Accelerating Patch Validation for Program Repair with Interception-Based Execution Scheduling

Yuan-An Xiao, Chenyang Yang, Bo Wang and Yingfei Xiong

***Abstract*—Long patch validation time is a limiting factor for automated program repair (APR). Though the duality between patch validation and mutation testing is recognized, so far there exists no study of systematically adapting mutation testing techniques to general-purpose patch validation. To address this gap, we investigate existing mutation testing techniques and identify five classes of acceleration techniques that are suitable for general-purpose patch validation. Among them, mutant schemata and mutant deduplication have not been adapted to general-purpose patch validation due to the arbitrary changes that third-party APR approaches may introduce. This presents two problems for adaption: 1) the difficulty of implementing the static equivalence analysis required by the state-of-the-art mutant deduplication approach; 2) the difficulty of capturing the changes of patches to the system state at runtime.**

To overcome these problems, we propose two novel approaches: 1) execution scheduling, which detects the equivalence between patches online, avoiding the static equivalence analysis and its imprecision; 2) interception-based instrumentation, which intercepts the changes of patches to the system state, avoiding a full interpreter and its overhead.

Based on the contributions above, we implement ExpressAPR, a general-purpose patch validator for Java that integrates all recognized classes of techniques suitable for patch validation. Our large-scale evaluation with four APR approaches shows that ExpressAPR accelerates patch validation by 137.1x over plain validation or 8.8x over the state-of-the-art approach, making patch validation no longer the time bottleneck of APR. Patch validation time for a single bug can be reduced to within a few minutes on mainstream CPUs.

***Index Terms*—Automated program repair, patch validation**

I. Introduction

AUTOMATED program repair (APR) has attracted much attention in the recent decade. Many approaches [1, 2, 3, 4, 5, 6, 7, 8] have been proposed, and companies like Bloomberg [9], Meta [10], and Alibaba [11] are already using APR tools to fix software bugs in nightly builds.

This work was supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China under Grant No. 62202040 and 62161146003, and ZTE Industry-University-Institute Cooperation Funds under Grant No.HC-CN-20210319008.

Yuan-An Xiao and Yingfei Xiong are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing 100871, China (e-mail: xiaoyuanan@pku.edu.cn; xiongyf@pku.edu.cn). Yingfei Xiong is the corresponding author.

Chenyang Yang is with the School of Computer Science, Carnegie Mellon University (e-mail: cyang3@cs.cmu.edu). This work was finished mostly when Chenyang Yang was an undergraduate student at Peking University.

Bo Wang is with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China (e-mail: wangbo_cs@bjtu.edu.cn).

Though researchers have made significant progress in the effectiveness of APR, its efficiency has received relatively limited improvement. Efficiency decides the time needed for repairing a bug and is an important limiting factor for using APR in practice. The state-of-the-art APR approaches may still take hours to repair a bug, and recent program repair experiments still set the timeout to multiple hours [2, 8, 12]. The response time of current APR tools greatly exceeds users' patience, as reported by recent studies [12, 13], and significantly limits the application scenario of APR approaches.

The execution time of APR is dominated by patch validation [14, 15]. Most APR approaches are test-based and follow the generate-and-validate pattern [16]: they first generate a bunch of patches, and then validate each patch using the test suite. Executing the test suite can take minutes, and hundreds of patches can be generated for one fault [17]. To accelerate APR tools, we need to reduce the patch validation time.

Patch validation has been recognized as a dual of mutation testing [18]. Mutation testing applies a set of mutation operators to the program to produce mutants, and then executes the test suite on each mutant to calculate the mutation score. Mutation testing is a dual of patch validation as both need to obtain the test result for each mutant/patch. Given many techniques have been proposed to accelerate mutation testing, they could potentially be adapted to accelerate patch validation.

Under this duality, multiple existing attempts to accelerate patch validation can be seen as adapting the corresponding mutation testing technique [15, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. However, all of them adapt only one or a few techniques. It remains unclear which techniques can be adapted and what is the combined effect of these techniques on patch validation. Furthermore, many attempts are special-purpose, depending on a specific APR approach and not generalizing to different ones.

To fill this gap, we first systematically investigate the existing acceleration techniques for mutation testing and analyze their suitability for patch validation. We identify five classes of suitable techniques, namely, mutant schemata [30], mutant deduplication [31], test virtualization [32], test prioritization [33], and parallelization. The remaining techniques are either covered by these five classes or are unsuitable for the patch validation of current APR approaches. Among them, the latter three classes are already used by recent general-purpose patch validators [15, 22], but mutant schemata and mutant deduplication have never been adapted to a general-purpose patch validator, as far as we are aware. Mutant schemata weave all mutants into one program and avoid redundant compilation of different mutants; mutant deduplication detects mutants that

are equivalent to each other, and executes only one mutant among all equivalent mutants.

Migrating these two techniques to patch validation is not easy. In mutant testing, mutants are produced from pre-defined mutation operators, and their impact on the system is under control. However, in patch validation, the patches are produced from third-party APR approaches that arbitrarily change the program. This imposes two problems:

The first problem is that the state-of-the-art mutant deduplication approach [34] requires a static analysis to determine the equivalence of the rest of the test executions, when two patches deviate from the original program. The precision of this analysis is directly related to the effectiveness of the technique. It is difficult to implement a precise and scalable static equivalence analysis given the possible changes that the patches may cause to the system.

To address this problem, we propose a novel approach: **execution scheduling**. Execution scheduling does not require offline static equivalence analysis, and subsumes mutant schemata and mutant deduplication. The basic idea is to detect the equivalence between patches online during patch execution. Doing so removes the need for static equivalence analysis, thereby preventing the imprecision of static analysis from affecting the system's effectiveness. We also introduce a novel data structure, the *state-transition tree*, to record the dynamic analysis results and reuse them across the execution of different patches. Furthermore, to perform the detection, we need to weave all patches into a single program, subsuming mutant schemata. This approach is compatible with test prioritization, test virtualization, and parallelization, allowing us to integrate all suitable techniques into one system.

The second problem is that implementing mutant deduplication or execution scheduling requires a component to capture how a patch changes the system state to analyze equivalence and replay the change later. Existing mutant deduplication approaches usually implement an interpreter to execute the changed code. However, unlike mutation testing where only a selected set of operators may be changed and interpreted, a patch may arbitrarily change a statement that may involve method calls or even system calls. As a result, the interpreter must support all features in the host programming language and interact with the original runtime. This not only requires huge implementation effort but is also technically difficult as the original runtime may be commercial and prohibit modification. Furthermore, interpreting a large chunk of code inevitably incurs a significant overhead, which may nullify the effectiveness of the technique.

To support the implementation of execution scheduling, we propose a novel approach for **interception-based instrumentation**. It executes a patch without an interpreter by instrumenting code before and after the patch to record the possibly changed states and revert the changes. In this way, the patch is executed in the original runtime, avoiding the implementation cost and the runtime overhead of an interpreter.

However, realizing interception-based instrumentation is not easy. We highlight our approach in three aspects: 1) **A patch may bring control flow changes** via statements like `break` and `throw`. Capturing such changes is difficult because the instrumented statement may be skipped. We propose a design process based on the operational semantics of the programming language to reliably capture and replay the control flow change of a patch. 2) **The scope of data changes** for a patch is large, because a patch may call other methods and arbitrarily modify the memory at any location. The cost of recording all possible changes may therefore cancel the benefit of mutant deduplication. So, we analyze the possibly changed locations in a preparation step and apply equivalence detection only when the change scope is small enough. 3) **A patch may not compile**, and weaving all patches together may make the whole program uncompilable. We cannot simply detect an uncompilable patch by compiling it, because the benefit of mutant schemata will be lost. Instead, we propose the concept of *isolation unit* where compilation errors within one unit would not affect other units. We separate each patch into an isolation unit and identify all uncompilable patches at once.

Based on the contributions above, we implement ExpressAPR [35], a general-purpose patch validator for Java, which integrates all suitable techniques for patch validation. ExpressAPR provides support for Defects4J and Maven, and allows configuration for other Java projects. We then conduct a large-scale empirical evaluation that consumes over 17 months of CPU time to understand its performance on mainstream APR approaches. The evaluation leads to multiple findings:

> ExpressAPR achieves an acceleration of 137.1x over plain validation and outperforms the state-of-the-art approach by 8.8x. Patch validation is now faster than patch generation for the first time. The time to repair a bug can be reduced to a few minutes, meeting the expectation of most developers [12].
>
> All adapted techniques are effective for patch validation, each contributing to a significant acceleration.
>
> While ExpressAPR only works for patches under certain assumptions (Section VI), it supports over 97% of patches in our experiment, and it brings correct validation results for over 99.9% of patches. Therefore, ExpressAPR has a negligible impact on APR effectiveness.

In summary, the contributions of this paper are twofold:

Technically, we propose two novel approaches to realize mutation schemata and mutant deduplication, which have not been adapted to general-purpose patch validation before. The execution scheduling approach (Section IV) subsumes mutant schemata and mutant deduplication, avoiding the need for a static equivalence analysis that can be imprecise. The interception-based instrumentation approach (Section V) provides the capture-and-replay component required by execution scheduling, avoiding the need for a full-featured interpreter with runtime overhead.

Empirically, we investigate existing techniques for accelerating mutation testing, and identify five classes of techniques suitable for patch validation (Section III). We systematically integrate the complete set of techniques (Section VII) and conduct a large-scale experiment evaluating their performance (Section VIII), leading to novel findings that can serve as a solid base for future research (Section IX).

TABLE I: Average time usage per bug when evaluating Recoder

Phase	Step	Technique	Time
Patch Generation		Recoder	0:18:39
Patch Validation	Patch Compilation	Defects4J	2:36:04
		UniAPR	0:25:34
	Test Execution	Defects4J	19:53:48
		UniAPR	1:16:37

```
c = a + b;
```
(a) Program to be mutated

```
// mutant 1:
  c = a - b;
// mutant 2:
  c = a * b;
// mutant 3:
  c = a / b;
```
(b) Generated mutants

```
c = metaFunc(a, b);
int metaFunc(int x, int y) {
  switch(Env.getMutId()) {
    case 1: return x - y;
    case 2: return x * y;
    case 3: return x / y;
  }
}
```
(c) The mutant schemata

Fig. 1: Mutants and mutant schemata created by AOR

II. Motivation

Efficiency is a critical factor of APR. How long an APR tool takes to repair a bug determines the scenarios where it can be used. Noller et al. [12] recently found that half of programmers cannot wait longer than 30 minutes for an APR tool to produce a result. They further investigated how existing APR tools on the C programming language performed within one hour, revealing that even the best tool could correctly repair only two bugs in the ManyBugs benchmark. Such performance is significantly lower than the results reported in their original papers with a timeout of 10-24 hours.

Due to this unsatisfactory efficiency, it is commonly believed that the application of APR can only be applied to offline scenarios where users do not wait for real-time feedback (e.g., repairing bugs discovered in automatic nightly builds [12, 13]). If we can reduce the time of program repair to a few minutes, APR tools can assist users with real-time feedback, unlocking many more application scenarios (e.g., being integrated into IDEs and code editors).

To improve the efficiency of APR tools, we need to understand how current APR tools spend their time. This paper focuses on generate-and-validate APR, which is the subject of mainstream studies. These approaches have two main phases: in the *patch generation* phase, they loop through each suspicious location and generate a set of candidate patches at each location; in the *patch validation* phase, they execute the test suite to check the plausibility of each patch. The patch validation phase can be further divided into *patch compilation* and the *test execution* for compiled languages.

To understand how much time existing APR tools spend on different phases, we conducted a pilot study on the state-of-the-art tools. For patch generation, we selected Recoder [8], an effective APR approach based on deep learning. For patch validation, we selected UniAPR [22], a state-of-the-art patch validator. We took all bugs in the Defects4J 1.2 benchmarks that Recoder successfully repairs [8]. For each bug, we first executed the patch generation phase of Recoder with default parameters, and then used UniAPR to validate them. Since Defects4J provides a default validator (`defects4j compile && defects4j test`), which is used in many existing APR tools, we also executed the Defects4J validator for comparison.

We recorded the average time used in each step, as shown in Table I. We can see that the patch validation phase is the time bottleneck, accounting for most of the time (84.57% for UniAPR and 98.64% for Defects4J) and making the total repair time much longer than the 30-minute expectation. Therefore, to improve the efficiency of APR tools, we need to reduce the patch validation time. Furthermore, both steps in patch validation cost a significant amount of time even with the best technique. Therefore, to reduce the patch validation time, we need to reduce both the patch compilation time and the test execution time.

III. Investigating Mutation Testing Acceleration

We review existing mutation testing literature to systematically find acceleration techniques applicable to patch validation. We first collect mutation testing acceleration techniques by reading through the existing surveys [36, 37, 38], searching for publications matching the term "mutation testing" after 2019, after which the papers were not covered by the surveys, and studying the web page collecting existing mutation testing tools [39]. We then analyze the suitability of each technique for accelerating patch validation.

Our analysis identifies a set of techniques that are suitable to be adapted to patch validation. The remaining techniques are either unsuitable for validating the patch generated by current APR tools, or covered by the identified set of techniques. In this section, we present the classification result and briefly introduce each technique to make the paper self-contained.

A. Techniques Suitable for Patch Validation

1) Mutant Schemata: In mutation testing, each generated mutant needs to be compiled. Different mutants share most of their code, which is repetitively compiled. Mutant schemata [30] is a common technique to avoid repetitive compilation. Multiple mutants are encoded in a *meta-program* and then dynamically selected during runtime, so the shared code is compiled only once, reducing the redundant compilation.

Figure 1 illustrates an example where the AOR (replace arithmetic operator) mutation operator is applied to the expression `a+b`, as shown in (a). AOR generates multiple mutants by replacing the plus sign with other operators, as shown in (b). Normally each mutated program is independently compiled, so other parts of the code are compiled multiple times. But with mutant schemata, all mutants at this expression are grouped into a *meta-function* that dynamically selects a mutant based on a runtime flag, as shown in (c). In this way, all mutants are encoded in one program, and other parts of the code are compiled only once. Mutant schemata can be generated by source code (AST) [31] or byte-code [40] transformation.

2) Mutant Deduplication: Since a mutant is created by mutating only one or a few statements in the original program, a mutant may be equivalent to the original program or another mutant. In a group of equivalent mutants, only one of them needs to be executed, saving the test execution time. Two types of equivalencies have been considered in existing work.

The first one is *full equivalence*, where the two mutants will produce the same test result on any test. For example, the statements `x+=2;` and `x+=1+1;` are fully equivalent. Given a group of fully equivalent mutants, only one needs to be executed on all tests. The second one is *test-equivalence*, where the two mutants will produce the same test result on a specific test. Test-equivalence is much more common than full equivalence. For example, the statements `x+=2;` and `x*=2;` are not fully equivalent but are test-equivalent if `x` equals 2 before invoking the mutated statement in a test. Furthermore, any two mutants whose mutated statements are not executed by a test are equivalent with respect to the test. Given a group of test-equivalent mutants with respect to test t, only one needs to be executed on t.

To deduplicate mutants, existing approaches employ an offline procedure to detect equivalent mutants before the test execution, and then select one mutant among each equivalence class to be tested. Baldwin and Sayward [41] and Papadakis et al. [42] utilize compiler optimization to detect fully equivalent mutants, while Pan [43] uses constraint solvers to identify the fully equivalent mutants.

Since test-equivalence is more common than full equivalence, the Major mutation framework [34] identifies test-equivalent mutants by a pre-pass executing the original program and interpreting the mutated expressions in the mutants along the execution. Since the pre-pass is performed on the original program, Major can successfully detect test-equivalence between the original programs and the mutants, but for test-equivalence between mutants, Major requires a static analysis to determine the equivalence for the rest of the executions after they deviate from the original program. More discussion can be found in the next section.

3) Test Virtualization: Before executing a test for the mutants, the test suite must be initialized for each mutant. The repetitive initialization is costly in languages based on a virtual machine (VM), such as Java, as booting the VM takes non-trivial time. Test virtualization approaches such as VMVM [32] reduce this cost by reusing the previous VM instance for the next mutant execution. The global variables changed by the previous test execution are identified and reset before the next round of test execution by instrumentation.

4) Test Case Prioritization: The idea of test case prioritization is that some test cases run faster or are more likely to fail, so that if these test cases run before other test cases, the mutant will be killed earlier when they fail [33, 44]. Different heuristic policies can be used to determine what tests get prioritized. In particular, in *regression testing*, of which APR patch validation can be seen as an instance, a typical heuristic is to prioritize the previously failed test cases. Another heuristic is to prioritize the test cases in the same package of the modified code, which are most likely to cover the modified code.

5) Parallelization: Many mutation testing tools [39, 45] seek parallel test execution on multicore processors. Because tests and mutants are independent of each other, parallelization can be trivially applied by dividing the work into multiple pieces to be consumed by a process pool, e.g., each piece dealing with a small group of mutants or tests. Some early-stage mutation testing approaches also utilize hardware parallelization mechanisms, such as SIMD [46] and MIMD [47].

B. Techniques Unsuitable for Current APR

A class of acceleration techniques uses an optimized execution engine to execute all mutants at the same time, reducing possible redundant execution when executing each mutant separately. We classify these techniques as unsuitable for general-purpose patch validation for current APR approaches, because they either require a specific platform feature, or incur too much overhead such that the benefit gained is difficult to surpass the overhead:

Fork-based mutation analysis tries to accelerate by sharing the same execution among mutants, relying on the fork mechanism of the POSIX systems. Spilt-stream execution (SSE) begins with one process representing all mutants, and then forks into multiple subprocesses when a mutated location in the program is reached, one subprocess representing one mutated statement [48, 49]. AccMut [50] and WinMut [51] are two enhanced versions of SSE. When a mutated location is reached, their engines analyze all mutants for their changes to the system state, and cluster the mutants based on their changes: the mutants whose changes lead to the same system state are in the same cluster. Finally, the engines fork only one process for each cluster, reducing the number of processes compared with SSE. However, all of the fork-based approaches require the fork mechanism, and are not suitable to be used in a general-purpose patch validation approach as we need to support different operating systems and programming languages that may not support the fork mechanism, such as Windows and Java.

Variational execution [52] employs a special execution engine allowing a variable to store a conditional value, which is a special data structure that records possible values of the variable in all mutants. Operations in the program, such as plus or minus, directly manipulate conditional values, and thus executing the program with conditional values produces the test result of all mutants. However, executing the statements over conditional values incurs significant overhead, and thus variational execution is effective only when the number of mutants is huge (e.g., exponential). As reported in an existing study [53], variational execution could slow down the execution when there are only tens of or hundreds of mutants, a scenario commonly encountered in patch validation.

Furthermore, all techniques discussed above are *lossless* acceleration techniques. There are also *lossy* acceleration techniques [36, 54, 55, 56, 57, 58, 59, 60], which approximate the test results on the mutants to produce the mutation score, which is defined as the ratio of mutants that cause a test failure. However, on patch validation, we need to get the accurate test result on each patch to identify a correct patch, rather than calculating a ratio, and thus the lossy techniques do not apply to the patch validation.

C. Techniques That Are Subsumed

Test case selection is an optimization used in several mutation testing tools [39, 61, 62] that skips test cases that

Fig. 2: State transitions with different patch validation approaches

Fig. 3: Iteratively building the state-transition tree in ExpressAPR

do not execute the mutated code. It is subsumed by mutant deduplication because if the test does not execute the mutated code, all mutants should be in a test-equivalence class.

IV. Execution Scheduling

A. Overview

Among all the five classes of suitable acceleration techniques, mutant deduplication and mutant schemata have not been adapted to general-purpose patch validation due to the differences between mutation testing and patch validation. In this section, we propose the execution scheduling approach, which is suitable for general-purpose patch validation and subsumes both acceleration techniques.

We first demonstrate the possible redundancies during patch validation with an example as shown in the code snippet below. In this code snippet are five patches, P_1 to P_5, each modifying either of the two statements in the function `f()` (S_1 for the former two patches, S_2 for the latter three patches). The test function `test()` determines the correctness of `f()` by asserting its behavior.

```
int i=2, j=1;
void f() {
    i+=2;   // S1
  // P1: "i  *=2;"  P2: "i=j+3;"
    j+=2;   // S2
  // P3: "j  *=2;"  P4: "j=i-2;"  P5: "j=2;"
}
void test() {
    f();    assert (i==4 && j==2);
    f();    assert (i==6 && j==4);
}
```

Figure 2(a) illustrates the redundant test executions in this example, where each vertical sequence shows the execution of the test on one patch, and each circle indicates the system state at a specific location. P_1 and P_2 are test-equivalent to the original program, because all of them effectively set `i=4` and `j=3` at the first call to `f()` and thus the first assertion in `test()` fails (State C). P_3 and P_4 are test-equivalent to each other, setting `i=4,j=2` at the first call to `f()` (State D), and `i=6,j=4` at the second call (State F), passing both assertions. P_5 is not test-euivalent to any other variant: though it sets `i=4,j=2` at the first call to `f()`, temporarily leading to the same state D as P_3 and P_4, their states deviate during the second call where P_5 sets `j=2` instead of 4 (State G). Therefore, if we take the plain patch validation approach, where the test iteratively executes over P_1 to P_5, test-equivalent patches always follow the same path of state transition, leading to redundancies as highlighted in yellow.

As mentioned, the state-of-the-art mutant deduplication approach, Major, detects test-equivalence by a pre-pass on the original program, and executes the test on only one patch from each equivalence class. In this example, Major will instrument the original program and execute the instrumented program as a pre-pass. When a location modified by patches is reached, e.g., S1, the instrumented code will invoke an interpreter to interpret all the patches based on the system state of the original program, record and compare their changes to the system state. If a patch always makes the same changes to the system state as the original program, it is test-equivalent to the original program, and thus does not need to test. In this example, it detects that P_1 and P_2 are test-equivalent to the original program. However, it is hard to identify whether P_3, P_4, and P_5 are test-equivalent to each other. Although they make the same change to the system state in their first invocation (state B), their behavior in the second invocation (state E) is unknown, because state E deviates from the pre-pass which follows the original program ($A \rightarrow B \rightarrow C$). Therefore, Major requires a static analysis to determine the equivalence between the patches whose execution deviates from the original program. How to implement such a static analysis is not discussed in the original publication, and implementing a precise and scalable static analysis of test-equivalence between patches is difficult. In this case, if we cannot statically determine the test-equivalence between P_3 and P_4, we have to execute the tests on both of them, leading to Figure 2(b).

Our execution scheduling approach overcomes this limitation by embedding the detection process into the test execution process instead of in a pre-pass. To achieve that, we record the runtime behavior of patches as a state-transition tree, as shown in Figure 3(a). Each node in the tree indicates an interesting program state of the test, where either the next statement to execute is changed by some patches, or the test completes. In the figure, a node annotated with $S_n(v_i; v_j)$ means that the statement S_n will be executed with variables `i` and `j` set to v_i and v_j, and a node annotated with "Failed" or "Passed" means the result of a completed test. Each edge $\xrightarrow{P}$ V indicates the transition of interesting states: when executing the variant P against the state U, the next interesting state will be V. For each patch, its execution corresponds to a path from the root

node to a leaf node.

The state-transition tree is built iteratively through multiple rounds of test execution. The current analysis result is recorded on the state-transition tree, so that in the next round we can test another patch that is not test-equivalent to any existing patches. As for the above example, P_1 to P_5 are validated with three rounds of test execution, as shown in Figure 3(b). In the first round, we randomly select a patch, P_1, and throughout its execution, we record state changes of other patches onto the state-transition tree. We observe that P_2 to P_5 deviate from P_1 (with the state-transition path $A \to B \to C$) at B, as marked by the ? symbol. Therefore, in the second round, we select a patch among P_3 to P_5 to explore the deviated $A \to B \to ?$ path. This process repeats until all paths are explored. In this way, we can achieve ideal mutant deduplication for this example, as illustrated in Figure 2(c), without the need for heavy static analysis in a pre-pass.

B. Problem Definition

Before introducing the approach, we define a set of concepts related to patch validation. These definitions abstract away details in different programming languages, such that our algorithm can apply to a wide range of programs that follow this definition.

We view the procedure of executing a test case as stepping through a state machine, as shown as Algorithm 1. The codebase consists of a set of locations and a mapping $Cb.Stmt$ from locations to statements, which represents all source code in the project including tests. When a test begins, the location of the initial state S points to the entry point statement of the test case ($Cb.TestEntry$). Then the statement gets executed, modifying S that includes the current location ($S.Loc$). This process repeats until the state becomes a termination state that represents a failed or passed test result. Please note this algorithm is conceptual and does not imply an interpreter. In compiler-based language, the algorithm is implemented by the hardware architecture and the language runtime.

Algorithm 1 Test execution

```
   Input: The codebase Cb
1: S ← {Loc: Cb.TestEntry}
2: while S is not a termination state do
3:     S ← EXECUTE(S, Cb.Stmt[S.Loc])
4: end while
5: REPORTRESULT(S)
```

A patch is defined as a modification to the codebase, which is a modified mapping that replaces one or a few statements in the codebase. Given a set of patches Ps, a plain patch validation procedure enumerates through the patch set to execute the test against each modified mapping of statements, as shown as Algorithm 2. If the codebase contains multiple test cases, this procedure is repeated for each test against patches that survive all previous tests.

Algorithm 2 Plain patch validation

```
   Input: The codebase Cb, the patch set Ps
1: for P ∈ Ps do
2:     S ← {Loc: Cb.TestEntry}
3:     while S is not a termination state do
4:         S ← EXECUTE(S, P[S.Loc])
5:     end while
6:     REPORTRESULT(P, S)
7: end for
```

C. The Execution Scheduler

In this subsection, we explain how execution scheduling works. Same as in the previous subsection, we shall only give an algorithm describing the process. How to implement it using instrumentation will be explained in Section V.

Our approach requires a component to capture the state changes of the patches for analysis. Concretely, we require that there exist a CAPTURE procedure and a REPLAY procedure, so that $C \leftarrow \text{CAPTURE}(S, T)$ extracts the state change of statement T over the state S into C, and $S' \leftarrow \text{REPLAY}(S, C)$ actually applies the state change C to the state S. In this way, we can check whether the state change of two patches is the same by comparing the value of C. We require that these two procedures are accurate, such that $\text{REPLAY}(S, \text{CAPTURE}(S, T)) = \text{EXECUTE}(S, T)$ for any state S and any statement T. Capturing and replaying changes will be discussed in Section V, and for now, we keep CAPTURE and REPLAY as abstract procedures in this section.

The execution scheduler is shown as Algorithm 3. It takes as input the codebase of the project and a patch set to be validated. It maintains a variable $Root$, which corresponds to the root of the state-transition tree (Figure 3). Each node in the tree is labeled with a status ("visited", "not-visited", or "test-finished"), a set of patches that belong to this node, and a mapping "Edges" from state changes to the child nodes. Initially at line 1, the tree has only a root node labeled as "not-visited" with all patches belonging to it, indicating that nothing has been explored yet. Then, the loop body starting at line 2 explores a path from the root node to a leaf node. The loop body resembles the plain patch validation, except that when there are some patches to the current statement ($Patches(S) \neq \emptyset$, at line 7), the EVALPATCHES procedure analyzes the state change of each patch, and chooses a child corresponding to a group of patches making the same state change to move forward. When a round of test execution finishes, test results for all patches belonging to the current node are reported (line 14).

EVALPATCHES is the critical procedure of the execution scheduling. When the current node is visited for the first time (line 18), it captures the state change of each patch, and inserts a child node under the current node for each unique state change. For each patch, it searches for an existing edge corresponding to the state change of the patch (line 21): if there is such an edge, the patch merges into the patch set of the sub-node; otherwise, the patch forms a new sub-node on its own. Finally, EVALPATCHES finds a child whose subtree includes at least one "not-visited" node to continue execution (line 28), updating the current node and system state (line 29).

Algorithm 3 Patch validation with execution scheduling

```
    Input: The codebase Cb, the patch set Ps
 1: Root ← { Status: "not-visited", Patches: Ps, Edges: ? }
 2: while there are "not-visited" nodes under Root do
 3:   Cur ← Root
 4:   S ← { Loc: Cb.TestEntry }
 5:   while S is not a termination state do
 6:     Pc ← { P ∈ Cur.Patches |
              P[S.Loc] ≠ Cb.Stmt[S.Loc] }
 7:     if Pc ≠ ∅ then
 8:       Cur, S ← EVAL PATCHES(Cur, S, Cur.Patches)
 9:     else
10:       S ← EXECUTE(S, Cb.Stmt[S.Loc])
11:     end if
12:   end while
13:   Cur.Status ← "test-finished"
14:   REPORT RESULTS(Cur.Patches, S)
15: end while

16: procedure EVAL PATCHES(Cur, S, Ps)
17:   if Cur.Status = "not-visited" then
18:     Cur.Status ← "visited"
19:     for P ∈ Ps do
20:       Ch ← CAPTURE(S, P[S.Loc])
21:       if S.Edges[Ch] is defined then
22:         S.Edges[Ch].Patches
              ← S.Edges[Ch].Patches ∪ { P }
23:       else
24:         S.Edges[Ch] ← {
              Status: "not-visited", Patches: { P }, Edges: ? }
25:       end if
26:     end for
27:   end if
28:   Ch, Cur ← FIND NOT VISITED CHILD(Cur.Edges)
29:   return Cur, REPLAY(S, Ch)
30: end procedure
```

D. Properties of the Execution Scheduler

Theorem (Efficiency). The number of rounds of test executions in Algorithm 3 is equal to the number of test-equivalence classes among patches, i.e., it removes all redundancies caused by test-equivalence.

Proof Sketch. Because two patches are test-equivalent if and only if they always make the same state change during the test execution, test-equivalent patches are never separated by EVAL PATCHES into different sub-nodes, and non-test-equivalent patches must be separated by EVAL PATCHES when they cause different state changes. Therefore, each leaf node labeled with "test-finished" corresponds to a test-equivalence class. We can see from Algorithm 3 that each round in the loop turns one label of a leaf node from "not-visited" to "test-finished" at line 13, so the number of rounds is equal to the number of "test-finished" leaf nodes, which is furtherly equal to the number of test-equivalence classes. □

Theorem (Soundness). Algorithm 3 reports the same patch validation result as Algorithm 2.

Proof Sketch. First, we generalize Algorithm 3 so that the initial S and $Root$ are read from inputs. Then we can prove that Algorithm 3 is equivalent to Algorithm 2 by induction on the depth of $Root$. For the base case when the depth is one (so $Root$ has no children, and thus EVAL PATCHES is never executed), two algorithms are apparently equivalent because both algorithms execute the original program, leading to the same test result for all patches. For the induction case, consider an arbitrary patch P that belongs to a leaf node L, which is reached by the path $Tree \to A \to \cdots \to L$. The test result for P must be reported in the round that explores this path. During this round, let us consider the moment when Cur changes from $Root$ to A, which is the moment when EVAL PATCHES is called (line 8) for the first time. Before this moment, both algorithms always execute the original program, so S will be the same for the two algorithms up to this moment. Then, S is updated to $S' :=$ EXECUTE$(S, Stmt)$ in Algorithm 2 and $S'' :=$ REPLAY$(S,$ CAPTURE$(S, Stmt))$ in Algorithm 3, where $Stmt := P[S.Loc]$. Based on the accuracy requirement for CAPTURE and REPLAY, we must have $S' = S''$. Then we can apply the induction hypothesis with the initial S set to S' and $Root$ set to A. □

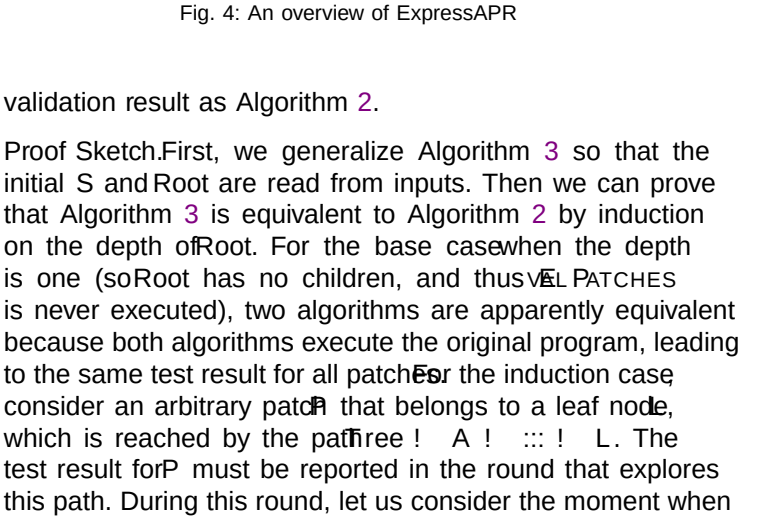

Fig. 4: An overview of ExpressAPR

V. INTERCEPTION-BASED INSTRUMENTATION

The execution scheduling algorithm can be implemented in different ways. To ensure the performance of the system, we implement this algorithm using instrumentation. That is, the tests and the patches are executed in the original runtime of the programming language, and only at certain locations, the instrumented code is invoked to detect equivalent patches and schedule executions.

An overview of the patch validation process is shown in Figure 4. It takes three steps to validate all patches: preparation, patch compilation, and test execution. In the first step, the codebase as well as the patches are instrumented to ensure the correct execution of the execution scheduler. In the second step, we compile the instrumented codebase, dropping patches that are uncompilable. In the third step, the execution scheduler iteratively runs the test suite, and stores the test result for each patch.

Implementing the overall process of Algorithm 3 with instrumentation is straightforward. First, a main procedure is injected into the codebase that initializes the state-transition tree and starts test executions. Then, at each patched location,

the code is instrumented to evaluate the patches and update the tree. The main challenging part is how to implement the CAPTURE and REPLAY in Algorithm 3.

Existing mutation-testing approaches [31, 52] instrument an interpreter at the mutated location for this purpose. However, in APR, since a patch may invoke other methods and then execute a large chunk of code, using an interpreter is not only expensive but also difficult to implement. To solve this problem, we propose interception-based instrumentation. Our approach inserts code before and after the patched statement, such that the patch itself executes normally in the original runtime, and the inserted code is responsible for detecting what changes the patch has made to the system state and reverting the changes, i.e., the change is intercepted.

In the preparation step, our approach adds instrumentation code around each patch, turning each patch into a *capture* component and a *replay* component. All such components are woven together into the codebase, effectively implementing mutant schemata. Finally, in the test execution step, the execution scheduler calls the corresponding component as the CAPTURE and REPLAY procedures.

In the rest of this section, we shall first introduce a small programming language, IMP+, for illustration. Then we describe how to intercept the changes based on IMP+. Finally, we show how to deal with uncompilable patches.

A. IMP+

IMP+ is an enhancement of the classic IMP language [63] with control flow constructs. IMP+ contains the commonalities between imperative programming languages, so we use it to illustrate our instrumentation process. The syntax of IMP+ is shown below:

```
s ::= s1 s2                          (statements)
    | x := e;
    | x := m(e1, ..., en);
    | if (e) s1 else s2
    | try s1 catch (x) s2
    | while (e) s
    | break;
    | continue;
    | return e;
    | throw e;
    | ...

e ::= e1 + e2                        (expressions)
    | ...
```

The runtime system state of an IMP+ program is a pair $\langle \sigma, \ell \rangle$, where the *data state* σ is a function mapping variables to their values, and the *control state* ℓ can be one of the following values that reflects the effect of control flow statements:

- `Normal`, indicating that we should normally execute the next statement;
- `Break`, indicating that a loop should break;
- `Continue`, indicating that a loop should skip to the end of its body;
- `Return` v, indicating that a method has returned v;
- `Exception` e, indicating that an exception e has been thrown.

The main operational semantic rules of IMP+ are shown in Figure 5, where $\sigma \xrightarrow{e} v$ means that expression e evaluates to v under the system state σ, `Normal`, and $\langle \sigma, s, \ell \rangle \Rightarrow \langle \sigma', \ell' \rangle$ means that statement s changes the system state from $\langle \sigma, \text{Normal} \rangle$ to $\langle \sigma', \ell' \rangle$. For simplicity, we only show the rules related to control flow change, and omit standard rules such as those for assignments and conditionals.

Since a state includes two parts, we need to intercept the changes to both parts. Below, we discuss how to deal with changes to the control state and the data state respectively.

B. Changes to the Control State

Recall that the *capture* component is responsible for detecting what changes a patched statement has made, and reverts the change; the *replay* component is responsible for replaying the recorded change. For changes to both states, the main challenge for implementing the two components lies in the control state, where the statement following the patched statement in the code file is not always the next statement to be executed at runtime. Therefore, we need a reliable way to detect and revert control state changes after the execution of the patch. Furthermore, since different programming languages have different language constructs for changing the control states, the design of these two components is by nature language-dependent.

To cope with the challenges, we propose a design process, Process 1, to systematically design *capture* and *replay* components for the control state given the semantic rules of the target programming languages. While this process involves human effort, it only needs to be done once when migrating ExpressAPR to a new language: once these components are implemented, they are fully automatic and can be used for all patches under this language.

Process 1 Capturing/replaying control state changes

```
Input: Semantic rules S
Output: Capture component C, replay component R
C := s; s
R := skip
T := possible types of ℓ in S
while (T ∩ {Normal} ≠ ∅) do
    Choose a t ∈ (T ∩ {Normal})
    T := T ∩ {t}
    #1: Pick an R_c ∈ S that can change ℓ from t to Normal, and an R_r ∈ S that can change ℓ from Normal to t.
    #2: Use R_c to write a capture component c_t(s) for any target statement s and the type t.
    #3: Refine c_t(s) to ensure that it preserves the semantics of s when it yields a state in T.
    #4: Use R_r to write a replay component r_t for type t.
    C := s; c_t(C(s))
    R := r_t; R
end while
```

The generated *capture* component takes a patch as input, and produces a code snippet for executing the patch and

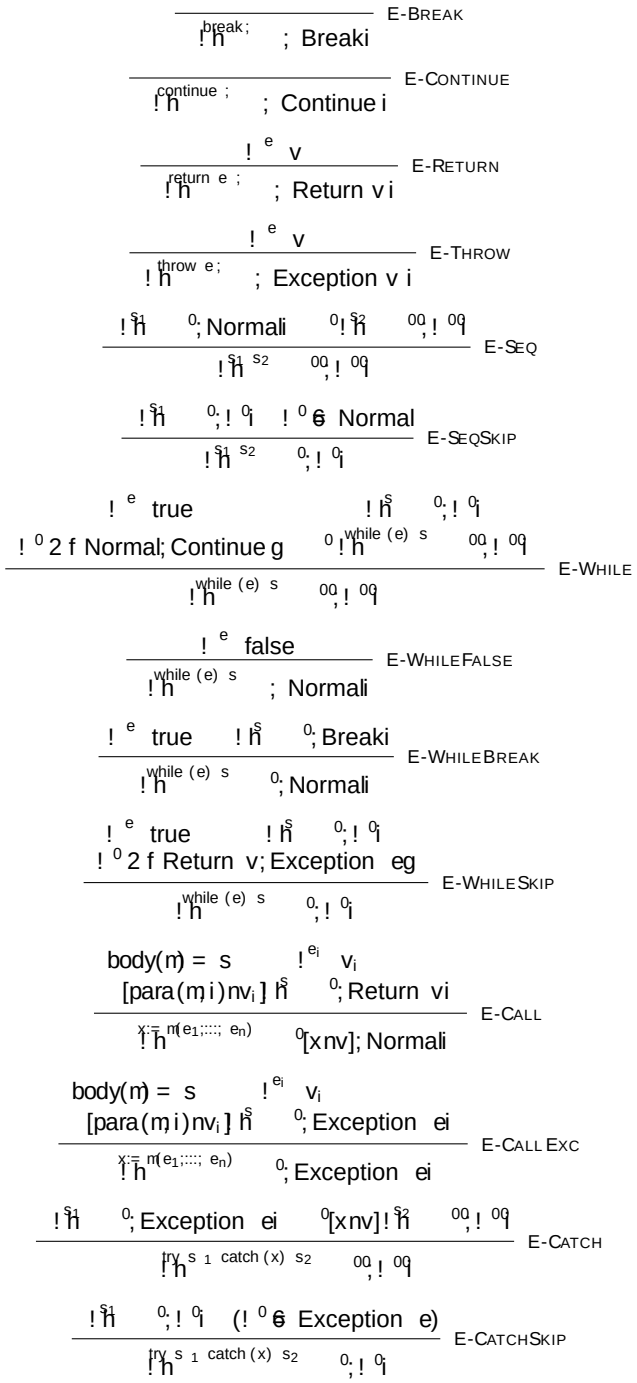

Fig. 5: Semantic rules of IMP+

capturing the change in variable change . The generated replay is a code snippet replaying the change captured in change . Process 1 works by capturing and reverting each type of abnormal control state at a time. For each type, there are four manual steps (#1 to #4) to craft a capture component and a replay component for this exact type of control state. Then, these components are stacked together (line 11-12) to cope with the whole control state.

We illustrate this process with the semantic rules of IMP+ as shown in Figure 5, where Break , Continue , Return v and Throw e are abnormal control states that we concern.

For the Break state: At step #1, we find that the only possible R_c is E-WHILEBREAK and the only possible R_r is E-BREAK. At step #2, we use this rule to write a capture component that turns a Break state into Normal :

```
while(true) {}
change := "break";
```

Among possible states in T = {Normal , Continue , Return v, Exception e}, the component above does not preserve the semantics of the former two states: if the patch ends up with a Normal or Continue state, it causes an endless loop. Therefore at step #3, We modify the capture component to fix these problems:

```
flag := 0;
while(true) {
  flag := flag+1;
  if(flag>1) break;
  {}
  flag := flag+1; }
if(flag=1) change := "break";
if(flag=2) continue;
```

The added flag variable will distinguish Break from Normal and Continue , so that it correctly preserves the semantics of s when it does not cause a Break state. Finally at step #4, we use E-BREAK to write a replay component:

```
if(change="break") break;
```

For the Continue state: At step #1, we find R_c = E-WHILE and R_r = E-CONTINUE. At step #2–#3, the capture component is mostly the same as the component for Break , but the last line should be changed to if(flag=2) change:="continue"; . At step #4, the replay component is if(change="continue") continue; .

For the Return v state: At step #1, R_c = E-CALL and R_r = E-RETURN. At step #2, the capture component wraps the statement into a new method and then calls it:

```
def m() {}
x := m();
change := "return"; retval := x;
```

For each state in T = {Normal , Exception e}, this component fails when the patch ends up with a Normal state. At step #3, we fix it by return ing a unique special value at the end of the wrapping method:

```
def m() { {} return "--special--"; }
x := m();
if(x!="--special--") {
  change := "return"; retval := x; }
```

At step #4, we come up with this replay component:

```
if(change="return") return retval;
```

For the Exception e state, we similarly use R_c = E-CATCH to write the capture component as try {} catch(e) { change:="exc"; exc:=e; }, and uses R_r = E-THROW to write the replay component as if (change="exc") throw exc; .

In the end, we get the following components that capture and replay all control state changes.

$$C(s) = c_{Exception}(c_{Return}(c_{Continue}(c_{Break}(s))))$$
$$R = r_{Exception}; r_{Return}; r_{Continue}; r_{Break}; skip$$

C. Changes to the Data State

After the control state is intercepted in Process 1, further intercepting changes to the data state is easier because we can detect and revert the modified variables in the next statement following the patch – the next statement is now guaranteed to run. On the other hand, unlike the control state where reverting always sets it to `Normal`, here we need an efficient way to detect and revert the data state to its version before executing the patched statement. Simply recording the whole data state is not feasible due to its size.

To cope with this challenge, we use lightweight static analysis to detect the change scope that each patch may bring to the data state, and record only the values within the scope. After executing the patched statement, the new values in the scope can be compared with the recorded ones for detecting and reverting the change. We further bypass the equivalence detection on patches whose change scope is too large: these patches are validated sequentially using the plain method but not with the execution scheduler.

Our approach does not restrict the choice of the static analysis and the definition of the change scope, and the implementation could use any static analysis algorithm that best fits the target programming language and the APR tools. In the following, we describe an implementation of the two components for the IMP+ language, which is close to those used in ExpressAPR.

In IMP+, the only two types of statement that can change the data state are $x := e$ and $x := m(e_1, \ldots, e_n)$, which change the value of a variable x to an expression or the return value of a method. As a result, we can define the change scope as a set of variables, and use an inter-procedural static analysis to produce this change scope. This static analysis can be easily implemented by scanning the code and collecting the left variables in all assignments. When a method call is encountered, all variables changed by the target method are also iteratively added until a fixed point is reached.

Given a set of variables $\{x_1, \ldots x_n\}$ returned by the static analysis, we can refine the `capture` component in the following way to intercept data state changes.

```
old_1 := x_1; ...; old_n := x_n;
(original capture component by Process 1)
if (old_1 != x_1) {change[0] := x_1; x_1 := old_1;}
else {change[0] := null;}
...
if (old_n != x_n) {change[n-1] := x_n; x_n := old_n;}
else {change[n-1] := null;}
```

We similarly refine the `replay` component.

```
if (change[0] != null) x_1 := change[0];
...
if (change[n-1] != null) x_n := change[n-1];
(original replay component by Process 1)
```

This implementation still has two issues in efficiency. First, the change scope may include too many variables, causing excessive overhead at runtime. To deal with this issue, we simply use a purity analysis [64] rather than an inter-procedural change scope analysis. A method is pure if it does not change the data state. A purity analysis involves only a Boolean result and can be efficiently conducted. If a patched statement calls any impure method, we consider its change scope too large and it will bypass the equivalence detection.

Second, a variable may store a large data structure where copying and comparison may be costly. To deal with this issue, we keep an allowlist of types that are either primitive types or compound types of a small, fixed size. If the change scope includes any variable whose type is not in the allowlist, we consider the change scope too large and the patch will bypass the equivalence detection.

D. Properties of Interception-based Instrumentation

Theorem (Efficiency). *The time cost of the designed `capture` and `replay` components in the above example is at most linear to the syntactic size of the patched statement, i.e., the overhead of interception-based instrumentation is small even if the patched statement may execute for a long time (e.g., contains a loop or a recursive call).*

Proof Sketch. The final `capture` and `replay` components consist of statements that deal with changes to the control state and the data state. For the control state, a constant number of statements are added to the component. For the data state, the number of added statements is linear to the number of possibly modified variables, which is no more than the syntactic size of the statement. □

Theorem (Soundness). $\textsc{Replay}(S; \textsc{Capture}(S; T)) = \textsc{Execute}(S; T)$ *for any state S and any statement T, i.e., calling the `capture` component and then the `replay` component for any patch is equivalent to executing the patch.*

Proof Sketch. The construction of both components is incremental, i.e., we build a component for each aspect of the state change, and combine them together. Each individual component is sound within its own scope by design, and the combining process is also sound: when combining components for control state changes, step #3 in Process 1 ensures that an added component does not break previous components; when refining the component for data state changes, the control state is already reverted, so that the refined component always executes normally regardless of control flow effects in the patch. □

E. Compilation Isolation

Unlike mutation testing that assumes mutants always compile, patches generated by an APR approach may not compile, leading to a challenge when all patches are woven into one program: if any patch cannot compile, the whole program cannot compile. It is impossible to detect and remove uncompilable patches by individually compiling each patch, because it will nullify the acceleration effect of mutant schemata.

An intuition to solve this challenge is to rely on the error messages from the compiler: if the error messages could precisely pinpoint all uncompilable code snippets, we can compile the codebase once and then remove the uncompilable patches based on the error messages.

However, the error messages are often not perfectly precise: for example, a patch in the middle of a method may lead to compile errors at the end of this method, and it is often the case that an error stops the compilation of the rest of the method. Therefore, we cannot rely on the error messages to precisely pinpoint all uncompilable patches.

To solve this problem, we propose the concept of *isolation unit* to measure the granularity of the preciseness in the error messages of a compiler: a compilation error within an isolation unit will only cause error messages within the isolation unit, but not affect the compilation of other parts of the code. An isolation unit must exist for any compiler: in the extreme case, the whole codebase is the isolation unit. However, our observation is that modern compilers usually have more fine-grained isolation units. For example, many compilers compile each file individually into an object file (.java files into Java bytecode files, and .c files into assembly files), and thus each file is an isolation unit. Furthermore, a method or a procedure in many imperative programming languages, including Java and C, is also an isolation unit, because these programming languages are carefully designed such that compilers only need to perform intra-procedural analysis when compiling the code (inter-procedural analysis is often applied at a much later stage for optimizing the code where the compilation errors have already been detected).

Therefore, to isolate compile errors, our interception-based instrumentation approach should wrap each patch in an isolation unit. Let us assume that methods are also isolation units for an IMP+ compiler. Each `capture` component produced in Section V-B is already in an individual isolation unit because we wrap the patch in a method to deal with the `return v` control state. Then we can compile the codebase woven with all patches with two rounds of compilation: the first round identifies all uncompilable patches from error messages, which will be removed, and the second round compiles the codebase with only the remaining patches.

VI. Limitation

While our approach is theoretically sound and can be applied to arbitrary patches under our idealized problem definition in Section IV-B, there are two practical limitations when applying it in the real world. We will evaluate them in our experiment.

Patch Limitation: We define a patch as a modification of statements. Therefore, patches that modify other parts in the program (e.g., the definition of a field) are out of the scope of ExpressAPR. We detect this case in the preparation step, and fall back to plain validation for affected patches.

Test Limitation: We define executing the test as stepping through a state machine, where the state-transition should be stable, i.e., being exactly the same across multiple runs. In practice, most test cases are stable – otherwise, it would be difficult to troubleshoot a test failure. We perform runtime checks in Algorithm 3 to detect rare unstable test cases[1], and fall back to plain validation for them.

Also note that ExpressAPR is only suitable for test-based generate-and-validate APR. While most existing studies focus on this category, many other kinds of defects, such as alerts by static analyzers [65] or anomalies in logs [66, 67], may also be targets of APR. ExpressAPR is not designed for these approaches because they do not run tests for patch validation.

VII. Implementation

After the challenges for adapting mutant schemata and mutant deduplication are solved, we build ExpressAPR, a general-purpose patch validator for Java, which adapts all five classes of acceleration techniques suitable for patch validation. We choose Java as the target programming language because it is supported by most APR tools[2].

To achieve mutant schemata and mutant deduplication, we implement the execution scheduling algorithm described in Section IV and the interception-based instrumentation by following the design process described in Section V.

The `capture/replay` component for Java is similar to the IMP+ example as described in Section V, because both languages share similar features. For the control state, control flow statements in Java can only be `break`, `continue`, `return`, or `throw` [68], which are all covered by IMP+. For the data state, Java programs similarly change the data state by assignment statements. The `capture/replay` component for Java has three differences compared against IMP+: 1) Java allows to assign a label to a loop and jumps out of a labeled loop. To support this syntax, we detect labeled `break/continue` statements in the preparation step, and add the same label in the generated `replay` component. 2) Java methods can declare a list of checked exceptions to be thrown. Generated components should inherit this list from the original method. 3) A patch may access local variables in the original method and fields in the original class. To make it possible, we put generated methods in the same class as the original patch, and declare auxiliary fields for passing around local variables between the original method and the generated methods.

Adapting the three other acceleration techniques is straightforward: Test virtualization is adapted by resetting global states (except for the state-transition tree that should be shared across runs) before each round of test execution. Current ExpressAPR implementation uses an existing library, VMVM [32], for test virtualization. Test case prioritization is adapted by sorting test cases in the test suite. Following the two heuristics mentioned in Section III-A4, ExpressAPR first executes the failing test cases, then other test cases in the same package of a patched location, and then test cases in other packages. Parallelization is adapted by spawning multiple instances of the patch validator with a process pool,

[1] In the EVAL PATCHES procedure and when the test ends, we assert that the observed system state should be consistent with the record on the state-transition tree. For example, if the test execution finishes but Out: Status= "visited", the test case is considered unstable.

[2] program-repair.org lists 23 APR tools for repairing Java programs, which is the largest among all programming languages.

one instance dealing with a subset of patches, each subset corresponding to all patches to the same fault location as generated by the APR approach.

VIII. Experiment Setup

A. Research Questions

Our evaluation aims to answer these research questions.

RQ1. Overall performance. How fast patch validation could be with ExpressAPR?

RQ2. Technique effectivenessDoes each technique in ExpressAPR contribute to the final performance?

RQ3. Feasibility. Will ExpressAPR fail or report incorrect validation results, affecting the ability of APR?

RQ4. Generalizability. How does the speed and feasibility of ExpressAPR generalize to different kinds of programs?

B. The Benchmark

We used Defects4J [69] v1.2 as the benchmark for RQ1 through RQ3. Defects4J v1.2 is a widely used APR benchmark on Java, containing 395 bugs in open-source Java projects. We chose four publicly available APR tools covering different kinds of APR approaches to be studied in this evaluation:

Recoder [8], a deep-learning-based tool fixing 53 bugs in the benchmark.
TBar [2], a template-based tool fixing 41 bugs.
SimFix [70], a heuristic-based tool fixing 33 bugs.
Hanabi [71], a decision-tree-based tool fixing 27 bugs.

We obtained the replication package for each tool, and collected all generated candidate patches under the original time-out value for evaluation.

We did not conduct the evaluation on all 395 bugs in Defects4J because the computational cost would be unacceptable to execute multiple baselines for the patches generated by the four APR tools. Instead, we sampled the dataset in two possible ways: 1) for each studied APR tool, choosing all bugs it can fix as reported by its authors (the fixable dataset); 2) choosing 30 bugs randomly from the whole dataset (the random dataset). The fixable dataset helps us understand whether APR can be accelerated: if the time used in each fixable bug is shorter, we can use a shorter time limit without affecting the effectiveness of the tool. The random dataset is supplemented to avoid the threat of possible selection bias on the fixable dataset. Results on both datasets are reported.

In RQ4, we additionally experimented with Recoder under 30 random bugs in the IntroClassJava [72] benchmark, as a supplement to Defects4J. This benchmark includes 297 small and buggy student assignments, which allows us to understand the generalizability of ExpressAPR.

The statistics of evaluated patches is shown in Table II. The current evaluation costs over 18 months of single-CPU-core time in total, which is of the largest scale among all existing studies on accelerating APR within our knowledge.

TABLE II: Statistics of evaluated patches

RQ	APR	#Bugs	#PatchSets	#Patches	Avg. #Patches per Patch Set
RQ1 3 (fixable)	Recoder	53	3096	101543	32.8
	TBar	41	3138	110882	35.3
	SimFix	33	1799	120788	67.1
	Hanabi	27	959	99627	103.9
RQ1 3 (random)	Recoder	* 29	1956	34722	17.8
	TBar	* 29	4602	148140	32.2
	SimFix	30	3161	227041	71.8
	Hanabi	* 29	650	87990	135.4
RQ4	Recoder	30	866	36565	42.2

*: Among the 30 selected random bugs, Recoder and Hanabi fails to run on Lang-25, and TBar fails to run on Mockito-20.

C. Experiment Methodology

In RQ1, the patch validation time per bug using ExpressAPR is compared with the following two baselines:

Plain. Each patch is compiled using the `defects4j compile` command that ships with the Defects4J dataset, which uses an Ant script to compile changed files and perform necessary user-defined actions. Compilable patches are tested using the `defects4j test` command. This reflects the normal practice of evaluating APR techniques.

State-of-the-Art. UniAPR [22], the current state-of-the-art patch validator on Java, is used to validate all patches. It incorporates test virtualization, test case prioritization, and parallelization, but not mutant schemata and mutant deduplication, the two techniques brought to patch validation by ExpressAPR for the first time. When compiling the patches, we use the OpenJDK compiler (the `javac` command) to compile only the patched file.

To simulate the degree of parallelization on average hardware, the patch validation job for each defect is distributed to eight CPU cores for every baseline. In other words, parallelization is already included in all the baselines. To understand the performance without parallelization, we also report the time needed when we perform serial validation in the Plain baseline. However, for a fair comparison, we do not treat the serial validation as a separate baseline, because otherwise the acceleration ratio can be easily manipulated: the more CPU cores we use in our experiment, the higher the acceleration ratio for the parallel validation.

We also measure the memory footprint of each approach during the experiment.

In RQ2, we discuss the individual effectiveness of each acceleration technique in ExpressAPR. For Parallelization, its effectiveness is straightforward (roughly N times acceleration when running on N cores). For the other four techniques, namely Mutant Schemata (MS), Mutant Deduplication (MD), Test Virtualization (TV), and Test Case Prioritization (TCP), we empirically measure their effectiveness.

Among these four techniques, Mutant Schemata accelerates the patch compilation step, and the other three techniques accelerate the test execution step. Besides the original ExpressAPR implementation, we add intermediate configurations by removing each technique from ExpressAPR one by one, and measure their time usage. For the patch compilation step, we add one intermediate configuration (“-MS”); for the test execution step, we add three intermediate configurations (“-TCP”, “-TCP -MD”, “-TCP -MD -TV”). In this way, we can understand the effectiveness of the removed technique between

two adjacent con gurations. Note that this is not an ablation study due to implementation-level dependencies – it can be hard to individually remove a technique from ExpressAPR[3].

In RQ3, we compare the validation result ("plausible", "implausible" or "fails to validate") of ExpressAPR against the Plain baseline. The ExpressAPR implementation may fail to accelerate the validation of a patch if it detects a violation of two limitations described in Section VI. It may also produce an incorrect result if it fails to detect such a violation that affects the validation result. In this RQ, we evaluate how often these cases happen. We count the number of patches that ExpressAPR fails to validate or reports incorrect results (reported "plausible" when should be "implausible" or vice-versa), and analyze the reason.

In RQ4, we compare the validation time and the result of ExpressAPR against the patch validation command under the IntroClassJava dataset (similar to the Plain baseline). This helps us to understand how much the result of previous RQs generalize to small programs that may not have many redundancies. Due to the computational cost, We do not experiment with other repair tools or other baselines.

D. Experiment Settings

The experiment is run on a Xeon® 8270 CPU server with eight processes, each using one dedicated CPU core. For patches ExpressAPR "fails to validate", we fall back to the State-of-the-Art baseline in RQ1. To deal with candidate patches with dead loops, following an existing study [15], we set the test timeout to 5 seconds plus 1.5 times over the original test execution time. UniAPR and ExpressAPR have of ine procedures to measure the original test execution time or to analyze the purity of methods (the analysis averagely takes 130 seconds), and they are excluded from time usage.

IX. Result Analysis

A. RQ1: Overall Performance

The total patch validation time per bug in both datasets using each approach is shown in Figure 6. The Y-Axis is logarithmic due to the huge difference across baselines. Note that UniAPR fails to run on 24% of bugs, hence some points in the SOTA series are missing in the gure. This result brings the following ndings for the APR community:

1. ExpressAPR accelerates patch validation to a new degree in a variety of settingsFrom Figure 6 we can see that in every studied APR tool, patch validation with ExpressAPR consumes signi cantly less time compared with the two baselines for almost all bugs. In the xable dataset, ExpressAPR shows an acceleration of 137.1x over the Plain baseline, or 8.8x over the State-of-the-Art baseline on bugs where UniAPR successfully runs. In the random dataset, ExpressAPR shows a similar acceleration of 108.9x over Plain or 10.3x over State-of-the-Art. This indicates that the performance of ExpressAPR generalizes to different APR tools and different bugs.

[3]For example, Mutant Deduplication requires the state-transition tree to persist across executions, which is implemented as part of the Test Virtualization technique. Therefore, we cannot remove TV without breaking MD.

(a) xable bugs

(b) random bugs

Fig. 6: Patch validation time (seconds) per bug, on a logarithmic scale

(a) xable bugs

(b) random bugs

Fig. 7: Percentage of time spent for patch validation

2. Patch validation is no longer the speed bottleneck of APR if mutation testing techniques are systematically adapted. The percentage of time spent for patch validation when repairing each bug in the xable dataset is shown in Figure 7. When using the stock Defects4J command (the Plain baseline), patch validation takes more than 95% of the repair time. If the previous best patch validation approach (the State-of-the-Art baseline) is used, this portion is reduced to 75%–95%, still the bottleneck. ExpressAPR has reduced this portion to about 50%, so patch validation is now as fast as patch generation. This suggests that future APR acceleration work should also consider the patch generation phase.

Fig. 8: Total repair time (seconds) of xable bugs using ExpressAPR

Fig. 9: Memory footprint of patch validation over one hour

3. The speed of APR satis es the expectation of most users. In our experiment with an 8-core con guration, when ExpressAPR is used, the total repair time for the xable dataset has greatly reduced to less than 3 minutes for more than half bugs and less than 10 minutes for nearly all bugs, as shown in Figure 8. This execution time satis es the expectation of most users in an existing survey [12].

4. The memory overhead of ExpressAPR is negligible. While ExpressAPR theoretically has a memory overhead for the State-Transition Tree, we do not observe additional memory usage in the experiment. In fact, ExpressAPR uses much less memory compared with Plain, because it avoids heavy frameworks (Maven or Ant) when running tests. Figure 9 shows the memory footprint of all three approaches over the rst hour of the experiment, which validates the candidate patches in the xable dataset in random order.

5. The acceleration ratio from parallelization is close to the number of CPU cores used. To understand the performance without parallelization, we further randomly sampled 300 patch sets from the xable dataset, and validated them with only one process on a single CPU core (in contrast to eight processes each using one core). The time usage becomes 7.2 times as long as Plain, which is close to the number of CPU cores (8). This is intuitive because the validation of patches at different locations has no dependency on each other and is thus naturally parallelizable.

(a) xable bugs

(b) random bugs

Fig. 10: Effectiveness of each acceleration technique

B. RQ2: Technique Effectiveness

In this RQ, we separately measure the effectiveness of four acceleration techniques in its step (patch compilation for MS; test execution for MD, TC, and TCP). Results on both datasets are shown in Figure 10. Each point $(x\%, y)$ in the plot indicates that the time usage of this step among $x\%$ patch sets is within y seconds. We have the following findings:

1. Acceleration techniques in ExpressAPR are effective on their own. Figure 10 shows that every addition of technique contributes to a signi cant acceleration in its step. The average contribution of each technique is over 3x. Because the Y-Axis is logarithmic, the acceleration ratio can be directly read from the distance of two series.

2. With all techniques used, patch compilation takes more time than test execution for most patch sets. This can be con rmed by comparing the ExpressAPR series in (i) and (ii). This is because: 1) When TCP is used, most patches fail the rst few unit tests, which costs little time. 2) The portion of patches that do not compile is considerable, precisely 54%, 51%, 76%, and 52% for the four APR tools, and test execution is skipped for them. It suggests that future work may spend more effort optimizing patch compilation.

TABLE III: The feasibility of ExpressAPR

Category		% of patches in ... xable bugs	random bugs
Correct result		97.197%	98.782%
Validation failure	(patch limitation)	1.882%	0.693%
	(test limitation)	0.886%	0.524%
Result misclassi ed	(as plausible)	0.018%	0.000%
	(as implausible)	0.017%	0.001%

TABLE IV: Acceleration techniques in general-purpose patch validators

Technique	ExpressAPR (ours)	UniAPR [22]	PRF [15]	DSV [23]	Wehne et al [24]	AE [18]	FRTP [25]
Mutant Schemata	!				#1		
Mutant Dedup.	!						
Test Virtualization	!	!		#2			
Test Case Prio.	!	!	!	!	!	!	!
Parallelization	!	#3	!	#3	#3	!	!

#1: Does not discuss how to deal with uncompilable patches.
#2: Reuses JVM without test virtualization, leading to incorrect results.
#3: No built-in parallelization, but is parallelizable.

3. The effectiveness of Mutant Deduplication depends on the patch space. We can see that MD performs best with Hanabi, by comparing the "-TCP -MD" and the "-TCP" series in different gures. This makes sense because Hanabi, as a decision-tree-based APR approach, naturally produces many patches changing Boolean conditions. These patches are more likely to be test-equivalent because Boolean conditions are either true or false.

C. RQ3: Feasibility

In this RQ, we count the number of cases where ExpressAPR fails to accelerate and classify them by their reasons. We also compare the validation result reported by ExpressAPR and the Plain baseline for detecting incorrect results. The result is shown in Table III, which leads to the following ndings:

1. The acceleration feasibility of ExpressAPR is high. In the xable dataset and the random dataset, 97.197% and 98.782% of patches are correctly validated with acceleration. It indicates that the two implementation-level limitations do not affect the majority of patches.

2. ExpressAPR has a negligible negative impact on the effectiveness of APR. Most limitation violations are detected by ExpressAPR by reporting a validation failure. In our implementation, the plain validation approach is automatically used for these patches as a fallback, keeping the validation result correct. Only a very small portion of patches (0.035% of the xable dataset and 0.001% of the random dataset) are misclassi ed because of undetected unstable tests. Among them, some are implausible patches misclassi ed as plausible, which can be ruled out by a post-check using the fallback approach on all plausible patches. So only plausible patches misclassi ed as implausible will have a negative impact on the effectiveness of APR. Since this portion is very small (0.017% and 0.001%), we believe the impact is negligible.

D. RQ4: Generalizability

When evaluating with Recoder under the IntroClassJava benchmark, ExpressAPR achieves an acceleration ratio of 41.5x. 99.77% of patches can be accelerated. All accelerated patches have a correct patch validation result. Therefore:

1. The acceleration of ExpressAPR generalizes to smaller programs. The acceleration ratio under IntroClassJava (41.5x) is worse than Defects4J (108.9x for the random dataset), but is still very signi cant. The difference is possibly due to the fact that IntroClassJava is made of simple programs instead of large open-source projects. Therefore, test execution is less redundant with smaller codebases and fewer tests.

2. The feasibility of ExpressAPR is improved for smaller programs. Under IntroClassJava, ExpressAPR supports 99.77% patches and achieves 100% correctness, which is signi cantly better than the result under Defects4J (98.782% supported for the random dataset). The improvement is intuitive, because simple programs are less likely to contain advanced syntax (breaking the patch limitation) or randomness (breaking the test limitation).

E. Threats to Validity

Threats to *internal validity* might come from the possible faults when implementing ExpressAPR and performing the experiment. To avoid faults in our implementation, we have added assertions and sanity checks in the code. We have also manually inspected misclassi ed patches in RQ3 and did not nd faults in the nal implementation. To mitigate timing errors due to varying system load, we used the *cgroup* mechanism in Linux to assign one dedicated CPU core and enough RAM resources to each process.

Threats to *external validity* lie in the representativeness of the benchmark. To mitigate the threats, we experiment under both Defects4J (containing defects in open-source projects) and IntroClassJava (containing buggy student assignments), two widely used datasets for APR evaluation. We collect candidate patches from four recent APR tools, covering multiple kinds of approaches. Therefore, our results have a high chance of representing the general use cases.

X. Related Work

A. Accelerating Patch Validation

Existing approaches that accelerate patch validation can be categorized as *general-purpose* or *special-purpose*. General-purpose approaches [15, 18, 22, 23, 24, 25] take an arbitrary set of patches as input, while special-purpose approaches [19, 20, 21, 26, 27, 28] are designed for and rely on a speci c patch generation algorithm. Table IV summarizes acceleration techniques used in general-purpose approaches for easy comparison. Below we compare our work against these existing approaches in three aspects.

1. Our empirical contribution is to understand the overall performance of integrating all ve techniques and their relative accelerations on top of the other techniques. No existing approach has integrated all these techniques, so these empirical results were previously unknown.

2. Our technical contribution is a set of novel techniques to overcome the challenges when adapting mutant schemata and mutant duplication to general-purpose patch validation. Three existing approaches have adapted mutant schemata within our knowledge. Two of them [19, 21] are special-purpose: they are designed for a specific patch space that does not have the problem of uncompilable patches. Though the approach by Mehne et al. [24] is proposed as general-purpose, it does not discuss how to deal with uncompilable patches, which is one of our technical contributions. Four existing approaches have employed mutant deduplication within our knowledge. Three of them [18, 19, 21] statically detect the equivalence of patches and only reduce the patch validation time for fully equivalent patches. Mechtaev et al. [20] uses the test-equivalence relationship to prune the patch space to avoid generating test-equivalent patches. However, their approach works for only a few special cases (interchangeable expressions and swappable statements) but cannot detect equivalence or test-equivalence in general (e.g., cannot detect `x=2;` and `x++;x++;` are equivalent).

3. Some approaches [15, 21, 24, 26, 28] have adapted test case selection. We do not adapt this technique as it is subsumed by mutant deduplication, as discussed in Section III-C.

B. Mutation Testing

There are multiple techniques for mutation testing acceleration, as surveyed in Section III-A. Our technical contribution is to adapt two classes of them, namely mutant schemata and mutant deduplication, to general-purpose patch validation. Below we compare our work against related mutation testing approaches.

1. Existing approaches with mutant schemata [30, 31, 40] only allow pre-defined mutation operators against the program, which are designed not to cause compile errors. In comparison, we allow arbitrary changes to statements, so the space of mutation is significantly enlarged, and compile errors are handled by compilation isolation. ExpressAPR is more suitable for patch validation against mainstream APR tools, where the patch space is huge, and many patches cannot compile.

2. Among four existing approaches with mutant deduplication [34, 41, 42, 43], three approaches [41, 42, 43] can detect only fully equivalent mutants, and are weaker than ExpressAPR, which detects test-equivalent mutants. The Major framework [34] detects test-equivalence by interpreting mutants in a pre-pass. As discussed in Section IV-A, if a mutant's state transition deviates from the original states, it requires a static analysis process, which may be hard to implement precisely, or fail to detect some test-equivalent mutants if the analysis is imprecise. In comparison, our execution scheduling approach does not need such analyses, so it is easier to implement and detect more test-equivalent mutants.

C. The Effectiveness Aspect of APR

Many existing papers improve the effectiveness of APR by filtering or re-ranking candidate patches [73, 74, 75, 76]. While they may have a side-effect of improving efficiency (because plausible patches are ranked higher), we consider them orthogonal to our work. ExpressAPR performs only lossless acceleration and does not re-order patches. Therefore, ExpressAPR can be used together with such approaches to achieve a better performance.

Also, readers may wonder about whether ExpressAPR can improve the effectiveness of APR ("Can we fix more bugs in the same time budget?"). While this is an interesting question, it is orthogonal to our work: we can already study this without ExpressAPR. Vu et al. [77] has shown that given exponentially more time, the APR effectiveness only increases linearly or not at all. Therefore we do not expect that ExpressAPR can help APR tools repair significantly more bugs in its original time limit (generally several hours [12]). That being said, ExpressAPR is useful for this kind of study, because the time required for experiments can be greatly reduced.

XI. Conclusion

We surveyed mutation testing acceleration techniques and identified five classes of applicable techniques for general-purpose patch validation. We proposed two novel approaches, namely execution scheduling and interception-based instrumentation, to overcome technical challenges when adapting two of the techniques for the first time. Our large-scale empirical experiment has shown that patch validation can be dramatically accelerated and no longer be the time bottleneck. When acceleration techniques are systematically used, thousands of patches can be validated in minutes, satisfying the expectations of users.

The ExpressAPR artifact, including the source code with documentation, a command-line interface for APR users, a Docker image to reproduce the experiment, and raw experiment results, is available on GitHub [35].

References

[1] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *23rd international conference on software analysis, evolution, and reengineering*, vol. 1, 2016.

[2] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-based Automated Program Repair," in *28th International Symposium on Software Testing and Analysis*, Jul. 2019.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *31st International Conference on Software Engineering*, 2009.

[4] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *40th International Conference on Software Engineering*, 2018.

[5] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *6th international conference on software testing, verification and validation*, 2013.

[6] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *35th International Conference on Software Engineering*, 2013.

[7] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *19th international symposium on Software testing and analysis*, 2010.

[8] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug. 2021.

[9] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "On The Introduction of Automatic Program Repair in Bloomberg," *IEEE Softw.*, vol. 38, no. 4, Jul. 2021.

[10] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated End-to-End Repair at Scale,"

41st International Conference on Software Engineering: Software Engineering in Practice, 2019.
[11] X. Zhang, C. Zhu, Y. Li, J. Guo, L. Liu, and H. Gu, "Precx: Large-scale patch recommendation by mining defect-patch pairs," in 42nd International Conference on Software Engineering: Software Engineering in Practice, 2020.
[12] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust Enhancement Issues in Program Repair," in 44th International Conference on Software Engineering, 2022.
[13] J. Liang, R. Ji, J. Jiang, S. Zhou, Y. Lou, Y. Xiong, and G. Huang, "Interactive patch filtering as debugging aid," in International Conference on Software Maintenance and Evolution, 2021.
[14] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," Software quality journal, vol. 21, no. 3, 2013.
[15] A. Ghanbari and A. Marcus, "PRF: A framework for building automatic program repair prototypes for JVM-based languages," in 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Nov. 2020.
[16] C. Le 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, 2011.
[17] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair," in 42nd International Conference on Software Engineering, Jun. 2020.
[18] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in 28th International Conference on Automated Software Engineering, ASE, 2013.
[19] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in 32nd International Conference on Automated Software Engineering, 2017.
[20] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," Transactions on Software Engineering and Methodology, vol. 27, no. 4, 2018.
[21] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in International Conference on Software Engineering, May 2018.
[22] L. Chen, Y. Ouyang, and L. Zhang, "Fast and precise on-the-fly patch validation for all," in 43rd International Conference on Software Engineering, May 2021.
[23] R. Guo, T. Gu, Y. Yao, F. Xu, and X. Ma, "Speedup automatic program repair using dynamic software updating," in 10th Asia-Pacific Symposium on Internetware, Oct. 2019.
[24] B. Mehne, H. Yoshida, M. R. Prasad, K. Sen, D. Gopinath, and S. Khurshid, "Accelerating search-based program repair," in 11th International Conference on Software Testing, Verification and Validation, 2018.
[25] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," International Conference on Software Maintenance, no. March, 2013.
[26] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in 28th International Symposium on Software Testing and Analysis, Jul. 2019.
[27] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues, "VarFix: Balancing edit expressiveness and search effectiveness in automated program repair," in 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, vol. 1, Aug. 2021.
[28] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in 12th annual conference on Genetic and evolutionary computation, 2010.
[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in 29th International Conference on Software Engineering, 2007.
[30] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," International Symposium on Software Testing and Analysis, 1993.
[31] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a java compiler," in 26th International Conference on Automated Software Engineering, 2011.
[32] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," International Conference on Software Engineering, no. 1, 2014.
[33] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," International Symposium on Software Testing and Analysis, 2013.
[34] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," International Symposium on Software Testing and Analysis, 2014.
[35] Y.-A. Xiao, C. Yang, B. Wang, and Y. Xiong. (2023) The replication package for ExpressAPR. [Online]. Available: https://github.com/ExpressAPR/ExpressAPR
[36] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE transactions on software engineering, 2010.
[37] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," Advances in Computers, 2019, vol. 112.
[38] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," Journal of Systems and Software, vol. 157, 2019.
[39] H. Coles. (2021, Jul.) Mutation testing systems for Java compared. [Online]. Available: http://pitest.org/java_mutation_testing_systems/
[40] Y. S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," Software Testing Verification and Reliability, vol. 15, no. 2, 2005.
[41] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations." Georgia Inst of Tech Atlanta School of Information and Computer Science, Tech. Rep., 1979.
[42] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in 37th International Conference on Software Engineering, vol. 1, 2015.
[43] J. Pan, "Using constraints to detect equivalent mutants," Master's thesis, George Mason University Master's thesis, 1994.
[44] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," Software testing, verification and reliability, vol. 22, no. 2, 2012.
[45] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009.
[46] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High performance software testing on simd machines," IEEE Transactions on Software Engineering, vol. 17, no. 5, 1991.
[47] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a mimd computer," in Proc. ICPP, 1992.
[48] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, "MuVM: Higher order mutation analysis virtual machine for C," in International Conference on Software Testing, Verification and Validation, 2016.
[49] R. Gopinath, C. Jensen, and A. Groce, "Topsy-Turvy: a smarter and faster parallelization of mutation analysis," in International Conference on Software Engineering, 2016.
[50] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in International Symposium on Software Testing and Analysis, Feb. 2017.
[51] B. Wang, S. Lu, Y. Xiong, and F. Liu, "Faster mutation analysis with fewer processes and smaller overheads," in 36th International Conference on Automated Software Engineering, 2021.
[52] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," ACM on Programming Languages, vol. 2, no. OOPSLA, 2018.
[53] C. P. Wong, J. Meinicke, and C. Kästner, "Beyond testing configurable systems: Applying variational execution to automatic program repair and higher order mutation testing," in 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018.
[54] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," Software Testing, Verification and Reliability, vol. 4, no. 1, 1994.
[55] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," IEEE Transactions on Software Engineering, vol. 45, no. 9, 2018.
[56] W. E. Howden, "Weak mutation testing and completeness of test sets," IEEE Transactions on Software Engineering, no. 4, 1982.
[57] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," IEEE Transactions on Software Engineering, vol. 20, no. 5, 1994.
[58] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," Journal of Systems and Software, vol. 31, no. 3, 1995.
[59] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in 15th international conference on software engineering, 1993.
[60] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis," in Seke, vol. 9, 2009.
[61] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test