Evaluating Fault Localization for Resource Adaptation via Test-based Software Modification

Arpit Christi
School of EECS
Oregon State University
Corvallis, Oregon, USA
christia@oregonstate.edu

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

Rahul Gopinath
CISPA
Helmholtz Center for Information Security
Saarbrücken, Germany
rahul.gopinath@cispa.saarland

***Abstract*—The ability to dynamically adapt to resource variations is critical for modern-day mission-critical systems that operate in ever-changing resource environments. Test-based Software Modification (TBSM) is a recently proposed technique to build Resource Adaptive Software (RAS) that relies on existing test infrastructure, test labeling, and program modifications. TBSM is simple and applicable, but an inefficient technique; the primary reason for inefficiency is the sheer size of the search space.**

In this paper, we propose AdFL, a repurposing of Fault Localization (FL) that can shrink (and prioritize) the search space for TBSM more effectively than previously proposed heuristics. We present complete case studies and an empirical analysis of a set of open source projects as evidence that AdFL can significantly reduce the search space in TBSM. We show how to combine AdFL with previous heuristics for TBSM, and propose an incremental, best-effort variant of TBSM that uses AdFL to prioritize the search.

I. Introduction

Modern day software systems have varying, complex resource needs that change frequently based on environmental variations, technology, and integrating system components. To ensure survivability in changing resource contexts, these systems must self-adapt based on changing resource needs and availability [1]. The inability of software systems to handle varying resource needs can lead to inferior and potentially vulnerable software. A *self-adaptive software* (SAS) changes its behavior in response to changes in operating conditions. A *resource-adaptive software* (RAS) is a SAS where the reason for adaptation is unavailability or variability of one or more resources. Researchers are devoting significant efforts to devise methods to effectively construct resource-adaptive software systems, and DARPA has launched BRASS (Building Resource Adaptive Software Systems), a major initiative devoted to the problem [2].

Researchers have proposed numerous tools, techniques, and approaches to build SAS that rely on modeling techniques, architectural specifications, domain-specific languages, and formal methods [3], [4]. While numerous tools exist, these tools are mostly domain or application specific. Further, they are seldom reusable nor sufficiently applicable to different systems [5], [6]. These tools usually require developers to learn modeling techniques, formal specification methods, domain-specific languages, etc., and map their applications accordingly [7], [8], [9], which presents a high barrier to entry.

While working with software developers building real-world RAS for a mission-critical software system, the Tactical Situational Awareness System (TSAS), Christi et al. proposed *a conceptually simple but highly applicable* technique called *Test-Based Software Modification* (TBSM) [10], inspired by delta-debugging variations [11], [12]. TBSM is similar to Automatic Program Repair (APR) for patch generation and relies on existing test infrastructure, combined with automatic program modifications, to build adaptations [13]. The technique relies on developers' understanding of tests and how tests relate to features and resource usage. Developers encode this information by merely *labeling the tests* related to functionalities. As developers do not need to learn any modeling or architectural technique, any specification languages, formal methods etc. *the entry barrier is low to developers*.

To simplify presentation and analysis, we assume in the remainder of this paper that a single resource change requires adaptation. Figure 1 is a simplified version of a figure in the original TBSM paper [10], explaining the basic idea of the approach. Tests in the labeled set mark tests that encode the adaptation objective: they test a functionality that is to be removed. The unlabeled tests define functionality the adapted system must retain. The reduction tool hddRASS takes as its input the original program and labeled and unlabeled test suites, and produces a "minimized" program that passes all the unlabeled tests, but does not have to pass the labeled tests. Ideally, this is a program with the targeted functionality removed. The operation of hddRASS is conceptually simple: it repeatedly removes parts of the code, and checks if the modified program still passes all unlabeled tests. If so, this becomes the new baseline program, and hddRASS attempts to minimize it, until no changes produce a program that still passes all required tests. If the labeled tests define a functionality that consumes resources, the modified program will consume fewer resources.

The major drawback of TBSM is efficiency — it is a slow technique. Since building resource adaptations with TBSM is similar to patch generation with APR, it has similar reasons for inefficiency — (1) an extremely large search space, (2) potentially long running time for test suites, and (3) inefficiency due to the underlying algorithm, a modified form of Hierarchical Delta Debugging (HDD) [14], which has worst-

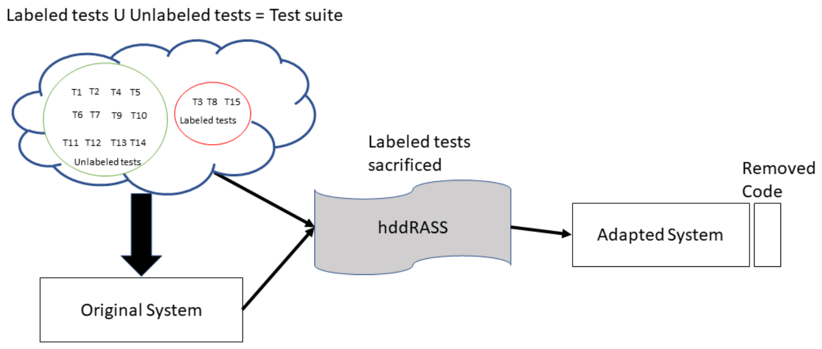

Fig. 1: TBSM approach to build adaptation for single adaptation objective scenario.

case complexity of $O(n^3)$. Of these problems, the sheer size of the search space is perhaps the worst (and drives HDD complexity): the search space in TBSM is the set of possible modifications to a whole program, and since it is a generate-and-validate technique, in principle TBSM may have to run the entire unlabeled test suite for every step of the search. We proposed three heuristics [15] to reduce the search space, of which one, called H3 or CBLS (Coverage-Based Likely Statements) relies on statement coverage information of labeled and unlabeled tests. This heuristic performed best in terms of both accuracy and reduction in search space size, often by a large margin. We therefore use CBLS as the starting point for our exploration of improving the efficiency of TBSM. Even with CBLS, computing an adaptation still required over an hour in a realistic case study.

While proposing Fault Localization (FL) as an early step of APR to tackle the search space problem, Le Goues et al. noted it as a significant repurposing of FL — the original purpose mainly being automated debugging [16]. The presence of the fault and the availability of passing and failing tests makes FL a natural choice for APR. Unfortunately, with TBSM, neither is present: TBSM aims to generate a minimal program to fit a test suite, not cause failing tests in a suite to pass. In this paper, we argue that the presence of labeled tests can be used to overcome this limitation. We propose changes in the definition of Fault Localization, primarily in the usage of passing and failing tests, to repurpose it for TBSM. We call this tweaked FL Adaptation FL, or AdFL. We demonstrate that AdFL can successfully predict which statements will be modified (removed) during adaptation more effectively than CBLS. Moreover, we show how to modify the TBSM process to incorporate the information provided by AdFL to produce accurate adaptations much more quickly. The contributions of this paper are: (1) We repurpose fault localization to tackle the search space problem for resource adaptations via Test-based Software Modification. This is accomplished by modifying the definition of FL. (2) We evaluate five different FL techniques repurposed for AdFL. (3) We conduct an empirical analysis to study the usefulness of AdFL on 800 data points across 40 subjects of 10 open source Java projects, using two test labeling schemes. (4) Using two real-world adaptation scenarios, we demonstrate that AdFL can predict modifications for real systems. (5) We consider the *stopping rule* problem for TBSM, and propose some heuristics, as well as an incremental version of TBSM.

II. Related Work

A. Self Adaptive Software Systems

Self-adaptive systems are well described in two roadmaps (Cheng et al. [17] and Delemos et al. [18]). Salehie and Tahvildari provide a summary of much early work in SAS [3], and describe some of the notable techniques proposed to build SAS. Krupitzer et al. summarize different engineering approaches to build SAS [4]. The approaches presented include model-based approaches, architectural-based approaches, control theory-based approaches, and learning-based approaches. The above-mentioned techniques, tools, and approaches target specific scenarios or applications with limited applicability and little reusability. This was noted by Garlan et al. while developing the RAINBOW framework [9], an early attempt to solve the reusability issue, and relies on architectural modeling combined with control and utility theory.

Elkhodary et al. argue that using feature or functionality as the core building block in SAS construction can alleviate some of the key challenges by abstracting underlying complexities [19], and develop a feature-oriented SAS framework [20]. Fredericks et al. suggest elevating tests to the status of first class citizens in SAS specification and verification. They propose MAPE-T, a test aware feedback loop where T stands for tests [21]. Self-adaptive software requirement specification is an open research question. Formal specifications, modeling languages, and domain-specific languages have been proposed [22], [23], [24]. Whittle et al. developed a DSL called RELAX that provides expressions to capture uncertainty in requirements [24]. Our work, in contrast, focuses on improving the performance of a recently proposed technique, TBSM, that avoids the need for additional specification, modeling, or architectural information.

Casanova et al. discuss the issues and limitations of utilizing FL to diagnose unobserved components in self-adaptive systems where the information collected for diagnostics may be insufficient or incomplete [25].

B. Fault Localization and Program Repair

Fault Localization ranks statements by the likelihood of the statement being faulty. Jones et al. used *spectra* of passing and failing tests to define the Tarantula technique [26]. Following that seminal work, researchers have proposed many (Spectrum-Based) FL techniques. Wong et al. summarize in detail recent advances in FL: according to Wong et al., Tarantula, Ochiai, Barinel, Op2, and DStar are the most studied FL techniques [27].

The recent surge of interest in Automated Program Repair (APR) largely began with GenProg, which modifies the Abstract Syntax Tree (AST) of a program until all tests in a suite pass and the fault is (presumably) fixed [13]. Le Goues et al. subsequently identified multiple major issues with APR, with a key issue being the search space problem [16], and argued that it is sufficient to modify only *likely faulty statements*. They mentioned FL as an imperfect, but best-available technique for the purpose. Further research in APR frequently uses FL as the first step of APR in order to tackle the search space problem [28], [29], [30], [31]. While evaluating different size search spaces, Long and Rinard noted that a small fault space might result in missed faulty statements while a larger fault space results in overfitting, making selecting the ideal fault space a difficult problem [32]. In their evaluation of FL techniques using APR, Qi et al. raised concerns about the usefulness of FL as an early step of APR [33]. Rather than using FL to improve APR, we modify FL and re-purpose it to the needs of TBSM.

III. Adaptation FL

Fault Localization (FL) is a natural fit for Automatic Program Repair (APR) because both methods assume (1) there is a fault, identified by at least one failing test and (2) there are both passing and failing tests to focus attention on the faulty aspects of a system. Both are missing when building adaptations using TBSM: the original program is assumed to pass all labeled and unlabeled tests, and a proposed adaptation step may not pass any tests. In this section, we present core concepts behind TBSM, the process of building an adaptation, and the use of labeled and unlabeled tests in the process. In doing so, we will construct a mapping between the core components of FL (*fault*, *passing tests*, *failing tests*) and core components of TBSM (*adaptation*, *labeled tests*, and *unlabeled tests*).

We define *minimization* as the process of applying hdRASS/TBSM to reduce a program's size. The modified program produced by applying TBSM is called the *adaptation*. The set of program statements modified (usually removed) by applying TBSM is called the *modification*; the modification is the diff between the original program and the adaptation. Tests that are marked as pertaining to a feature to be removed from the program are called *labeled tests*. Labeled tests are also known as removed tests, as the TBSM process involves removing them from the test suite so the program is no longer required to pass those tests. Tests that are not labeled tests are called *unlabeled* or *retained* tests. Adaptation via TBSM guarantees that the adaptation passes all unlabeled tests.

At a high level of abstraction both APR and TBSM modify a program until a certain set of tests passes. APR starts with a program that does not pass all tests, and aims to modify it so that it passes all tests; TBSM starts with a program that does pass all tests, and aims to modify it until it can no longer be modified and still pass all unlabeled tests. There are other differences — e.g. in APR we expect repairs to be small, involving changes to only a few lines of code, while in TBSM changes may be very large, depending on the functionality in question — but this is the essential difference. Our problem, then, is to map FL for APR to the different setting of TBSM.

A. Mapping FL to TBSM

1) Faults and Modifications: The underlying idea of spectrum-based FL is to identify code that is likely to be faulty; however, another way to think about this idea is that FL aims to identify code that is *part of the fix to a fault* — almost by definition, faulty code is code that needs to be changed.

The "faulty code" in TBSM is code that needs to be modified or removed for an adaptation: the difference between the original program and the adaptation. Such code is even, in fact, "faulty" when seen from a larger perspective. The purpose of resource adaptations is to avoid faults (either in correctness or performance) that occur in low-resource settings. We may not have any tests that represent such a scenario, but nonetheless we can easily conceive that code that should be modified in TBSM is the code that is involved in some resource-based fault scenario. E.g., if a program crashes because it allocates too much memory in a low-memory environment, we can say that the code that is 1) not essential to the functioning of the system and 2) allocates memory is *all* faulty. TBSM (and other resource adaptation) can be seen as APR for such faults. However, there remains a very significant difference between the techniques: in APR there is at least one test case that fails due to the fault being repaired. TBSM instead begins with a functionality to remove, since adaptation is often a pre-emptive effort rather than a response to a particular concrete failure. APR does not address purely "hypothetical" bugs, obviously, while minimization, in contrast, operates without access to a concrete failure scenario.

2) Failing Tests and Labeled Tests: FL for APR relies on the existence of at least one failing test, and identifies code more associated with failing than passing tests. In TBSM, we do not have failing tests. However, we do have labeled tests, and these can serve the same purpose. Recall that our failures, while unavailable, are due to resource uses of the functionality to be removed. This means that code more associated with the removed functionality is the code of potential interest for our repair. Not all code in labeled tests is resource-using, or related to removed functionality, but this is exactly the situation in FL in general: most code executed by failing tests is not faulty; the problem is to identify code that is somehow (causally, statistically, etc.) more related to failing than passing tests, and thus highly suspicious. Given this understanding, it is clear that *the "failing tests" in FL for TBSM are the labeled tests*, the tests that will be *allowed* to fail in the adaptation. Note that we are *using tests that are not executed at all in the minimization process* to approximate the *goal of that process*.

3) Passing Tests and Unlabeled Tests: Given the above mappings, it should be clear that *unlabeled tests, the tests the adaptation must still pass, correspond to the passing tests in FL for APR.* These tests serve as the "background" for the "figure" of the labeled tests.

4) Putting it all Together: FL for TBSM: To perform FL for TBSM, Adaptation FL (AdFL), then, we apply any fault localization algorithm, but replace failing tests with the labeled tests and passing tests with unlabeled tests.

To make the mapping here concrete, consider one of our case studies in IV-B, adapting the NetBeans IDE by removing undo/redo functionality. The labeled tests for the functionality do not fail, but they all (1) test an undo and/or redo action and (2) allocate memory for an undo buffer. The unlabeled tests never test undo/redo actions, but do allocate memory for the undo buffer in many cases. The strong association of the "fault" (undoing/redoing) with the labeled tests should still make effective FL algorithms label code that implements undo and redo as suspicious.

5) The Need for a Stopping Rule: CBLS has one obvious advantage over FL. CBLS provides a set of statements, not a ranking. In CBLS it is clear how the search space is reduced: only statements in the CBLS set are considered for modification. AdFL provides a *ranking* of statements. Having a ranking is clearly useful in TBSM, providing an ordering attempting to modify/remove statements, and making the program smaller much more quickly, but it does not, on its own, reduce the search space . To reduce the search space, we also need a *stopping criteria*. Constructing such a criteria, however, is not a simple matter, and previous work in FL for APR does not, in a sense, *need* such a criteria: when the modified program passes all tests, APR can stop. The only use for a stopping criteria is to abandon the search and consider the program unrepairable. In TBSM, however, we do not have such a convenient way to detect when we are done.

As it turns out, one (weak) stopping criteria requires no empirical data to justify. Statements not in CBLS are, by definition, never covered by any labeled test. This has two consequences. First, for any widely used SBFL approach we are aware of, such statements will have a suspicousness score of 0.0, and thus rank at the bottom of any ranking of statements in AdFL. Second, intuitively, it seems absurd to consider a statement not covered by any test "defining the functionality to be sacrificed" as a candidate for modification. Such statements are clearly, if our tests are any good, not part of the functionality we are adapting. Therefore, as a simple, default, stopping criteria for using AdFL in TBSM, we ignore all statements not in CBLS, as these statements will always have a suspicousness of 0.0. In fact, when statements are both covered by unlabeled tests (so presumably part of non-sacrificed functionality) and not covered by labeled tests, it seems *dangerous* to remove them. We therefore call AdFL using CBLS to limit the space *Guarded* AdFL (G-AdFL) because it guards against this possibility. This stopping criteria also points out an interesting corner case. CBLS includes "dead code" statements not executed by any labeled or unlabeled test. For many FL formulas (e.g., Tarantula) this results in an undefined suspiciousness. We assign such code a suspciousness of $\top$, since we know hddRASS without heuristic guidance will always remove such statements.

IV. Experiments

Having defined a procedure for Adaptation FL, we need to determine if the proposed equivalences are useful for building adaptations using TBSM. We demonstrate the utility of AdFL using two real-world single adaptation objective scenarios, building class-level adaptations in a controlled setting. We also show that AdFL performs well for a large set of synthetic adaptation problems using open source subjects used in previous evaluations of TBSM improvements [15].

In what follows, the *baseline* is the result of the application of TBSM in its original form without any heuristic or AdFL guidance. *CBLS* refers to application of TBSM where the search space is first reduced using the H3/CBLS heuristic [15], and *AdFL* to application of TBSM where the search space is first reduced using our new AdFL technique. We use the five most commonly studied SBFL techniques as our FL algorithms for AdFL [27]. Hence, for each AdFL, we produce five separate results.

A. Case Study: TSAS Elevation-API

For TSAS, we present a scenario that the development team calls the *Elevation-API*. Here, the *variable resource* is one of the libraries and the *variation* is the availability of a newer library. Via test labeling, the developer indicated sacrificable-functionality. With the availability of a newer library, certain features implemented in TSAS are implemented by the library, making the TSAS implementation code redundant. The test labeling here was in terms of features only, not resources. The sandboxed TSAS server component that we used consists of 70 Java files and is 5571 LOC in size. The developer labeled 5 tests, and the remaining tests were considered unlabeled. We applied the *baseline*, *CBLS*, and *AdFL* techniques to produce adaptations. Developers confirmed that all four necessary modifications were performed correctly by all the techniques, and that the adapted TSAS versions all worked correctly.

B. Case Study: NetBeans IDE undo-redo

The original work on TBSM discussed a NetBeans IDE undo-redo adaptation scenario in detail where memory consuming functionality was correctly modified to preserve memory [10]. The module under consideration, the `openide.awt` module, consists of 69 Java files with 11,284 LOC and 146 tests. We continue to use the three labeled tests that were labeled as *undo-redo feature related* in the original work. We applied the baseline, CBLS, and AdFL TBSM techniques to build a memory-adaptive NetBeans IDE. We confirmed, again, that all the techniques correctly removed all 19 resource consuming statements (as identified in previous research [15]). By building the adapted NetBeans IDEs and using them for a while, we were also able to confirm the normal operation of NetBeans IDE with undo-redo ability removed.

C. Synthetic Adaptation Analysis

We also compared AdFL to other methods using synthetic scenarios used in previous TBSM research. This involves 800 adaptations of 40 subjects from 10 open source Java projects, using two random labeling schemes. Our intent with the synthetic problems is not to produce practically useful adaptations, but to produce accurate adaptations for a given test suite and labeling. We control all variables such as test suite, labeling scheme, and class subjects. The only variation allowed is the choice of search space selection technique.

1) Subjects and Tests: Previous work on TBSM noted that at the class level, adaptations are meaningful and useful [10]. Hence, we focused our study on class-level adaptations. Subject details are available in our previous paper on TBSM heuristics [15]. Subjects have an average of 387 LOC (for the whole class) and 132 statements in non-constructor methods.

Previous work noted that arbitrary subsets of tests are not interesting (reduction will usually target some actual functionality), and some tests are not relevant to a particular class. Based on this, previous work used direct coverage as a criterion for test selection, and we adopted the same approach. However, in our case, test selection is not relevant as long as tests and labels remain the same across techniques. We have 24 tests per subject class, on average. For 32 out of 40 subjects, statement coverage of the tests is more than 80%, and it is more than 70% for 37 subjects. We can, therefore, say that we generally have subjects with good coverage. This is an important experimental criteria, since TBSM would only be applied to systems with good test suites.

2) Procedure: For each class, we first randomly labeled 10% of the tests and computed the adaptation. By keeping the test suite and labeling same but using the search space defined by the CBLS and AdFL techniques we repeated the process (details in Section IV-C3). We repeated this procedure 10 times for each class, randomly labeling tests each time, yielding 10 results per class, generating 400 results for 40 subjects. During each run, if the labeled tests overlapped with the unlabeled tests, or concerned only minor functionality, modifications are minimal, but if a very important test is labeled the modifications may be significant. Labeling tests randomly and repeating the process 10 times provide us with an idea of typical results. Developers label tests based on some feature the test targets. Such labeling is highly context-specific, and lacking in our open source projects' test suites. Because we are using random labels, rather than developer-provided labels, we lack a solid basis for guessing the size of a typical set of labeled tests representing a feature. Hence, we repeated the same procedure, but with 20% of the tests labeled, to produce another 400 adaptations. For each of the 800 data points, we have 3 results: baseline, CBLS, and AdFL. Each AdFL result can be broken down into 5 separate results for the different SBFL techniques.

3) Measurement: TBSM performs adaptations by modifying (in the current implementation of hddRASS, removing) program statements. The set of program statements that are considered by TBSM is the search space. For AdFL without a stopping criteria, the real search space is the same as for baseline, except with the statements prioritized so that more likely-removed statements are considered first. However, we can use the ranking of the worst-ranked modified statement to see *what stopping criteria could have been used* while preserving a "perfect" adaptation. It is important to note in what follows that we are comparing real search spaces for baseline and CBLS with a theoretically ideal search space for an AdFL with a perfect cutoff/stopping rule. This ideal search space will inform our effort to devise concrete methods for using AdFL to improve the efficiency of TBSM.

For *baseline*, the search space is the whole program. For *CBLS*, the search space is statements in the CBLS set. For AdFL's theoretical ideal search space, we use the *EXAM SCORE*, the most common measure used to compare FL techniques [34]. We apply AdFL to all the program statements and sort them by ranking. Then we look for the modified statement with the worst ranking. If the modified statement with the worst ranking is tied with other statements, we resolve the tie randomly. For AdFL, the "search space" is then all the statements up to the last statement that was modified. We have five distinct AdFL results based on the SBFL techniques: Tarantula, Ochiai, Barinel, Op2, and Dstar; the most widely studied SBFL formula [27]. The formula can be found in Pearson et al. [34]. For the remainder of the paper, % improvement of technique B over technique A is measured as %improvement = $((|B| - |A|)/|A|) * 100$. With a fixed cutoff rule (CBLS or a percent of highest ranked statements for AdFL) we measure % accuracy as % of statements modified out of statements modified by the baseline technique that searches the entire space of modifications.

V. Evaluation

We focused on three key research questions: **RQ1:** Is there a best formula for AdFL, among the five SBFL techniques evaluated? **RQ2:** How does AdFL compare to the baseline and CBLS for synthetic adaptations? **RQ3:** Most importantly, is AdFL an effective technique for real-world adaptations?

A. RQ1: Comparison of SBFL techniques for AdFL

To compare each of the five techniques for AdFL, we used the same comparison methodology used by Pearson et al. for evaluating and improving fault localization techniques

TABLE I: MEAN/MEDIAN are for EXAM SCORE. FLT=Fault Localization Technique. #Worse presents the number of techniques worse by tournament ranking. FLT Rank is mean across 800 data points.

FLT	MEAN	MEDIAN	#Worse	FLT Rank
Tarantula	0.490	0.521	0	1.966
Ochiai	0.511	0.540	0	2.095
op2	0.509	0.534	0	2.033
Barinel	0.505	0.541	0	2.053
DStar	0.513	0.537	0	2.13

TABLE II: MEAN/MEDIAN represents mean/median % improvement while choosing AdFL vs. baseline. AvgAdFL and WorstAdFL have 100% accuracy. G-AvgAdFL and G-WorstAdFL have mean accuracy of 79% (identical to CBLS).

Method	MEAN	MEDIAN	p-value	std dev	MIN	MAX
AvgAdFL	47.06	43.71	<0.005	29.31	0.34	97.01
WorstAdFL	41.40	37.70	<0.005	31.81	0.22	96.91
AvgG-AdFL	60.88	65.32	<0.005	23.85	4.93	97.28
WorstG-AdFL	42.71	41.30	<0.005	29.28	0.00	97.14

(FLTs) [34], with three evaluation metrics: (1) mean EXAM SCORE, (2) tournament ranking: pairwise comparison to determine if one FLT is statistically superior to another, and (3) mean FLT rank: using each data point to rank SBFL techniques from 1 to 5 and then averaging the rank across 800 data points

For tournament ranking, we used a *paired t-test*. Table I shows results. The #Worse columns shows that, for all 10 pairings, no technique was statistically significantly better than the other. MEAN, MEDIAN and FLT Rank values are also very close for all techniques. Hence, we can say that no clear winner was found among the five SBFL techniques. Therefore, in the following results, we report both average FLT result (AvgAdFL) and worst FLT result (WorstAdFL).

B. RQ2: AdFL vs. Baseline and CBLS

Table II compares AdFL, with and without the guard (pruning statements by CBLS) to baseline. If we used an ideal cutoff (stopping the search after the last modified statement), how many percent fewer statements would we have to examine if we used AdFL (or G-AdFL) instead of the full program? The results show that, with an ideal cutoff, (G-)AdFL performs considerably better than baseline, often removing nearly half the statements proposed from consideration. Guarded AdFL improves on baseline by a larger margin, at the cost of some accuracy compared to non-guarded AdFL. Guarded AdFL is, of course, exactly as accurate as CBLS, since CBLS provides the cutoff for AdFL.We also conducted paired t-tests, comparing AvgAdFL with CBLS and WorstAdFL with CBLS. For AvgAdFL vs. CBLS, the mean difference is 15.2 ($p < 0.005$) with 95% confidence interval (12.68, 17.77). Also, when comparing AvgAdFL with CBLS, the effect size is large (Cohen's $d > 0.5$). For WorstAdFL vs. CBLS, the mean difference is 9.48 ($p < 0.005$) with 95% confidence interval (6.87,12.26). From all the results, we can say that AdFL performs better than CBLS in search space reduction. Given that AdFL is better than CBLS, we also want to know the average improvement by choosing AdFL over CBLS. For that, we used AvgAdFL and compared it directly with CBLS. We define the % improvement $((|AvgAdFL| - |CBLS|)/|CBLS|) * 100$. where $|CBLS|$ is the size of the CBLS set and $|AvgAdFL|$ is the mean EXAM SCORE across all 5 techniques. A one-sample t-test on % improvement shows mean improvement of 23.20% with $p < 0.005$ and 95% confidence interval of (19.71, 26.69).

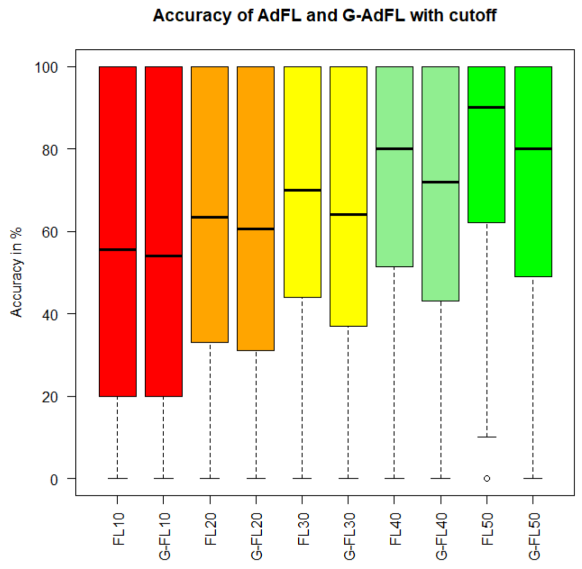

Fig. 2: % Accuracy of AvgAdFL and AvgG-AdFL with different cutoff criteria. FL10 and G-FL10 show 10% cutoff.

1) Using Real Cutoffs: Results up to this point rely on using an EXAM SCORE provided ideal cutoff that is, in practice, not possible for developers. However, the results also show that the ranking position of the last removed statement is such that we could, almost always, prune much more of the search space than just using CBLS. In practice, developers using an AdFL technique to prune the search space must pick some cutoff, and ignore statements below that suspicousness. Given a cutoff, note that the effectiveness of search space pruning is fixed — if a developer decides to only consider the top 10% of AdFL-ranked statements for removal, the reduction compared to baseline is obviously 90%. What can vary is the *accuracy* of the produced adaptation. Real cutoffs are highly context-specific, trading more time required for adaptation off for less likelyhood of not removing some statements. We therefore used 10%, 20%, 30%, 40% and 50% cutoff values, with 10% cutoff meaning that TBSM only considers the 10% top ranked statements by AdFL. Figure 2 shows how accuracy varies with these cutoffs for our 800 synthetic adaptations. In non-guarded AdFL, we directly apply the cutoff; in G-AdFl we may end up with a smaller search space than the cutoff, if CBLS itself prunes the space more than the cutoff would. Even with synthetic test suites with non-meaningful labels (the

TABLE III: Percent improvements over baseline. AdFL vs. CBLS shows the % improvement of average AdFL over CBLS.

Application	CBLS	Tarantula	Ochiai	Op2	Barinel	DStar	AvgAdFL	WorstAdFL	AdFL vs. CBLS
Elevation-API	55.17	91.72	93.79	90.34	91.72	92.41	92	90.34	66.75
NetBeans IDE undo-redo	90.71	92.28	93.42	96	93.85	94.42	94	92.28	3.62

equivalent of a fault with no reliable location), for non-guarded AdFL 45% of data points have 100% accuracy at a 50% cutoff.

C. *RQ3: AdFL in real-world adaptation scenarios*

How does AdFL perform in the real world? Table III shows EXAM SCORE derived ideal cutoffs for two actual adaptation scenarios with meaningful test labelings. For the industrial scenario from the TSAS system, CBLS is only able to prune 55% of the program statements, but using AdFL and a cutoff of 10% would produce a 100% accurate result, even using the very worst FLT. For the NetBeans IDE, CBLS performs quite well, but even the worst FLT still improves on it by a few percent. For the elevation-API adaptation, baseline TBSM without heuristic guidance requires 460 minutes. Using CBLS, this can be reduced to 118 minutes, and using AdFL with Op2 (the worst performing FLT) and a 10% cutoff only requires 49 minutes. For the NetBeans IDE adaptation, baseline TBSM without heuristic guidance requires 175 minutes. Using CBLS, this can be reduced to 61 minutes, and using AdFL with Op2 and a 10% cutoff only requires 57 minutes, slight improvement. Developers do not know in advance how well CBLS will perform, but in both scenarios here, they can safely use AdFL, which has no additional computation cost, and a 10% cutoff, without loss of accuracy.

VI. Threats to validity

We used open source Java projects and the NetBeans IDE in order to compare with previous work on TBSM. The only proprietary program used in our analysis is TSAS. For TSAS, tests were labeled by developers, again avoiding any bias on our part. We used the standard EXAM SCORE measure to study the effectiveness of SPFL techniques.

All the projects used in the synthetic study as well as case study scenarios are Java projects. In order to verify generality, we need an hddRASS-like tool that can modify/reduce programs written in other programming languages. The primary threat to Java generalization is that while we suspect our synthetic results are typical of similar artificial adaptation problems, the most meaningful data is our two realistic case studies. Our non-proprietary data and results are available online at https://github.com/amchristi/AdFL.

VII. Discussion: Best-Effort Incremental TBSM

The primary question raised by our experiments is how AdFLizer decides which statements are "unlikely" targets for modification: does the developer provide a cutoff? Our synthetic results suggest that developers *could* provide a 50% cutoff, and likely see no significant loss in accuracy. This would be useful, and is better than CBLS on synthetic adaptation problems, but is far from ideal. Our results on real-world adaptations suggest that a 10% cutoff might be safe for actual adaptation based on meaningful test labels, even if the developer picks a "bad" SBFL technique. However, we only have two real-world scenarios, so we hesitate to claim that 10% is really safe for most adaptations. What cutoff should we pick if only 20 minutes are available to achieve the resource adaptations? There is a way to sidestep the entire issue: provide AdFLizer directly with a time budget, rather than a fixed percent cutoff.

TABLE IV: Different TBSM strategies with corresponding time (in minutes) needed to build correct adaptations.

Application	baseline	CBLS	AdFL-10% cutoff	AdFL-inc
Elevation-API	460	118	49	35
NetBeans IDE undo-redo	175	61	57	20

The problem of determining a stopping criteria for TBSM is more acute than in APR: APR can stop whenever it has a version of a program that passes all tests. However, this distinction can be considered in a different light, to the advantage of TBSM. Namely, APR is useless until it produces a program that passes all tests, while TBSM is useful so long as it has removed some resource-using statements, and at least partly disabled problematic functionality, while preserving a useable system. If we assume that passing all unlabeled tests is usually an indicator a system is useable, even if not optimally adapted, then we can see that TBSM could be used as an incremental algorithm. In order to demonstrate the effectiveness of best-effort incremental TBSM, we used AdFL-driven TBSM to produce resource adaptations for both of the case study scenarios. We started with a computation budget of 5 minutes and continued to increment it by 5 minutes until all resource consuming statements were correctly modified. Table IV represents the time budget required to build a correctly adapted version for different TBSM strategies: baseline, CBLS, AdFL with low cutoff, and AdFL with fixed computation budget, approximating incremental TBSM. The time budgets required to compute correct adaptations for the Elevation API scenario and NetBeans IDE are 35 minutes and 20 minutes respectively, saving almost 30% and over 60% of the cost for even a low cutoff. Because incremental results always pass all retained tests, the approach is both safe and effective, even when the time budget available is very limited.

VIII. Conclusions and Future Work

Resource-adaptation is crucial for the survivability of modern, mission-critical software systems. TBSM is a technique for building resource adaptations that relies on *test labeling* and *program modification*, and has been applied by the developers of TSAS in real world adaptation problems. This paper presents a novel application of Fault Localization (FL) techniques to improve the efficiency of TBSM, inspired by the application of FL to Automated Program Repair (APR) and Spectrum-based Feature Comprehension. We show that using FL in real world scenarios would allow up to 90% of the search space to be pruned in TBSM. Furthermore, using FL in TBSM makes it possible to reconsider TBSM as an incremental best-effort adaptation method.

References

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

[2] 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.

[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] S. W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *Workshop on Software Engineering for Adaptive and Self-Managing Systems*, May 2009, pp. 132–141.

[6] Y. Al-Nashif, A. A. Kumar, S. Hariri, Y. Luo, F. Szidarovsky, and G. Qu, "Multi-level intrusion detection system (ML-IDS)," in *International Conference on Autonomic Computing*, 2008, pp. 131–140.

[7] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson, "QoS aspect languages and their runtime integration," in *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1998, pp. 303–318.

[8] J. Dowling and V. Cahill, "Self-managed decentralised systems using k-components and collaborative reinforcement learning," in *ACM SIGSOFT Workshop on Self-managed Systems*, 2004, pp. 39–43.

[9] 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.

[10] 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.

[11] 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.

[12] ——, "Cause reduction for quick testing," in *IEEE International Conference on Software Testing, Verification and Validation*, 2014, pp. 243–252.

[13] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: a generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[14] 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.

[15] 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.

[16] 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.

[17] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[18] R. De Lemos, H. Giese, H. A. Muller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezze, C. Prehofe, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, R. De Lemos, H. Giese, H. Müller, and M. Shaw, Eds. Springer, 2013, vol. 7475, pp. 1–26.

[19] A. Elkhodary, S. Malek, and N. Esfahani, "On the role of features in analyzing the architecture of self-adaptive software systems," in *4 th Workshop on Models@ run. time at MODELS 09*, 2009.

[20] A. Elkhodary, N. Esfahani, and S. Malek, "FUSION: a framework for engineering self-tuning self-adaptive software systems," in *Foundations of Software Engineering*, 2010, pp. 7–16.

[21] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng, "Towards runtime testing of dynamic adaptive systems," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2013, pp. 169–174.

[22] M. Luckey and G. Engels, "High-quality specification of self-adaptive software systems," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2013, pp. 143–152.

[23] F. Fleurey and A. Solberg, "A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems," in *International Conference on Model Driven Engineering Languages and Systems*, 2009, pp. 606–621.

[24] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "RELAX: incorporating uncertainty into the specification of self-adaptive systems," in *IEEE International Requirements Engineering Conference, RE*, 2009, pp. 79–88.

[25] P. Casanova, D. Garlan, B. R. Schmerl, and R. Abreu, "Diagnosing unobserved components in self-adaptive systems," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014, pp. 75–84.

[26] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.

[27] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[28] 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 *International Conference on Software Engineering*, 2012, pp. 3–13.

[29] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *International Conference on Automated Software Engineering*, 2013, pp. 356–366.

[30] Q. Xin, S. P. Reiss, and S. Krishnamurthi, "Program repair using code repositories," Brown University, Tech. Rep., 2016.

[31] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *International Conference on Software Engineering*, 2013, pp. 772–781.

[32] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *International Conference on Software Engineering*, May 2016, pp. 702–713.

[33] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *International Symposium on Software Testing and Analysis*, 2013, pp. 191–201.

[34] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *International Conference on Software Engineering*, 2017, pp. 609–620.