Building Resource Adaptations via Test-Based Software Minimization: Application, Challenges, and Opportunities

Arpit Christi
School of Computing
Weber State University
Ogden, UT, USA
arpitchristi@weber.edu

Alex Groce
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
agroce@gmail.com

Austin Wellman
Raytheon BBN
Boston, MA, USA
austin.wellman@raytheon.com

***Abstract*—Building resource adaptive software systems is a challenging problem. Researchers have proposed many techniques and tools to build such systems. We previously proposed a technique called Test-based Software Minimization (TBSM) that relies on using tests to define functionality that can be sacrificed to achieve resource gain. We demonstrate easy-applicability, usability, and effectiveness of TBSM by building resource adaptations for two real-world scenarios. We also discover significant challenges associated with the practical application of TBSM. Based on our attempt to overcome the challenges for two scenarios, we summarize possible solutions to the challenges.**

I. Introduction

Ultra Large-Scale Systems, Internet of Things applications, and Cyber-Physical Systems face increased resource variability compared to programs of the past, with real-world physical consequences. Resources may be explicit: e.g., CPU, memory, or storage; or they may be implicit, e.g., libraries and protocols. The inability of software to handle resource changes effectively and automatically can produce inferior or potentially vulnerable systems [1].

Self-Adaptive Software Systems (SAS) modify their own behavior in response to changes in operating conditions. Resource Adaptive Software Systems (RASS) are a subset of SAS where the trigger for adaptation is variability or unavailability of resources. For longevity and survivability of modern systems, self-adaptation for resource changes is essential hence the problem of self-adaptation is well studied [2]. To this end, DARPA started BRASS (Building Resource Adaptive Software Systems) program, a major initiative to bring researchers and practitioners together to solve the problem of building resource adaptive software.

Researchers have proposed many techniques, approaches, and tools to build SAS automatically [1], [3], [4]. We previously proposed a solution called Test-Based Software Minimization to build SAS to overcome some of the limitations of previous approaches based on our work with Raytheon developers as part of DARPA BRASS program [5]. In this work, we applied TBSM to produce adaptations of a mission-critical military system and the popular NetBeans `Java` IDE. Based on our observations:

1) We demonstrate that unlike most other previous approaches, resource adaptation via TBSM is a conceptually simple, usable, and applicable technique.
2) We present challenges we faced while attempting to produce accurate resource adaptations automatically.
3) We present the solutions we employed to solve some of those challenges.

In the process, we identify opportunities to further aid developers in effectively employing test-based minimization to achieve resource adaptations.

II. Background

While working with developers attempting to build SAS for a real-world, mission-critical system, we observe that (1) developers speak the language of tests (2) tests are the only semi-formal specification available. To exploit both, the widespread availability of tests for mission-critical systems and developers' familiarity with tests, we proposed capturing resource adaptation specifications using tests. TBSM relies on developers understanding of tests, and they relate to program features and resource usage. Developers encode this information by simply labeling the tests. Test labels can be a multi-dimensional space in feature, resource, and priority. To demonstrate the concept, we assume a single adaptation objective: a single resource faces variability or unavailibility. The concept is demonstrated in Fig. 1 (a simplified version of Fig 2. in the original work [5]). Labeled tests define functionality that can be sacrificed to reduce resource usage. Unlabeled tests define the functionality that is to be retained and not modified. A tool called hddRASS takes as its input the test suite with labeled and unlabeled tests and the original program, and produces a minimized program such that the functionality marked by labeled tests is removed. hddRASS achieves this by temporarily removing the labeled tests from the suite and using the Hierarchical Delta Debugging (HDD) algorithm to find a minimal program, such that retained tests continue to pass [6]. The basic idea is that if the labeled tests define a functionality that uses resources, removing that functionality will ensure resource adaptation. TBSM builds

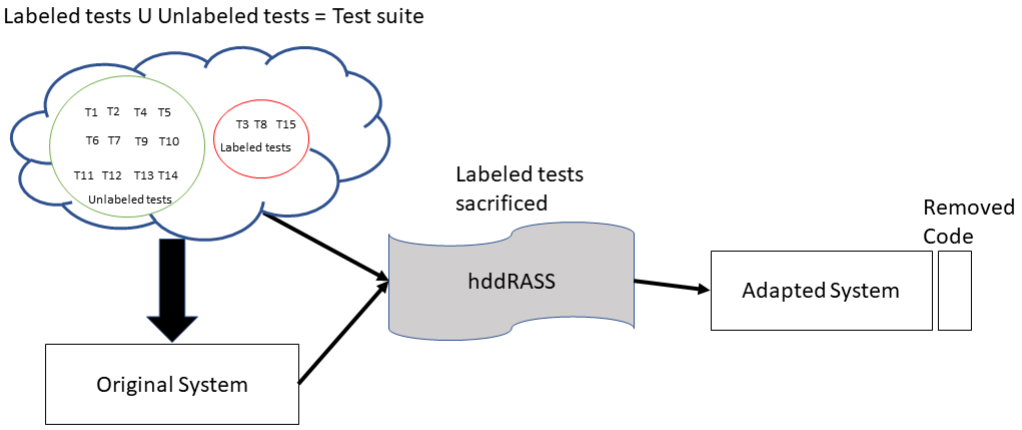

Fig. 1: TBSM approach to build adaptation for single adaptation objective scenario. Labeled tests encode adaptation specification for single resource variability.

adaptations, in a sense, in the same way that Automatic Program Repair fixes faults. In APR, the goal is to modify the program to pass a set of previously failing tests (without failing previously passing tests) [7]. In TBSM, the goal is to modify the program while removing as much code as possible, while still passing a set of unlabeled tests.

III. Case Study

Building an adaptive software system is not (yet) a common practice. The example benchmarks available as part of self-adaptive exemplars are unfortunately not applicable to TBSM, due to the nature of their tests and the resources adapted [8]. Hence, we rely on two real-world case studies.

We worked with a group of developers attempting to build adaptations for a mission-critical system called the Tactical Situational Awareness System (TSAS). To focus our discussion, we present one adaptation scenario, the *Elevation API* scenario. The resource under consideration is a library (an implicit resource) that has been updated to a new version. Developers labeled tests in terms of the feature tested. Certain features were marked as sacrificial because the new version of the library provides functionality that was originally part of the TSAS code. The sandboxed TSAS version that we used consists of 70 `Java` files, with 5,571 LOC, and developers labeled 5 tests as representing sacrificial functionality; the remaining tests were considered unlabeled. The developers used TBSM to build an adaptation.

For our next case study, we used the popular *NetBeans IDE* for `Java`. We discussed *NetBeans IDE* case study in detail in our original work [5]. The resource under consideration in this case was memory. To save memory, we adapted the IDE by disabling undo-redo functionality, which is potentially memory-intensive. Developers working in a memory-limited setting would prefer to lose undo-redo rather than face constant crashes due to memory exhaustion. Our target for adaptation is the `openide.awt` module, which consists of 69 `Java` files, 11,284 LOC, and 146 tests. After carefully studying the code and tests we labeled 3 tests as undo-redo related. We then applied TBSM to build a memory adaptive *NetBeans IDE*. As most of the 69 `Java` files are clearly not related to the undo-redo feature, we chose the 2 most likely files as reduction targets to reduce processing time [5].

IV. Evaluation

We evaluate resource adaptation via TBSM/hddRASS along several dimensions: effectiveness, usability, applicability, and scalability.

A. Effectiveness

For the *Elevation API* scenario, all 4 necessary modifications were correctly performed automatically by TBSM. The development team was able to confirm the adaptations for library change by (1) examining the code for necessary modifications (2) running the tests and (3) using the application. As TSAS is a proprietary system, we have to rely on conformation from the development team. All the necessary modifications to adapt the *NetBeans IDE* were also performed correctly. We observe that all 19 resource consuming statements (as identified in previous work [9]), the statements that fill the undo-redo buffers, were correctly modified by the technique. We have shown previously that the adapted IDE 1) cannot perform undo-redo operations and 2) uses far less memory, in a controlled experiment setting allowing only edits to a single

text file [5]. We also confirm otherwise normal operation of the IDE.

B. Usability

Most adaptation approaches require developers to learn a new specification language or add complex annotations to code. To use such techniques, developers typically need to use modeling approaches, architectural specifications, formal specificatios etc., to map their applications; obviously a time consuming and error-prone task [3], [4]. TBSM only requires developers to look at their tests and make a "good guess" about the feature(s)/resource(s) relevant to each test. Developers are familiar with tests and can likely perform this task without additional training. For the *NetBeans IDE* case, it took us only a few hours to determine 3 tests to label out of 146 tests, despite the fact that we are not developers. For the *Elevation API*, actual developers were instantly able to determine the tests pertaining to certain features.

C. Applicability

While developing the RAINBOW framework, Garlan et al. noted that most previous approaches for SASS were scenario-based or application-specific, and not very reusable [10]. To evaluate applicability, we measured the effectiveness of hddRASS as a tool. We believe hddRASS to be complete for `Java` 7: it applies to any programs written in `Java` 7. To confirm this, we checked for thrown exceptions or unprocessed statements when minimizing real-world systems. While applying hddRASS to TSAS, we processed 70 `Java` files with 338 non-constructor methods. For the *NetBeans IDE*, we processed 2 `Java` files with 38 non-constructor methods. We also applied hddRASS to 40 randomly selected classes with a total of 1,207 methods across 10 open source projects, using a random labeling scheme [9]. We observed neither exceptions nor unprocessed statements across these experiments. Based on this, we can say that hddRASS as a tool, and TBSM as a technique, is generally applicable to `Java` applications. To extend it to other programming language, one needs to build hddRASS like tool for that programming language.

D. Scalability

For the *Elevation API* the application of hddRASS on all 70 `Java` files took 7 hours 50 minutes. For *NetBeans IDE*, adapting 2 files in `openide.awt` took 2 hours 35 minutes. As the program size increases or test suite runtime increases, scalability becomes the limiting factor in TBSM. For offline adaptations, scalability may not be a limiting factor. But for live systems deployed in the wild, the system needs to be halted until adaptations are performed and a long offline period is not normally acceptable.

V. Challenges

We describe the major challenges that we faced while applying TBSM to build real-world RASS, and how we attempted to solve these challenges for the case study scenarios. In the process, we identify major research challenges in TBSM. We note the similarity between the challenges in TBSM and the challenges observed by researchers in APR [11].

A. Search Space

TBSM indiscriminately processes all the statements of a program, so the search space is accordingly vast. This can be significantly reduced by selecting only statements likely to actually be modified. We empirically evaluated large number of adaptation related modification for 800 synthetic adaptations to derive heuristics to guide target selection. We derived statistics-based heuristics (H1, H2) and dynamic-analysis-based heuristics (CBLS, AdFL) based on our evaluation [9], [12]. CBLS heuristic relies on coverage information of labeled and unlabeled test suite. We repurpose Spectrum-Based Fault Localization (SBFL) to derive AdFL heuristic by establishing equivalence between core components of SBFL and core components of TBSM. We found that dynamic-analysis-based heuristics perform better in empirical analysis. For *Elevation API* scenario, CBLS heuristics reduces search space by 55% and AdFL heuristics reduces search space by 92% bringing the wall clock time to build adaptations down from 460 minutes to 118 minutes for CBLS and 49 minutes for AdFL. For *NetBeans IDE* scenario, CBLS heuristics reduces search space by 90% and AdFL heuristics reduces search space by 94% bringing the wall clock time down from 175 minutes to 61 minutes and 57 minutes for CBLS and AdFL respectively.

CBLS heuristics provides search space reduction while AdFL prioritizes and sorts the search space based on likeliness of modification. We demonstrated that the likely target of modifications would appear earlier in the sorted order if we use AdFL [12]. We utilized this fact to circumvent the search space issue all together. We proposed best-effort incremental TBSM as a technique were developers provide a time limit to finish the adaptations [12]. The best-effort incremental TBSM will always produce a useable system in the given amount of time with resource adaptation achieved entirely or partially. For *Elevation API* and *NetBeans IDE* scenarios, a time limit of 35 minutes and 20 minutes were sufficient to perform complete resource adaptation using Best-effort incremental TBSM. Best-effort incremental TBSM performs better than AdFL heuristics with 90% search space reduction for both the scenarios.

B. Test Suite Runtime

Like APR, TBSM is a generate-and-validate technique that must execute a potentially large test suite to evaluate each modification. TBSM benefits from a large test suite, but running all tests is often prohibitively slow. The version of *NetBeans IDE* we used has a test suite that requires 7+ hours to run for all 1193 modules. TSAS also exhibits unacceptably slow complete-test suite runtime. For both case studies, we observe that running only modified-module-related tests is sufficient as modules are self-contained and have good individual test suites. We observe that even such a simple ad-hoc test selection reduces test runtime by 34 seconds

and 18 seconds for the *NetBeans IDE* and *Elevation API*, respectively. For most resource adaptation scenarios, such an ad-hoc technique is not feasible and scalable. We need test selection and prioritization (since early failures also limit runtime) tailored to TBSM.

C. Inefficient Algorithm

hddRASS uses a modified HDD* algorithm with worst-case running time of $O(n^3)$ where n in our case is statement nodes of the abstract syntax tree of the program [6]. We employ HDD* algorithm as the underlying driver for TBSM because it guarantees convergence and minimality. By minimality, we mean that any output program produced by TBSM is minimal and cannot be modified any further. One solution we employ is to forgo the minimality guarantee of HDD* and use a pure greedy search, trading accuracy for efficiency. With a greedy search strategy, we can build resource adaptations 1.72 times faster while retaining 94% accuracy for *Elevation API* and 1.90 times faster while retaining 100% accuracy for *NetBeans IDE*. We need further evaluation of different search strategies and heuristics-based modifications to HDD*.

D. Overfitting

The issue of overfitting is well studied in APR, and we face the same problem[13]. TBSM produces test-adequate adaptations, adaptations that are correct with respect to a test suite and test labeling scheme. For our case studies, Type 1 errors are rare, and even related statements are usually removed; e.g., if a single resource consuming statement is the body of a for loop, the loop is removed as well. We consider directly or indirectly resource-adaptation-related modifications as true modifications. We do observe a large number of false modifications, modifications that are not directly or indirectly related to resource adaptations: these are Type 2 errors. Indeed, the false modifications outnumber true ones in our scenarios. We observe 49 and 121 modifications are performed by TBSM for the *Elevation API* and *NetBeans IDE* respectively. We observe 91% and 51% false modifications for the *Elevation API* and *NetBeans IDE* adaptations, respectively. As TBSM generated test-adequate correct adaptations overfit to a given test suite and labeling scheme, it is vulnerable to both (1) inadequacy of test suite (2) labeling errors. Developers reported instances where TBSM generated modifications removed untested but desired functionality.

VI. Test Adequacy and Adaptation Quality

As developers using TBSM express more concern over overfitting than any other problem, in this section we discuss overfitting as a result of test inadequacy and our attempts to solve it for the case study scenarios.

A. Test Inadequacy

Because resource adaptations generated by TBSM are test-adequate, if the tests are inadequate, we are bound to produce sub-standard adaptations. The primary problem with test inadequacy is, as we can see, overfitting: Alex: Simplify rest of the statement, too difficult to read for someone who is not familiar with our work. when there is no test for a feature that is not targeted for removal, or when the only tests for such a feature are mixed with a test for a feature that is to be sacrificed, TBSM has no way to distinguish this case from the code for that feature being part of the sacrificed functionality. Based on our observation of overfitting modifications in our case studies, we propose some basic guidelines for quickly improving a test suite to help TBSM distinguish between true and false modifications.

B. Code Coverage

TBSM will remove any code that is not covered by any tests (not removed from the suite) and not required for compilation. The easiest way to avoid this problem is to start with a high-coverage test suite. Developers can target poorly covered code with manual tests or use targeted test generation tools to cover such code. Improving code coverage using random testing reduced overfitting by 23% and 8% for *Elevation API* and *NetBeans IDE* respectively. Naïve test generation will not necessarily improve matters, however. Developers have to label any generated tests, and if tests mix multiple features, there still may often be cases where a feature not to be removed is never tested in isolation. We propose using delta-debugging to help isolate features in generated test cases [14]: if a generated test covers two features, perhaps one can be removed by reducing the test with respect to targeted code for one feature only.

C. Better Tests

Based on our study (with the developers) of overfitting, we identified a few categories covering most of the incorrect removals. For the *Elevation API*, there were no tests for logging, and so all LOG statements were removed. This can be easily remedied by explicitly testing system logging as a feature. Similarly, exception-handling code was often inadequately tested. We worked with the developers to produce a new test covering each major overfitting category identified. We produced just 3 and 4 tests, respectively, for the *Elevation API* and *NetBeans IDE*, resulting in 20% and 66% reduction in overfitting. Identifying overfitting categories can be a highly context-specific task, but we suspect new automated test generation methods targeting specific under-tested categories such as logging and exception handling may be useful here.

VII. Tool and Data Availability

hddRASS tool is available at https://github.com/amchristi/hddRASS. It can be utilized to execute the baseline TBSM technique or to use the TBSM technique with heuristics guidance (H1, H2, CBLS, or AdFL). It also provides support to execute TBSM in greedy mode. As TSAS is a proprietary system, we cannot make its source code, test suite, or tests labels available. NetBeans IDE (and hence `openide.awt`

module) source code is available online. We provide the exact `openide.awt` module source code, test suite and test labels that we utilized available at: https://github.com/amchristi/AdFL. The same link contains 800 synthetic adaptation scenarios with source code, tests, and test labels for empirical evaluation and comparison.

VIII. Ongoing Work and Future Directions

As part of DARPA BRASS project, we work with multiple software development teams and other collaborators. Our communication with these stakeholders as well as the feedback that we received from the development teams utilizing our work are driving our ongoing work and future research direction.

A. Ongoing Work

1) Improving Applicability: The underlying tool hddRASS that drives TBSM is written for `Java` program only. Raytheon development team utilizes TBSM for TSAS, an application written in `Java`. We also developed a `C++` version of hddRASS that is intended to be used by ROS (Robotics Operating System) application developers to adapt against ROS version changes and other ROS package changes. Developers can use it to build any different kind of resource adaptation for `C++` applications by providing correct test labeling. We plan to publish it soon.

2) Minimization vs Modification: The primary concern that any development team might raise before evaluating TBSM to build resource adaptation is: It only offers minimization(reduction). When the software system adapts itself, the adaptation manifests itself in 3 ways (1) reduction (2) replacement (3) enhancement [1]. The published TBSM version only supports reduction. While developing a `C++` version of hddRASS, we incorporated many other modification operators studied and used by APR to fix the fault, making few replacement capabilities available [15], [16], [17]. We plan to continue to improve hddRASS to provide more modification capabilities.

B. Future Direction

We are currently investigating multiple other ways to speed up TBSM, including using static and dynamic analysis to precompute the effects of program statements on test oracles and using test case selection and prioritization to reduce the running time of tests in TBSMs generate-and-validate loop.

The original work in TBSM emphasized the need for a good test suite. Previous heuristics suggested coverage as a way to define goodness of a test suite; the CBLS heuristic depends on coverage information. Because of the success of AdFL heuristics in isolating and prioritizing modification targets, we plan to consider test suite diagnosability metrics as a more refined way to define the goodness of a test suite for TBSM, using approaches proposed by Baudry et al. and Perez et al. [18], [19]

For our work, we mostly utilized existing test suites provided by the developers. For such test suites, we observe that sometimes a test that developer labeled as pertaining to a particular feature may contain code that exercises other features also. Similarly, an unlabeled test, apart from testing functionality that needs to be retained, may exercise sacrificial feature. Availability of such tests makes it harder for TBSM to differentiate between an adaptation related modification and a pure accidental modification, resulting in underfitting or overfitting. To mitigate the situation, we plan to utilize (1) Test-case purification to segregate tests by features [20] (2) Test-case reduction using delta debugger to remove irrelevant features from tests [14], [21].

IX. Conclusions

Test-based software minimization offers a conceptually simple, widely applicable, easily understood approach to generating resource adaptations. While TBSM faces significant scalability challenges, and cannot be applied where tests are impossible to label, or to resources not associated with removable features, it also benefits from a powerful synergy: namely, the best way to improve TBSM is to improve software test suites. Making test suites faster to execute, improving their granularity so that tests cover distinct features, increasing code coverage, and improving test prioritization techniques all lead not only to more efficient and effective TBSM, but to better fault detection and debugging.

Acknowledgments: this work was partly funded by the DARPA BRASS [1] program, and the authors would like to thank our collaborators at Oregon State University and Raytheon/BBN.

References

[1] J. Hughes, C. Sparks, A. Stoughton, R. Parikh, A. Reuther, and S. Jagannathan, "Building resource adaptive software systems (BRASS): Objectives and system evaluation," *SIGSOFT Softw. Eng. Notes*, vol. 41, no. 1, pp. 1–2, Feb. 2016.

[2] D. Hughes, "Seams 2018 keynote speech," https://conf.researchr.org/track/seams-2018/seams-2018-papers#program, accessed: 2018-08-09.

[3] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[4] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive Mob. Comput.*, vol. 17, no. PB, pp. 184–206, Feb. 2015.

[5] A. Christi, A. Groce, and R. Gopinath, "Resource adaptation via test-based software minimization," in *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, Sept 2017, pp. 61–70.

[6] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 142–151.

[7] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018.

[8] "Self-adaptive systems artifacts and model problems," https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/, accessed: 2018-10-05.

[9] A. Christi and A. Groce, "Target selection for test-based resource adaptation," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2018, pp. 458–469.

[10] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct 2004.

[11] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, Sep 2013.

[12] A. Christi, A. Groce, and R. Gopinath, "Evaluating fault localization for resource adaptation via test-based software modification," in *IEEE International Conference on Software Security and Reliability*, 2019, to be published.

[13] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 532–543. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786825

[14] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: Delta debugging, even without bugs," *Journal of Software Testing, Verification, and Reliability*, vol. 26, no. 1, pp. 40–68, Jan. 2016.

[15] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. ACM, 2009, pp. 947–954.

[16] A. Arcuri, "Automatic software generation and improvement through search based techniques," Ph.D. dissertation, University of Birmingham, UK, 2009.

[17] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. IEEE Computer Society, 2010, pp. 65–74.

[18] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. ACM, 2006, pp. 82–91.

[19] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 654–664.

[20] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 52–63.

[21] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*, 2018, pp. 184–191.