPUs and 100GB memories, and all the IaaS servers are connected to the same LAN.

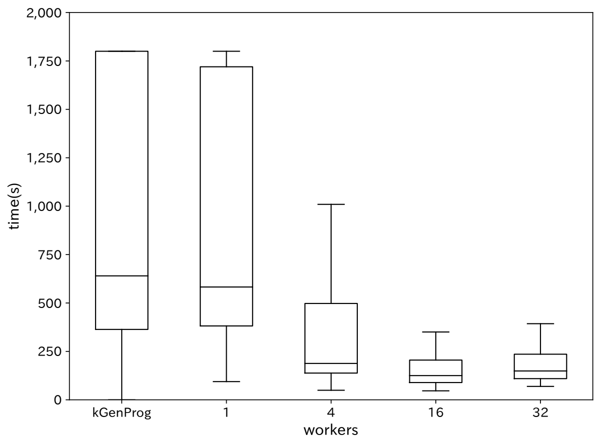
We launched the virtual machines which has specifications as much as their IaaS servers. The virtual machines were registered k8s as a slave node. A coordinator and workers were launched in the k8s.

Table I shows each node specification. The client is imported to kGenProg, thus, the table shows the specification of kGenProg and the client.

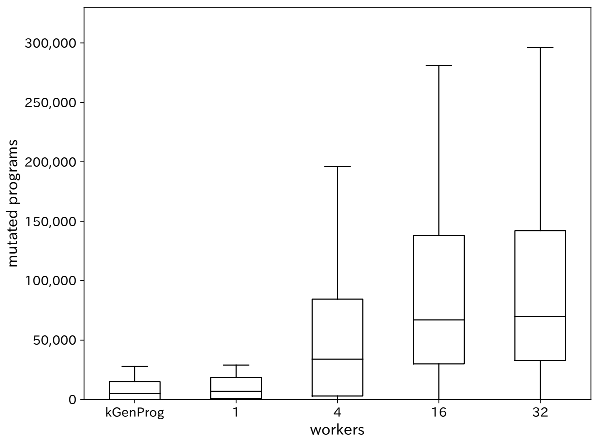
C. Configuration

We expect that the more workers are added the more the performance is improved. Thus, in the experiment, we compared the performances of stand-alone kGenProg (in short, SA) and our tool when 1, 4, 16 and 32 workers were launched.

Our tool distributes build and test in a generation. We considered that, when the number of workers in a generation is low, our tool cannot evaluate enough because the advantage of distributing build and test is not large. Thus, we set 1,000



(a) Execution time



(b) # mutated programs

Fig. 5. The results of the experiment

as the number which kGenProg generates programs in a generation.

The time limitation for each bug was set 30 minutes, the maximum generation was set unlimited and the termination condition was set time out or finding a solution.

D. Results

Table II shows the number of found solutions according to the number of workers. In the case that multiple workers were launched, the more bugs were repaired than the case of SA.

Figure 5(a) shows the execution time in the case that kGenProg found a solution with any of SA and 1 $\sim$ 32 workers. The time of 4 workers is shorter than the time of 1 worker and the time of 16 workers is shorter than the time of 4 workers. The reason why the time of 32 workers is longer than the time of 16 workers is that the client takes longer time to send the project files to more workers.

Generating more mutated programs increases the possibility of finding a solution. Thus, the number of mutated programs is essential. Figure 5 (b) shows the number of mutated programs in the case of time out. The number of mutated programs generated by 4 workers is higher than 1 worker and the number of mutated programs generated by 16 workers is higher than the 4 workers. The reason why the performances of 16 workers and 32 workers do not change significantly is that the ratio of non-distributed processing gets larger, and the effect of distributing build and test gets smaller.

V. Threats to Validity

Each node can also communicate with others via a global network. In such a situation, we expect the performance will be lower than the measured performance. However, we do not know the impact because we have not measured the performance in such a situation.

kGenProg mutates a program based on a random seed. In the experiment, we used only one seed. Thus if we used multiple difference seeds, the results (the performance and the number of solutions) will be changed.

TABLE II
The number of solutions which kGenProg found

SA	1 worker	4 workers	16 workers	32 workers
25	29	36	36	36

VI. Discussion

In this paper, we mainly presented our tool, which distributes the builds and the tests. On the other hand, our tool does not execute the other phases (e.g., the generation phase) in parallel. It is not realistic to parallelly execute such phases in kGenProg due to its architecture. For example, in the case of the generation phase, the reason is that kGenProg remembers all mutated programs to avoid generating the same mutated programs, and so, the order that kGenProg mutates programs is important.

However, this is an implementation problem of kGenProg. While we adopted kGenProg because it is easy to distribute the builds and tests that mainly impact the performance, the other implementations of GenProg may be able to execute the fault localization and the generation in parallel. We consider that the selection cannot be executed in parallel regardless of the implementation of GenProg, because GenProg selects a fixed number of programs from many mutated programs.

VII. Related Work

Claire Le Goues et al. [11] proposed a technique which executes GenProg in parallel using cloud computing resource. However, the technique simply executes GenProg in parallel using different random seeds. Thus, the tool generates the same mutated programs many times between different random seeds, and the performance per a single random seed is not improved. Since our technique executes kGenProg in parallel using a single random seed, the performance per a single random seed is improved.

VIII. Conclusion

We proposed a tool which improves the performance of GenProg. Our tool improves the performance by distributing the build and test. By applying the tool to open source software, we confirmed that multiple workers improve the performance than a single worker.

As future work, we will propose more intelligent mutation by using our tool. While GenProg mutates programs per statement, our tool can mutate programs per node of abstract syntax tree because the number of programs which our tool can build and test is larger than the one of GenProg within the same time.

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 17H01725.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software - quantify the time and cost saved using reversible debuggers," 2013.

[2] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *In Proc. International Conference on Software Engineering*, 2009, pp. 364–374.

[4] C. Le Goues, M. Dewey Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *In Proc. International Conference on Software Engineering*, 2012, pp. 3–13.

[5] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kgenprog: A high-performance, high-extensibility and high-portability apr system," in *the 25th Asia-Pacific Software Engineering Conference*, 12 2018, pp. 697–698.

[6] A. da Silva, Meyer, A. Augusto, Franco Garcia, A. Pereira, de Souza, and C. L. de Souza, Jr., "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l)," *Genetics and Molecular Biology*, vol. 27, no. 1, pp. 83–91, 2004.

[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *In Proc. International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[9] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Springer Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2016.

[10] M. Martinez and M. Monperrus, "ASTOR: A program repair library for java," in *In Proc. International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[11] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13.