# Poster: USE: Unified and Split Execution for Exposing Divergences between Versions

Hongliang Liang, Wenying Hu
Yini Zhang, Lin Jiang
Beijing University of Posts and Telecommunications
Beijing, China
{hliang,wendia,zhangyn2012,jianglin}@bupt.edu.cn

Wuwei Shen
Western Michigan University
Kalamazoo, MI, USA
wshen@wmich.edu

## ABSTRACT

How to generate an appropriate set of test cases which can effectively show the difference between an old and new version of a program becomes a challenging research topic. In this paper, we consider both control divergence and data divergence to explore the difference between two versions of code. To do so, we present a novel model called Use, which executes the common code only once but has separate execution traces for the different code in a single dynamic symbolic execution instance, to generate test cases efficiently. Furthermore, we leverage Use to expose control and data divergences introduced by a patch. The initial experimental results show that our methods can efficiently and effectively generate test cases demonstrating the divergence between two versions of code.

## 1 INTRODUCTION

In regression test, developers want to know whether the changes or patches to a program fix the bugs, and how these changes affect the unchanged code in unforeseen ways.

Different from previous approaches [2, 4–7], our research goal is to identify the output divergence (i.e. both behavior/control and data divergence) between two versions of a program automatically and accurately. To achieve this goal, in this paper, we investigate control divergence and data divergence via a control-flow graph (CFG). Namely, after converting a program to a CFG, we study behaviors of a program based on a test case from the following two aspects: 1) (control aspect) a sequence of edges in the CFG of a program traversed during the execution; and 2) (data aspect) a set of pairs each of which consists of a value and a live variable produced during the execution. A set of pairs is checked when a function returns so the set is also called an external data summary (EDS) of a function. EDS provides the inter-procedure data change information produced by a function, thus becoming a piece of vital information for exploring data divergence. Consequently, *control divergence* is defined as the different sequences of edges traversed by two versions on the same test case. And *data divergence* is defined as the different EDS values of functions when functions return or exit.

Based on the key observation that two versions of a software system share a large portion of code, and with the success of dynamic symbolic execution techniques [1, 3], we propose a novel Unified and Split Execution model, called Use, which executes two versions of a program in a single dynamic symbolic execution instance, to generate test cases showing both divergences between versions.

## 2 APPROACH

With a test case covering a given patch, Use executes a unified program, which merge two versions via *change*() annotations [4], in one dynamic symbolic execution instance.

We present the execution modes and mode switches of Use, as shown in Figure 1. There are a unified mode (UE for short) and two split modes (SPEO and SPEN for short) in Use. There are multiple edges coming out of SPEO and SPEN, and so we attach a number for each edge for clarity. "▹" means the sequence order of events occurred. In the UE mode, Use executes the statements in the unified program shared by two versions, while in split modes, Use executes two different sequences of statements in the unified program.

The initial mode of Use is the UE mode. Use continues the UE mode till reaching an exit or a return statement, or finding a bug (U1 edge in Figure 1). If the two versions follow different branches in the same location, a control divergence is found and the location is called a *split point*. Therefore, Use enters a split mode in which Use executes an old version (U2 edge) followed by a new version of a program. The four-way forking strategy in [4] is used to explore the unified program and find control divergences between two versions.

Use switches from a split mode to the unified mode (O1▹N1 edge) when finding a new sequence of same statements in the unified program. The first statement to switch back to the UE mode, denoting an end of control divergences, is called a *merge point*. There are two kinds of merge points: conservative and optimistic merge points (omp for short). In general, a return statement in a function is considered as a conservative merge point since the executions of two versions finally reach. But in most cases, a merge point can be found before a return statement, and thus we call it an omp. For instance, if a split point contains function pointers, the next statement after the split point is an omp. Use attempts to find an omp as early as possible using the Algorithm 1.

Given the following inputs: the CFG of a function $F$ with a unique root node *entry*, and starting nodes to be executed after a split point in two versions, $s_o$ and $s_n$ respectively, the algorithm outputs a mapping ($M$) structure, which maps each omp to a set


*ICSE'18, May 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183419

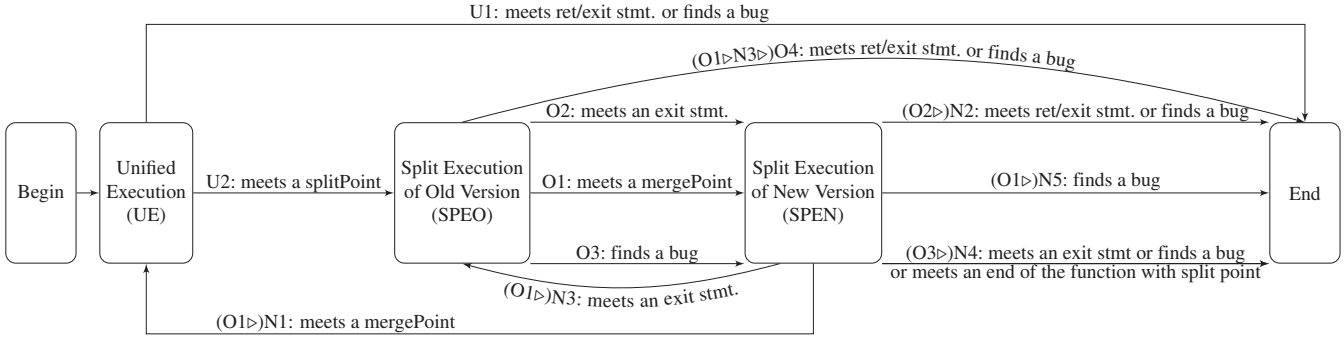


**Figure 1: The Execution Modes and Their Switches in Use**

of the reachable edges (RE for short) that represents all the paths from $s_n$ to the OMPs, and a set of RE *exitRE* that denotes all the paths from $s_n$ to the *exit* nodes (all leaf nodes in CFG except the return node). *cChilds* is the set of potential merge points. Function findMerge() removes from *cChilds* all redundant nodes to which all the paths from $s_o$ and $s_n$ must go through at least a different node in *cChilds*, and thus *mpSet* is the set of OMPs. To collect RE for a specific node pair $(start, end)$, first, getCE() searches all paths in a bottom-up manner via traversing the CFG from *end* to *entry* and stores the paths in *mpCE* or *exitCE*. Second, getRE() searches all paths in *mpCE* or *exitCE* in a top-down manner from *start* to *end* and stores the paths in $M[mp]$ or *exitRE*.

Use analyzes data divergence when a return/exit statement in a function is reached. As mentioned before, we mainly focus on the function-level data information (i.e. EDS) which includes global variables, arguments with pointer type and variables holding a return value of a function in that they can propagate values among the functions in a program. Use exposes the data divergences by checking the changes of EDS of a function. For each variable V in EDS, if it contains shadow expressions in its values, Use first constructs two symbolic expressions for two versions, i.e., $V_{seo}$ and $V_{sen}$ respectively. Next, a constraint $pc$ && $V_{seo} != V_{sen}$ is built, in which *pc* is the current path constraint. If the constraint is satisfiable, a test case is generated by an SMTsolver, and we claim that a data divergence is found between two versions of a program and Use also keeps the further analysis on data dependence.

## 3 PRELIMINARY EVALUATION AND CONCLUSIONS

We use the state-of-the-art tool KLEE as the baseline to compare with Use on CoREBench [8]. Results show that Use can generate test cases for exposing the divergences introduced by patches and explore 2%-50% of the paths explored by KLEE with the same input and time limit. Furthermore, we find that 1) changes to control flow may cause data divergence in some cases (e.g. ID-11 in CoREBench) but not in some cases (e.g. ID-13); and 2) changes to data flow may cause control divergences in some cases (e.g. ID-12) but not in some cases (e.g. ID-1). So we argue that taking both data divergences and control divergences into account is necessary for regression testing. We plan to build unified programs automatically and give Use more evaluation.

**Algorithm 1:** Finding OMPs and collecting reachable edges

**Input:** **CFG**: the CFG of a function; $\mathbf{s_o}$,$\mathbf{s_n}$: the starting node for old and new version.
**Output:** **M**: the mapping OMP → {RE}; **exitRE**: {RE} from $s_n$ to exit nodes.

```
1  Childs_o = getDescSet(CFG, s_o);
2  Childs_n = getDescSet(CFG, s_n);
3  cChilds = set_intersection(Childs_o, Childs_n);
4  if (cChilds.size() == 1) then
5      return ∅;
6  mpSet = findMerge(cChilds, Childs_n);
7  for each mp in mpSet do
8      mpCE = getCE(CFG, mp);
9      M[mp] = getRE(CFG, s_n, mp, mpCE );
10 exits = getExits(Childs_n);
11 for each exit in exits do
12     exitCE = getCE(CFG, exit);
13     exitRE = exitRE ∪ getRE(CFG, s_n, exit, exitCE);
14 return M, exitRE;
```

## ACKNOWLEDGMENTS

The work is supported by the National Natural Science Foundation of China under Grant No.: U1713212.

## REFERENCES

[1] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. CACM, 56(2):82-90, 2013.
[2] W. Le and S. D. Pattison. Patch verification via multi-version interprocedural control flow graphs. In Proc. ICSE'14, pp.1047-1058, 2014.
[3] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In Proc. ICSE'12, pp. 716-726, 2012.
[4] H.Palikareva,T.Kuchta,and C.Cadar,Shadow of a doubt: testing for divergences between software versions, In Proc. ICSE'16, pp. 1181-1192, 2016.
[5] D. Qi, A. Roychoudhury, and Z. Liang, Test generation to expose changes in evolving programs, In Proc. ASE'10, pp. 397-406, 2010.
[6] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In Proc. ISSTA'11, pp. 1-11, 2011.
[7] W. Jin, A. Orso, T. Xie, Automated behavioral regression testing, In Proc. ICST'10, pp. 137-146, 2010.
[8] M. Bohme and A. Roychoudhury. Corebench: Studying complexity of regression errors. In Proc. ISSTA'14, pp.105-115, 2014.