Iter-T: ITERative Test suite generation for automated program repair

Ariel Godio*, Simón Gutiérrez Brida†, Germán Regis†, Hamid Bagheri‡, ThanhVu (Vu) Nguyen§, Nazareno Aguirre†, Marcelo F. Frias¶
*Instituto Tecnológico de Buenos Aires, Argentina
agodio@itba.edu.ar
†Universidad Nacional de Rio Cuarto, Argentina
{sgutierrez,gregis,naguirre}@dc.exa.unrc.edu.ar
‡University of Nebraska-Lincoln
bagheri@unl.edu
§George Mason University
tvn@gmu.edu
¶University of Texas at El Paso
mfrias4@utep.edu

***Abstract*—Test-based automated program repair (TB-APR) techniques automatically fix buggy programs by relying on a failing test suite. This test suite serves a dual purpose: pinpointing bugs and evaluating the validity of potential patches. However, the effectiveness of TB-APR techniques in generating correct patches is highly dependent on the test suite utilized. The primary shortcoming of TB-APR techniques arises from the intrinsic incompleteness of test suites, resulting in a significant drawback: overfitting, i.e., the generation of 'overfitted patches', patches that pass the given test suites but fail to repair the subject program correctly regarding its more general intended behavior.**

To address this challenge, we present a novel technique designed to enhance the effectiveness of TB-APR methods by automatically generating test suites tailored for program repair. Unlike prior TB-APR techniques, it is rooted in the recognition that edge cases that invalidate overfitted patches play a pivotal role in guiding the repair process away from incorrect solutions. This technique leverages formal specifications and bounded verification to evaluate candidate patches and transforms the counterexamples (CEs) obtained from verifying candidate patches into tests for program repair. The efficacy of iteratively using such CEs as tests for TB-APR is substantiated by Iter-T our implementation of this technique for Java programs and JML specifications, evaluated on a benchmark of 717 buggy Java programs drawn from the APR literature. By progressively constructing test suites exclusively from CEs of overfitted candidate patches, Iter-T increases the odds of fixing a bug by about 58% compared to the originally provided test suites. Moreover, in cases where a TB-APR tool repairs a program using its original suite, employing CEs alone as test suites reduces the median time required to generate a correct patch by 42%. Remarkably, the generated CEs-based test suites are very small, accomplishing these results with only 2.4 tests on average.

***Index Terms*—Test suite generation, Automated program repair, Specification-based program repair, Bounded verification**

I. Introduction

Among the different stages of software development processes, it is widely acknowledged that an important part of software development is devoted to software debugging. The reason is well known: building software is extremely difficult, and despite the many techniques to improve the development of quality software, software bugs still make it to later development stages and deployed software. The amount of time and other resources that are dedicated to software debugging makes any effort put into improving the debugging processes highly relevant. This situation has led to an increasing interest in the development of tools and techniques that can assist developers in both software verification and software debugging. In effect, in the last decade or so, a variety of techniques have been proposed to automatically repair programs, i.e., given a buggy program, to attempt to generate a syntactic variant of the program that gets rid of existing bugs in it. Automated program repair techniques can vary in many aspects: how bugs are localized, how candidate repairs are generated, and how candidate repairs are assessed. In this last aspect, most automated program repair tools use *tests*. In fact, tests usually serve a dual purpose in program repair: they are employed for fault localization (i.e., identifying the statements in the program that are most suspicious to contain bugs) and for patch validation, i.e., to decide when a candidate patch is considered a fix.

Despite the inherently partial nature of tests as specifications, and the fact that existing test suites may originate from a variety of concerns not directly related to automated repair (e.g., to guide incremental coding in Test Driven Development [1] or to witness previous bugs maintained for regression [2]), tests are still the most widely adopted mechanism to decide the correctness of candidate repair patches. This has led to a prominent issue in program repair: patch overfitting [3]. A candidate patch is considered an overfit patch when, despite passing all tests in the suite used for patch validation, the patch does not meet the intended program behavior. The problem has been extensively studied by researchers, and various solutions have been proposed, including the use of high-coverage test suites, the development of (test-based) semantic-based repair

techniques that seek to provide more general patches, and the combination of program repair with automated test generation [3]–[6]. However, the intrinsic partiality of test suites as descriptions of the intended software behavior makes the overfitting problem a fundamental problem in most automated program repair approaches [3], [7].

Recently, there have been some efforts to deal with patch overfitting in the context of automated program repair by making software *specification* part of the repair environment. In particular, in Gissurarson et al. [5] specifications in the form of general assertions known as *properties* in the context of property-based testing [8], are used in combination with random testing to identify buggy programs and to assess the correctness of Genprog-like candidate patches [9]. The approach is implemented for Haskell and has been shown to help in reducing overfitting. Similarly, in Nilizadeh et al. [10], the program specifications given in terms of JML contracts [11] are used to automatically assess patches produced by automated program repair tools as correct fixes or overfit patches. Moreover, the OpenJML tool is used to produce a CE for overfit patches that, when added to the test suites used for program repair, reduced overfitting. Le et al. [12] proposed JFix, a semantics-based repair framework for Java that leverages symbolic execution and multiple synthesis engines to reduce overfitting. Their evaluation used independent, held-out test suites to measure patch generalizability, demonstrating that JFix's ability to combine several synthesis solvers significantly increases the likelihood of producing non-overfitting repairs. Shariffdeen et al. [13] introduced Concolic Program Repair (CPR), a technique that directly tackles the problem of patch overfitting by jointly exploring the patch and input spaces. Instead of relying solely on test suites, CPR leverages concolic execution to systematically generate new inputs and refine patch candidates against user-provided specifications, such as crash-freedom or assertion satisfaction. This dual exploration allows the approach to discard overfitting patches that pass the given tests but fail under unseen inputs.

In this paper, we present a technique that also leverages formal specifications and automated verification to iteratively produce test suites tailored for program repair. Iter-T, as the implementation of the presented technique is called, receives a program with a contract given in terms of pre- and post-conditions specified in the Java Modeling Language (JML) [11]. SAT-based bounded verification using the TACO tool [14] is used to verify the program against its contract, and, when a CE is found, it is automatically turned into a bug-revealing test case. The obtained test suite (which at this point contains a single test) is fed together with the buggy program to an off-the-shelf TB-APR tool. If a patch is obtained, it is checked against the specification using TACO again to check for (bounded) correctness, or alternatively to produce an additional bug-revealing test for a new repair attempt. This leads to an iterative process in which overfitted patches contribute new tests to the test suite until a fix (a correct, non-overfitted) patch is produced by the underlying TB-APR tool.

Contributions of the paper: The message we convey is that the test suites generated from tests derived along the automated detection of patch overfitting significantly improve the performance of the TB-APR tools and therefore are natural candidates to be used for TB-APR. We support this message with an empirical evaluation that allows us to answer the following research questions.

- **RQ1:** How effective are the test suites generated by Iter-T compared to the provided test suite for program repair?
- **RQ2:** How many iterations/test-generations does Iter-T require to converge to a repair?
- **RQ3:** What is the relative overhead of Iter-T compared to the underlying TB-APR tool when using only the provided test suite?

We show in Section IV-E, among other results, that the test suites generated by Iter-T increases the odds of fixing a bug by about 58% compared to the originally provided test suites. Moreover, the test suites generated by Iter-T are small, having, on average, 2.4 tests, and the median repair time is reduced by 42%.

While Iter-T, a language-specific implementation of the presented technique, generates test suites from verification attempts over JML-annotated Java methods using the bounded-verification tool TACO, the technique itself is general and may be ported to other programming languages using alternative tools. For instance, to repair C programs using CBMC [15] (a bounded model checker for C) instead of TACO and GenProg [16], as the TB-APR tool for C. However, we acknowledge that the availability, maturity, and expressiveness of specification languages and verification tools can vary significantly across programming languages, which may impact the effectiveness of this technique.

In summary, the contributions of this paper are: *(i)* a technique for generating test suites specifically tailored to enhance the performance of TB-APR tools, *(ii)* an implementation of the technique, called Iter-T, and *(iii)* an evaluation and results showing the effectiveness of Iter-T using existing benchmarks with existing TB-APR tools. In addition, this work also provides formal specifications for the well-known Introclass benchmarks and new test suites that are small but important for validating patch quality for buggy programs in existing benchmarks. All results and data are open source and are made available in Section IV-I.

II. Motivating example

Let us start with an example to demonstrate our approach. Listing 1 shows a buggy version of a Java implementation of binary search, equipped with its specification using JML. The bug has to do with an inaccurate check for the case in which the key was not found, and can be fixed by modifying line 27 as shown in Listing 2 using the `diff` format.

```
1  //@ requires (\forall int j;
2  //@   0 <= j && j < arr.length;
3  //@   (\forall int i;
4  //@     0 <= i && i < j; arr[i] <= arr[j]));
5  //@ ensures \result == -1 <==>
6  //@   (\forall int i;
7  //@     0 <= i && i < arr.length; arr[i] != key);
8  //@ ensures \result != -1 ==>
9  //@   0 <= \result && \result < arr.length &&
10 //@      arr[\result] == key;
```

```
11
12  public int binary(int[] arr, int key) {
13    if (arr.length == 0)
14      return -1;
15    else {
16      int low = 0;
17      int high = arr.length;
18      int mid = high / 2;
19
20      while (low < high && arr[mid] != key) {
21        if (arr[mid] < key)
22          low = mid + 1;
23        else
24          high = mid;
25        mid = low + (high - low) / 2;
26      }
27      if (low > high)
28        return -1;
29      return mid;
30    }
31  }
```

Listing 1. P_0: Buggy binary search method.

```
27c27
<     if (low > high)
---
>     if (low >= high)
```

Listing 2. A fix to method binary search.

The only pre-condition of this program states that the input array must be non-decreasingly sorted, and the post-conditions state that the result must be the index of the searched element `key` in the array, or `-1` if there is no such index.

Let P_0 be the program in Listing 1 and T_0 the empty test suite. First, we run the TACO bounded verification tool on all the methods of the class containing P_0. Since P_0 is buggy, it violates its specification and we obtain a CE from which we can extract the actual parameters of method `binary`: `arr = [2, 2]` and `key = 4`. With this input, we write a unit test t_0 and add it to T_0. After running t_0, we confirm that it is a bug-revealing test since it makes the method `binary` return `2` instead of `-1`. Now we have the buggy program P_0 and the test suite $T_1 := T_0 \cup \{t_0\}$ with at least 1 bug-revealing test (actually, T_1 only contains t_0), a common requirement to run TB-APR tools.

We now attempt to repair P_0. We use an off-the-shelf TB-APR tool and T_1. This produces one patch shown in Listing 3. This patch modifies one line of P_0, and the program resulting after applying the patch, namely, P_1, passes test suite T_1.

```
27c27
<     if (low > high)
---
>     if (arr != null)
```

Listing 3. First patch of method binary.

We are now in the same scenario as in the beginning. We have a (new) program P_1 equipped with specifications, so we verify P_1 against the specification and obtain another CE, from which we synthesize the test t_1: `arr = [6]` and `key = 6`, which is also a bug-revealing test. We add t_1 to T_1 and try to repair P_0 again, except this time we use $T_2 := T_1 \cup \{t_1\}$. This time, the TB-APR tool produces the patch shown in Listing 4. When applied to P_0, it produces the new program P_2.

```
27c27
<     if (low > high)
---
>     if (low == high)
```

Listing 4. Second patch of method binary.

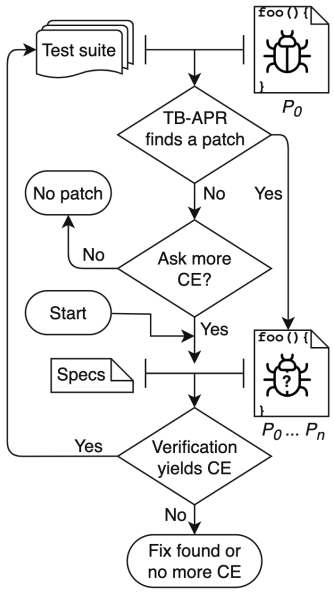

Fig. 1. Flowchart of Iter-T

At this point, it should be clear that this is an iterative process. P_2 is correct, but it could be the case that it is not, and we can continue to add bug-revealing tests and try to repair P_0. Even though P_2 is correct, we do not know this until we verify P_2 with TACO and get no CE.

Let us make some remarks on this process:

- We always try to repair the original program P_0 with a new, and more comprehensive test suite, each time.
- P_1 is an overfit patch.
- T_1 and T_2 are guaranteed to have at least 1 bug-revealing test for P_0.
- We do not know whether the remaining tests that were added are bug-revealing tests for P_0. What we *do* know is that these tests will prevent the TB-APR tool from producing the same overfitted patch as in previous iterations.
- The use of specifications has three main advantages. First, they allow us to assess the correctness of a patch. Second, they allow TACO to provide CEs whenever the patch is incorrect. Third, specifications allow us to single out the buggy methods from the non-buggy methods in a class. This allows us to concentrate the repair on specific buggy methods and automatically generate tests only to invoke such methods under repair (see Section III-D).

III. Iter-T under the hood

In this section, we describe in detail how Iter-T works. Iter-T is essentially a bash script that integrates various TB-APR tools (nine tools in total, described in Section IV-B), and the TACO verification tool, described in Section IV-C. Fig. 1 shows a flowchart that depicts its main aspects.

A. Initial Step

We start the repair process with P_0, the buggy program we want to repair, and the empty test suite T_0. We cannot yet use any TB-APR tool because it requires a test suite with at least one bug-revealing test and T_0 is empty. Instead, since P_0 is fully specified with a JML contract, we verify P_0 with the TACO bounded verification tool. In Fig. 1 P_0 would take the place of the bottom buggy `foo()`, the one with the question mark. If we do not get a CE, we cannot continue. This is equivalent to traditional TB-APR: When there are no bug-revealing tests in the provided test suite, the repair process cannot start. However, if we get a CE t, after checking that t is a bug-revealing test we add it to T_0 to obtain the test suite $T_1 := T_0 \cup \{t\}$. Now that we have a test suite with at least one bug-revealing test, we can proceed with running a TB-APR tool to attempt to repair P_0. At this point, P_0 would take the place of the top buggy `foo()` of Fig. 1. In Section III-E we explain why it may be the case that TACO produces no CE when verifying a buggy program, or why the CE it produces may not yield a bug-revealing test.

B. Iterative Step

After trying to repair P_0 using a TB-APR tool with the test suite T_1, there are 2 possible outcomes, described below.

If we get a patch, say P_1, we analyze it with TACO to assess whether it is correct for the given specification. If TACO does not find a CE, our repair process is complete and we classify P_1 as a correct patch. However, if we get a CE t, we proceed as in Section III-A: we check whether the test reveals a bug, add it to T_1 to get $T_2 := T_1 \cup \{t\}$, and try to repair P_0 again. In this case, P_1 is an overfit patch. Notice that we always try to repair P_0, and intermediate patches $(P_1, \ldots, P_n)$, when TACO deems them incorrect, are only used to enhance the test suite with the CE found. This prevents the TB-APR tool from producing previously overfitted patches.

If we do not get a patch, our process might be concluded. However, we ask TACO for a new CE (different from the previous ones) to add to the test suite T_2. One might think that there is no gain in doing so, as adding more tests to T_2 will only restrict the space of candidate patches that the TB-APR tool considers. Still, some tools benefit from this decision. Some TB-APR tools are non-deterministic (for example, ARJA-e [17]). Thus, just running the tool again (and with an extra test) might allow a patch to be generated.

Another argument supporting this decision is that some TB-APR tools might widen the space of candidate patches when a new test is added to the test suite. This is the case of Nopol [18], which prunes trivial patches from its search space (ones that would pass the given test suite but contain constant conditional expressions such as `if (true)` or `if (false)`. In this scenario, adding more tests could help Nopol synthesize a new patch that will not be pruned (without constant conditional expressions). While pruning these patches seems to be necessary to produce a correct patch, two observations have to be made: first, there are many bugs, at least on the studied benchmarks, that can be fixed with such a simple conditional expression. For example, `if (false)` is equivalent to removing a statement that might fix the bug. Second, Iter-T requires the underlying TB-APR tool to keep producing patches, regardless of their correctness. This is because further patches enable Iter-T to add more tests to the test suite, thus refining the production of patches on each iteration, leading eventually to a correct patch or to not having a patch at all (which is preferable to having an overfitted one). Finally, adding more tests might change the ranking of the source locations produced by the underlying fault localization technique, thus changing the space of candidate patches.

We limit the number of tests that can be added whenever the TB-APR cannot find a patch to three. In Section IV we provide empirical evidence supporting this decision. If the TB-APR tool still cannot find a patch after adding three tests on the current iteration, our repair process is finished without a patch.

C. Termination and Undecidability

APR is undecidable in general [19], as it combines undecidable sub-problems such as bug localization and patch validation. Iter-T addresses these theoretical limits by enforcing explicit practical bounds, which guarantee termination while yielding conservative results. In Sections III-A and III-B, we describe the algorithm implemented within Iter-T. Bug localization and patch generation are delegated to the underlying TB-APR tools, which are treated as black-box components whose execution is guaranteed to terminate by setting a timeout. Patch validation is performed by TACO using bounded verification, which restricts loop unrolling and input domains as explained in Section IV-C, making the verification problem decidable within a finite search space. Finally, each new call to TACO is constrained to produce a CE different from the previous ones, ensuring progress and preventing infinite refinement loops.

D. Writing Tests

In this section we describe how we create the tests to evaluate Iter-T and some considerations about these tests.

Whenever TACO analyzes a class, it is invoked on the methods of the class, and the latter are partitioned among (possibly) correct and buggy. If TACO determines that a method is buggy, i.e., it violates its specification, it produces a CE from which a bug-revealing test is synthesized. This test builds the receiver object on which the buggy method will be invoked, as well as the parameters and static attributes. The test then invokes *only* the method under repair, which of course might call other methods as well. It is worth remarking on the important role contracts play in Iter-T since they allow us to identify buggy methods. We believe that this may help the underlying TB-APR tool in the fault localization stage.

When the verification tool finds a violation of the post-condition, it typically produces a concrete input. For such inputs, we determine the corresponding expected outputs to fully construct a unit test, including the appropriate assertion. If the reported failure corresponds to an unhandled exception, no expected output is considered. Thus, we need an oracle to determine the expected output. The benchmarks used in

this work come with reference implementations for every case study that we use for this purpose (we run the reference implementation on the produced input and collect the obtained output to write the corresponding assertion). In a real-world setting, reference implementations are typically not available. However, there exist other viable alternatives given the fact that we have a (violated) post-condition for the identified CE inputs. An alternative is to employ a run-time assertion checker, e.g., the one accompanying OpenJML [20], to extract the assertion checking code and use it directly as the unit test assertion. Another alternative is to query the solver underlying the verification tool, TACO in our case, to produce concrete expected values (values that satisfy the post-condition) that correspond to the input from the CE. In this last option, we do not need to encode the program (which is incorrect) or the pre-condition (which the inputs of the CE are guaranteed to satisfy); only the post-condition, instantiated with the inputs of the CE, suffices. This is a significantly cheaper SAT problem to solve than the verification problem, and thus the SAT overhead should be minimal.

Currently, the conversion of CE into unit tests is semi-automated. The approach relies on manually written test templates containing placeholders that are instantiated with concrete values extracted from the CE. After invoking the failing method, these tests assert the post-conditions and class invariants. While such manual templating would not be practical in a realistic setting, the primary goal of Iter-T is to evaluate and compare the repair capabilities of existing, off-the-shelf TB-APR tools when supplied with CE-derived test suites. Fully automating this step is orthogonal to our evaluation and can be achieved using existing techniques, such as TACO's built-in support for generating unit tests via Java reflection or the automated CE-to-test translation approach proposed by Nilizadeh et al. [21]

Another observation that has to be made is that Iter-T only adds bug-revealing tests to the suite, but this does not necessarily mean that the TB-APR tool has only bug-revealing tests available for repair. For example, consider a case where Iter-T iterates 3 times to repair P_0. The intermediate patches P_1 and P_2 are overfitted, and P_3 is a fix. In this case, T_3 would consist of 1–3 bug-revealing tests for P_0, 1–3 for P_1, and 1–3 for P_2. Notice that the bug-revealing tests for P_1 and P_2 might be passing tests for P_0.

E. No CE - No Bug-revealing Test

This section describes some issues regarding the finding of CE and bug-revealing tests.

When the TB-APR tool cannot find a patch, we ask TACO for more CEs following the procedure described in Section III-B. However, there may be no more CEs. This happens when there are a reduced number of inputs that cause the program to violate its contract. For example, consider the buggy program in Listing 5, corresponding to bug 9 of class `FindFirstZero`. The method `findFirstZero` should return `-1` when the array is empty; however, it returns `0`. If the TB-APR tool cannot find a patch only with this test, we cannot continue the repair process, since there are no new CEs to enhance the test suite. This is the case of Nopol, jGenProg, and Cardumen, three of the TB-APR tools used in the experimental evaluation (see Section IV-B). The same scenario arises when asking TACO for a new CE (different from the previous ones) as described in Section III-B, but all possible CEs have already been found and the TB-APR tool could not find a patch.

```
1    public static int findFirstZero(int[] x) {
2      if (x.length == 0) {
3        return 0;
4      } else {
5        // Code to handle non-empty arrays
6      }
7    }
```

Listing 5. Method with only one input exposing the bug.

Even if TACO finds a CE, it is possible that, when translated into a test, this results in a positive test. Consider again the bug shown in Listing 1, which can be fixed by modifying the guard at line 27 as shown in Listing 2. ARJA-e (one of the TB-APR tools used in the experimental evaluation, see Section IV-B) finds a patch for this bug that is shown in Listing 6: it inserts a new assignment statement just before line 27.

```
26a27
>      low = mid + 1;
```

Listing 6. Patch generated by ARJA-e.

The patch found by ARJA-e is overfitted and fails with an array of length `Integer.MAX_VALUE` and a key greater than all the elements stored in the array. With such inputs, the patch produced by ARJA-e will lead to an overflow setting `low` with `Integer.MIN_VALUE`, making `binary` return `mid` (which equals `Integer.MAX_VALUE` and is not even a valid index) instead of `-1`.

When this patch is analyzed with TACO, it finds the following CE: `arr` = {`0, 0, 0, 0, 0, 0, 0`} and `key = 2` which does not expose the bug when run as a test. However, this is not an error of TACO. TACO can be instructed to consider the full bitwidth of Java integers or a restricted one (see Section IV-C), and in this particular case it was instructed to use a bitwidth of 4. Let us name the Integer class with a restricted bitwidth of n $\texttt{Integer}_n$; then, $\texttt{Integer}_4$ allows values ranging from `-8` to `7`. With this setting, TACO finds the same failure scenario: an overflow occurs setting the variable `low` to $\texttt{Integer}_4$`.MIN_VALUE`, making `binary` return `mid` (which equals $\texttt{Integer}_4$`.MAX_VALUE`) instead of `-1`. Notice that the length of the array `arr` is also $\texttt{Integer}_4$`.MAX_VALUE`. This case is correctly identified by Iter-T as a "possible overflow-related bug" and the repair process is concluded since Iter-T could not provide a bug-revealing test. Still, it is important to make some remarks:

- TACO found a bug that manifests itself only with arrays of length `Integer.MAX_VALUE` using a restricted bitwidth of 4.
- Even if TACO was run with an unrestricted bitwidth, it would not be easy to run a test holding such a big array, since it would require almost 8 GB of heap memory only to store the array.

F. The Contracts

In this section, we discuss some issues regarding the use of contracts in this work. Contracts have many applications, and many times contracts are written with a specific verification task in mind, and even on the assumption of implicit properties of the code being verified. For example, if one only wants to verify that a given method does not throw exceptions, there is no need to provide a specific post-condition (other than the implicit "no exception thrown"). Another example concerns sorting: a typical post-condition for sorting algorithms only states that the resulting array or collection is increasingly sorted (this is sufficient in cases in which, e.g., array modifications are only performed by array value swapping). However, a method that just returns an empty array for whatever input array it receives, would satisfy such (weak) specification (an additional assertion stating that the resulting array must be a permutation of the input array should be added to prevent these incorrect outputs). Iter-T uses the contracts as the oracles for repair: imprecise or weak contracts might lead Iter-T to produce overfitting and incorrect specifications, in the sense of not satisfying the intended behavior of the program. Notice that the same applies to traditional TB-APR techniques: the inherent incompleteness of test suites makes them weak as oracles, and an incorrect test case may lead to incorrect repairs. Contracts, however, provide at once a specification of the intended behavior of a program, replacing the need for possibly infinitely many tests of an equivalent test-based specification.

The contracts in the studied benchmarks do not consider the time complexity. Then, for example, a $\Theta(n)$ patch for a program expected to be $\Theta(\log n)$ would be deemed correct if the input-output behavior is correct. This is the case of the correct patch found by ARJA-e for method `binary`, which is shown in Listing 7. The bug was introduced in line 14 and ARJA-e patched the same line. The resulting program performs a linear search instead of a binary search, but still produces the expected result. The same issue is discussed in Nilizadeh et al. [22] and we got similar results regarding a change in time complexity. In this work, as in Nilizadeh et al. [22], these patches are considered correct.

```
1  public int binary(int[] arr, int key) {
2    if (arr.length == 0)
3      return -1;
4    else {
5      int low = 0;
6      int high = arr.length;
7      int mid = high / 2;
8
9      while (low < high && arr[mid] != key)
10       if (arr[mid] < key)
11         low = mid + 1;
12       else
13         high = mid;
14       mid = low - (low - low) / 2;
15     }
16     if (low > high)
17       return -1;
18     return mid;
19   }
20 }
```

Listing 7. Correct $\Theta(n)$ patch for binary search.

Another issue arises when the specification does not capture the expected behavior of the program but only the input-output relation. Consider the buggy method `timer` from class `Time` of Listing 8. The method `decr` decreases the time by one second, and the loop guard should be `!isTimeZero()`. However, the specification only states that the final state must be `hour == 0 && minute == 0 && second == 0`, which does not prevent a patch that sets these fields to `0` while at the same time removing the while loop from being considered correct.

```
1  //@ ensures hour == 0 && minute == 0 && second ==
       0;
2  public void timer() {
3    while (isTimeZero()) {
4     decr();
5    }
6  }
```

Listing 8. Buggy method timer.

The contracts in the BuggyJavaJML benchmark (one of the studied benchmarks, see Section IV-A) include invariants and variant functions for loops. Unlike the pre- and post-conditions, these contracts are tied to the body of the method they were written for. Although they are useful to verify *that* specific implementation, they may generate spurious CEs during verification that lead to false positives in the testing process. In effect, if a correct patch (a fix) is produced by the TB-APR tool that modifies the loop body in a way that violates the loop invariant or in a way that does not decrease the value of the variant function, TACO would deem that (correct) patch as incorrect. In particular, the invariants included in BuggyJavaJML were written for the correct version of the programs, those that once mutated spawned all the bugs of the benchmark. Including these contracts during the verification phase sets the unrealistic expectation of TB-APR tools undoing the mutation in a syntactic sense.

Iter-T ignores these contracts and the rationale behind this decision is presented below. Consider the correct patch generated by ARJA-e for method `addLoop` of Listing 9.

The bug was introduced in line 8: `while (n >= 0)` instead of `while (n > 0)`. However, ARJA-e managed to fix this bug by adding the assignment `sum = sum - 1` at line 4. Now, let us analyze the invariant at line 7. Recall that the loop invariant must hold at the beginning of each iteration, and upon loop termination, the invariant and the negation of the loop guard must also hold. The patch of Listing 9, while correct for the input-output behavior, does not satisfy this invariant: first, `sum == x + y - n` does not hold at the beginning of the loop (`y` equals `n` and `sum` equals `x - 1`). Second, immediately after the loop, the negation of the loop guard `!(n >= 0)` conjoined with the invariant (consider its second conjunct) `0 <= n` must hold; however, this is a contradiction. Considering this invariant during the verification phase, any pair of integers `(x, y)`, where `(y > 0)`, would produce a CE, yet none of them would produce a bug-revealing test. Even if the program is not correct for the expected input-output behavior, a CE might expose a violation of the invariant with a set of inputs that do not expose a violation of the post-condition. In the last case, the execution of Iter-T would

terminate, since there is no bug-revealing test to continue the repair process.

```
1  public int addLoop(int x, int y) {
2    int sum = x;
3    if (y > 0) {
4      sum = sum - 1;
5      int n = y;
6
7      //@ maintaining sum == x + y - n && 0 <= n;
8      while (n >= 0) {
9        sum = sum + 1;
10       n = n - 1;
11     }
12   } else {
13     // Code to handle y <= 0
14   }
15   return sum;
16 }
```

Listing 9. Fix for addLoop not complying with the invariant.

There are other issues related to loop contracts. Let us consider a patch, most likely an intermediate one, that completely removes a loop while keeping the loop contract. This might lead the verification tool to an error since there is a missing loop. The converse situation is also possible: a patch that adds a new loop but its contracts would be missing.

Finally, the use of loop contracts might be necessary to find a CE. After TACO unwinds the loops with the user-provided upper bound (see Section IV-C), it adds the necessary constraints to the underlying SAT problem to prune any path of execution that exceeds the aforementioned bound. By doing so, it might prune *all* the paths if the program loops infinitely for every valid input. Then, the verification would pass vacuously producing no CE. The current version of Iter-T does not address this issue, but it can be instructed to enable loop contracts and restart the verification phase whenever it produces no CE. Some cases fall into this category and are discussed in Section IV.

IV. Evaluation

In this section, we present the results of the experimental evaluation and address the research questions RQ1–RQ3.

A. Benchmarks

To evaluate Iter-T, we use two benchmarks: BuggyJavaJML [22] and IntroclassJava [23], resulting in a total of 717 different bugs. Although TACO is precise in verifying the supported constructs of Java and JML, it does not support all of them, such as the Java types `String` and `Short`, or control statements like `continue` and `switch`. Regarding the JML constructs, TACO does not support the full JML language specification. As a result, the number of bugs considered in this work is a subset of the original benchmarks. In particular, all bugs from the IntroclassJava benchmark output strings that are compared against the expected ones in the test assertions, so we replaced every output with integers, e.g., `"5 is the smallest"` with `5`. During this instrumentation, many bugs involving incorrect or misspelled strings with otherwise correct values, e.g., `"5 is the smallesst"` were corrected. From the original BuggyJavaJML benchmark containing 597 bugs, we removed 48 bugs, and from the original IntroclassJava benchmark containing 297 bugs, we removed 129 bugs.

The bugs in BuggyJavaJML consist of mutants generated with PiTest [24] from correct versions of a variety of programs. Each bug comes with a specification in JML and a test suite generated using Kelinci [25].

The bugs in IntroclassJava consist of student-written versions of 6 small C programming assignments in an introductory, undergraduate, C programming course that were automatically translated into Java. IntroclassJava includes an instructor-developed solution for each of the 6 assignments but does not come equipped with JML specifications. We took the specifications from [26], which were written using Code Contracts [27] for Pex [28], and translated them into JML. We then verified the solutions developed by the instructor against these specifications to assess their correctness.

The test suite provided in this benchmark is a combination of white-box tests, generated with KLEE [29] and manually written black-box tests. The availability of specifications on these benchmarks makes them suitable for this study.

B. TB-APR Tools

To evaluate Iter-T we use nine TB-APR tools that we describe below.

- AlphaRepair [30]: A cloze-style APR approach that uses large pre-trained code models under a zero-shot learning setting to directly generate patches.
- TBar [31]: A template-based technique that systematically attempts to apply fix patterns to program bugs.
- ARJA-e [17]: An evolutionary approach to program repair, more precisely genetic programming, that combines two sources of fix ingredients: the statement-level redundancy assumption and repair templates.
- jGenProg [32]: A redundancy-based repair approach that synthesizes candidate fixes from existing code written somewhere else in the system under repair.
- jKali [32] and KaliA [33]: Java versions of Kali [7] from the Astor [32] and ARJA [33] repair libraries, respectively. jKali and KaliA carry out an exhaustive search of the repair space and implement three syntactical changes: removal of statements, change of if-conditions to `true` and `false`, and insertion of return statements.
- jMutRepair [32]: A repair approach that mutates suspicious if conditional statements, performing one single change to the condition, including relational, logical, and unary operators.
- Cardumen [34]: The Cardumen mode of Astor [32]. A program repair algorithm that is designed to maximize the number of test-suite adequate patches based on mining repair templates. It uses a probabilistic heuristic for prioritizing candidate patches.
- Nopol [18]: Nopol dynamically gathers information about if-conditions. Then, it transforms this information into an SMT problem, and the solution to this SMT problem is then translated into a corrected if-condition.

C. Verification Tool

TACO [14] (for **T**ranslation of **A**nnotated **CO**de) is a tool for bug detection in Java methods. It receives as input a Java method annotated with a JML contract and produces an input exposing a failure (if one such input exists) or informs that no bug has been detected in the method under analysis. TACO performs a *bounded* verification, that is, it searches for bugs under some constraints:

- The number of loop iterations is bounded by the user, and, in the experiments we conducted, it is usually set to 4 iterations. In some experiments, it had to be increased to detect bugs in the source programs. The actual values for each experiment can be found in the replication package (see Section IV-I).
- Data domain sizes are also bounded to some maximum sizes. For the integer primitive data type, TACO allows the user to constrain the range of integers to be considered via the prescription of a *bitwidth*, i.e., the number of bits to be considered for representing integer numbers when using 2's complement arithmetic. TACO also allows for the use of full-fledged integer representations of integers as in Java (with 32 bits) and also full-fledged representations of longs (64 bits) and floats (32 bits).

These parameters are not independent and must be chosen consistently: a bitwidth that is too small with respect to the loop unrolls may cause overflows before reaching their intended iterations, yielding spurious CEs, while an excessively large bitwidth increases verification costs without improving failure detection. We select low but sufficient values for both loop unrolls and bitwidth, ensuring that TACO can expose violations while keeping bounded verification tractable. Section IV-F includes a sensitivity analysis for these parameters.

Internally, TACO translates the contract-annotated code to an Alloy [35] model that is analyzed using the Alloy Analyzer [35] with the aid of an off-the-shelf SAT-solver. If a bug is detected by TACO, from the satisfying valuation produced by the SAT-solver a program input is synthesized by TACO. When the method is executed, this input exposes the contract violation.

D. Evaluating Patch Correctness

Even though we use TACO to assess the correctness of candidate patches, every patch when deemed correct is subject to additional analysis to capture situations where the specification does not capture the bug. For every patch of the BuggyJavaJML benchmark, we run two independent analyzes: *(i)* we use an off-the-shelf `diff` tool to compare side-by-side the patch under analysis and the reference implementation. In the vast majority of cases, the fixes are trivially equivalent to the reference implementation. And *(ii)* we check that the patch under analysis passes the suite provided for the corresponding case study.

IntroclassJava does provide reference implementations, but the bugs on this benchmark are not derived from them as in BuggyJavaJML. Instead, they are implementations of the same problem from scratch, making it infeasible to use a `diff` tool. To further evaluate the correctness of these patches, we used bounded exhaustive suites of 1000+ tests taken from [26].

This additional correctness assessment showed that Iter-T achieves a 92% precision rate.

Note that we do not distinguish patches that are syntactically identical to the reference solution from those that are semantically equivalent but syntactically different. Instead Iter-T relies on TACO to assess the correctness of candidate patches.

E. Experimental Evaluation and Research Questions

In Section III we describe how Iter-T works. To evaluate Iter-T, we compare the performance of the TB-APR tools running with the test suites generated by Iter-T and with those that accompany each bug in the benchmark. Regardless of the origin of the test suite, whenever a TB-APR tool finds a patch with the benchmark suite, we analyze it with TACO to assess its correctness. If a CE is found and produces a bug-revealing test, we classify the patch as overfitted; otherwise, we evaluate the patch as described in Section IV-D.

From now on, we refer to a buggy program as a *bug*, e.g. `bug3` of class `Time` from the BuggyJavaJML benchmark, and we call *case study* or *case* for short, the application of a TB-APR tool to a specific *bug*. With a total of 717 bugs and 9 different TB-APR tools, we generated 6453 *cases*, each of which we evaluated using 2 test suites: the benchmark-provided suite and the one generated by Iter-T, resulting in a total of 12906 experiments.

Let us start with a summary of the results. Table I shows the overall results without distinction of the benchmarks or the TB-APR tools used. This table has a twofold aim: to enable us to provide answers to the RQs and to help the reader gain a high-level understanding of the results and possible outcomes of the experiments.

The first column of Table I lists all possible outcomes of the experiments described below. Note that it does not include "plausible patch" as a possible outcome since these are immediately subjected to verification and classified as either fix or overfit. What is commonly referred to as a "plausible patch" in the APR literature corresponds, in Iter-T, to an intermediate patch.

- Fix: The generated patch is deemed correct.
- Overfit: The generated patch is overfitted.
- No patch: No patch was generated.
- No negative: No bug-revealing test is available to run the TB-APR tool.
- Timeout: The TB-APR tool reached the one-hour timeout.
- Error: There was an error running any of the tools involved.

The last column of Table I shows the number of cases that fall into each difference, the intersection and the union. That is, there were 1875 cases fixed with either source of the test suite, of which 162 (8.6%) were fixed only using the benchmark suite, 1249 (66.6%) with both suites, and 465 (24.8%) only with Iter-T. The second row presents the same numbers broken down by bugs instead of cases. That is, if a bug is fixed with at least one TB-APR tool, we report it here. Remarkably, there are 68 bugs that can only be fixed with Iter-T. The

percentages accompanying each value in Table I correspond to the difference Iter-T makes over the benchmark suite, e.g., there is an increase of 21% in the number of fixes when we use the suite generated by Iter-T. 18 out of the 162 cases fixed only using the benchmark suite correspond to situations where the TB-APR tool could not find a patch even after adding three tests to the current iteration.

While reporting raw proportions gives a first glimpse of the results, to assess whether this difference is statistically significant while accounting for variability across classes, bugs and tools, we performed a mixed-effects logistic regression with random intercepts for class, bug, and tool. This model isolates the effect of the test suite itself. The analysis shows that Iter-T increases the odds of fixing a bug by about 58% compared to the originally provided test suites (odds ratio = 1.58, 95% CI $[1.48, 1.70]$, $p < 0.001$), confirming that the improvement is robust and not driven by particular bugs or tools.

These numbers show a significant difference in favor of Iter-T providing a clear answer to RQ1. The test suites that Iter-T produces improve the performance of the TB-APR tools compared to that of using the ones originally provided within the benchmarks. The number of overfitted patches is decreased by 77%. This result is further supported by a similar logistic model as before, which indicates that Iter-T reduces the odds of overfitting to about 14% of the odds under the provided test suites (odds ratio = 0.14, 95% CI $[0.12, 0.17]$, $p < 0.001$). This is aligned with the way Iter-T synthesizes the test suite. Nevertheless, there are 155 cases for which Iter-T did not provide a bug-revealing test. This usually happens because the specification is not strong enough to expose the bug, or a loop in the buggy method does not terminate.

The number of cases classified under the No patch outcome increases by 10%. While this might initially appear to be a negative result, the primary goal of Iter-T is to provide a test suite tailored for program repair, guiding the TB-APR tool to either produce a valid fix or conclusively determine, through unskippable bug-revealing tests, that it cannot fix the bug. In other words, Iter-T shifts cases from the overfit outcome to either the Fix or No patch outcomes.

The inherent incompleteness of test suites not only results in overfitting but also may fail to provide bug-revealing tests for the to-be-repaired program, preventing the TB-APR tool from starting. The cases classified under the No negative outcome illustrate this scenario. The benchmark test suite did not include a single bug-revealing test for 423 cases. Iter-T is also subject to this problem resulting in a total of 125 cases, TACO could not find a bug-revealing test in the first verification attempt. The reason is the same as that for the overfitted patches described earlier, except that happening on the first iteration discards the possibility of calling the TB-APR tool.

Before we continue with the Timeout outcome, recall that Iter-T might call the TB-APR tool multiple times so we compare the sum of all calls against the timeout to classify accordingly. This outcome also shows an improvement for Iter-T. We believe this is related to the size of the suites that Iter-T produces, which we discuss later. Finally, the number of errors incurred by the underlying tools, while small with respect to the 6453 cases, is 246% greater for Iter-T. This is not surprising, if we consider that the experiments with the benchmark test suite require just 1 call to the TB-APR tool and, at most (if there is a patch) 1 call to TACO. While Iter-T calls these tools up to 3 times (each) per iteration.

The rest of this section focuses mainly on the fixes found.

TABLE I
#CASES BY EXPERIMENT OUTCOME: BENCHMARK TEST SUITE VS. ITER-T GENERATED TEST SUITE

Outcome	**Benchmark** A	**Iter-T** *(Diff)* B	$A \setminus B$	$A \cap B$	$B \setminus A$	$B \cup A$
Fix (bugs)	535	576 *(+7%)*	27	508	68	603
Fix (cases)	1411	1713 *(+21%)*	162	1249	465	1875
Overfit	685	155 *(-77%)*	657	28	127	812
No patch	3733	4111 *(+10%)*	315	3418	692	4425
No negative	423	125 *(-70%)*	366	57	68	491
Timeout	124	104 *(-16%)*	109	15	89	213
Error	77	246 *(+219%)*	22	55	191	268
Total	**6453**	**6453** *(-%)*		-		

TABLE II
#FIXES BY TB-APR TOOL AND BENCHMARK: BENCHMARK TEST SUITE VS. ITER-T GENERATED TEST SUITE

TB-APR tool / **Benchmark**	**#Cases**	**Benchmark** **#Fixes**	**#Tests**	**Iter-T** **#Fixes** *(Diff)*	**#Tests**	**#It.**
AlphaRepair	717	474	61	420 *(-11%)*	2.4	2.4
ARJA-e	717	266	55	402 *(+51%)*	2.7	2.6
Cardumen	717	64	52	111 *(+73%)*	2.0	1.9
JGenProg	717	57	34	56 *(-2%)*	1.5	1.5
JKali	717	21	17	20 *(-5%)*	1.0	1.0
JMutRepair	717	145	47	183 *(+26%)*	1.6	1.6
KaliA	717	43	51	46 *(+7%)*	2.0	2.0
Nopol	717	53	33	97 *(+83%)*	5.3	4.3
TBar	717	288	47	378 *(+31%)*	1.9	1.9
Total	**6453**	**1411**	**52**	**1713** ***(+21%)***	**2.4**	**2.3**
BuggyJavaJML	4941	1198	59	1489 *(+24%)*	2.2	2
IntroclassJava	1512	213	15	224 *(+5%)*	2.8	3
Total	**6453**	**1411**	**52**	**1713** ***(+21%)***	**2.4**	**2.3**

Having presented the overall results, we can break down the cases by TB-APR tool and benchmark to discard any bias. Table II shows the number of fixes found by each TB-APR tool for both test suites. For example, ARJA-e produced 266 fixes using the benchmark test suite, with an average of 55 tests per suite, and 402 fixes using the suite provided by Iter-T, increasing 51% the fixing ratio. The suites produced by Iter-T for ARJA-e have an average of 2.7 tests and ARJA-e had to patch each bug 2.4 times on average to find a fix. Table II shows an increment of the fixing ratio for most tools. Interestingly, jKali required only 1 iteration and 1 test to find a fix for each of the 10 bugs, when run within Iter-T.

AlphaRepair, just as many large language model-based (LLM) repair tools [36]–[39], produces a pool of patches rather than a single candidate. While this demonstrates the expressive power of such tools, it introduces the question of patch selection. Iter-T picks one of them at random, following the ranking of buggy statements, to continue to the next

iteration. However, when running the experiments with the benchmark's suite, we lack such a mechanism and report the experiment as a fix if we have at least one plausible patch.

Although this approach may overestimate the number of fixes, we chose to favor the baseline technique, believing that the comparison still highlights the benefits of Iter-T. In particular, Iter-T automatically filters plausible patches and executes less code through its minimal test generation strategy, thereby shortening the ranking of buggy statements and reducing repair time.

One might wonder whether multi-patch repair tools benefit differently from iterative test suite augmentation. Iter-T operates on a uniform abstraction that effectively treats single-patch and multi-patch tools equivalently, as Iter-T always selects and evaluates a single patch per iteration, independently of how many candidate patches were originally produced.

When supported by the APR tool, we explicitly configure it to generate a single candidate patch, as this is sufficient for Iter-T and avoids wasting time exploring parts of the search space that would be immediately discarded in later iterations. AlphaRepair, however, does not expose such a configuration option, therefore it produces multiple candidate patches by design.

Table II also breaks down the cases by benchmark, showing a better performance of Iter-T for both of them. For Buggy-JavaJML, the suites generated by Iter-T allow one to fix 24% more cases. Similarly, for IntroclassJava, the number of fixes grows by 5%.

Since Iter-T initiates the test suite generation process from an empty test suite, the number of iterations required (and tests generated) to converge to a test suite that allows the TB-APR tool to produce a non-overfitting patch may be prohibitively large. RQ2 dwells on this problem. Table II shows that 2.3 iterations are required on average, and 2.4 tests are generated. Notice that the number of tests is substantially smaller than the number in the provided test suites which contain, on average, 52 tests. While it is clear that both suites have different goals and should therefore not be compared regarding their usefulness as specification artifacts (the provided suites should be definitely better in that sense), Fig. 2 shows that the distributions of the time required by TB-APR tools to find a fix are skewed toward lower values when they are fed with the Iter-T test suites. This trend applies to all TB-APR tools except jKali, for which the execution times remain nearly the same. Fig. 2 is limited to the 1249 cases in which both test suites led the TB-APR tools to a fix. Recall that Iter-T increases the odds of fixing a bug by about 58%. For Iter-T, only the last call to the TB-APR tool, the one that successfully produced a fix, is considered.

As with proportions in Table I, the times reported in Fig. 2 do not account for variability across classes and bugs. For this reason, we fit these times to a log-transformed linear mixed-effects model with the same random intercepts as before to isolates the effect of the test suite itself. The analysis shows that the time required to find a fix with the Iter-T test suite is reduced by 42.5% (95% CI $[38.0\%, 46.8\%]$, $p < 0.001$) compared to the provided suite.

Fig. 3 shows the number of cases fixed by Iter-T as a function of the size of the corresponding test suite. For example, Iter-T fixed 627 cases using a test suite containing only 1 test. Adding more tests increases the number of bugs repaired, but the gain diminishes rapidly, with 97% of the fixes found using up to 6 tests.

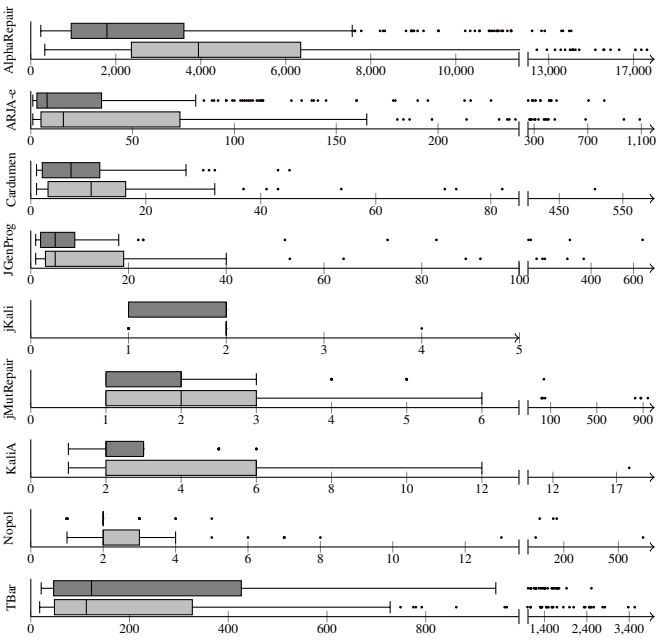

Fig. 2. Time in seconds to generate a fix by TB-APR tool: benchmark test suite (bottom/lightgray) vs. Iter-T generated test suite (top/gray)

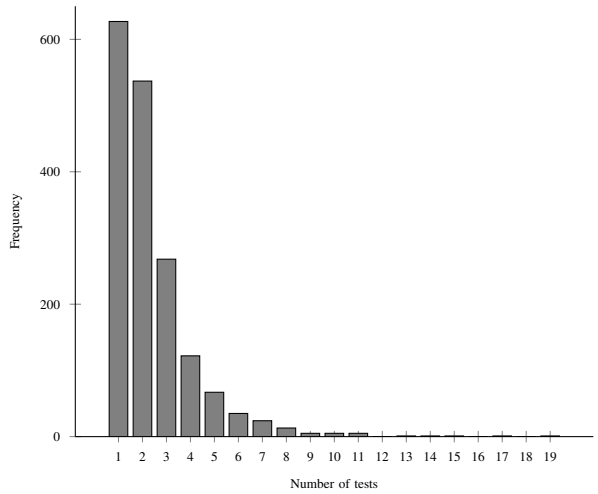

Fig. 3. Number of fixes found with Iter-T by number of tests

RQ3 asks if the overhead added by Iter-T makes its use unappealing. Given a buggy method, recall that Iter-T carries out an iterative process that extends an initially empty test suite one test at a time until the TB-APR tool (using a test suite T_{final}) produces a patch that is deemed correct by the TACO verification tool.

Table III reports the total time on average to fix a bug. Unlike Fig. 2, we now include all the overhead incurred by calls to TACO to build the suite and verify the fix and all the calls to the TB-APR tools, including the ones that produced the

fix. Table III, like Fig. 2, is limited to the 1249 cases in which both suites lead the TB-APR tools to a fix. Remarkably, the overhead incurred by Iter-T when running TACO or making intermediate calls to the repair tool is compensated by the lower program repair times (See Fig. 2), to the point that Iter-T is near 0.

On the other hand, we compare the time Iter-T took to repair a bug (including all the overhead) versus the time the TB-APR tool took to return a negative result (all outcomes except Fix and Error) using the provided suite in Table IV. Interestingly, Iter-T requires almost the same time and ends up providing a correct patch. Recall that verifying a fix usually takes longer since the underlying decision procedure must analyze the whole search space. This time, again, is compensated by the lower repair times (See Fig. 2). That is, not only is the quality of the produced patch better, but it is also generated efficiently.

Some experiments may reach the timeout, which was set to one hour for all tools, except for AlphaRepair, for which we used a five-hour timeout following the authors' configuration [30]. Test suites generated by Iter-T yield 16% fewer timeouts. Whenever a TB-APR tool cannot find a patch, Iter-T adds another test (up to 3 per iteration) to the suite and retries the repair with the enhanced test suite. If we consider all intermediate patches despite the final result of Iter-T, 95.8% of them were found after adding only one test. 3.9% after adding 2 tests and 0.3% after adding 3 tests.

TABLE III
AVERAGE TIME TO FIX IN SECONDS BY CLASS: BENCHMARK TEST SUITE VS. ITER-T GENERATED TEST SUITE

Class	**Benchmark**	**Iter-T** *(Diff)*
Absolute	19	26 *(+135.65%)*
AddLoop	1563	1248 *(-20.1%)*
Alphabet	172	337 *(+96.2%)*
BankAccount	113	214 *(+89.0%)*
BinarySearch	152	116 *(-24.0%)*
BubbleSort	190	206 *(+8.6%)*
Calculator	288	268 *(-7.0%)*
CombinationPermutation	16607	17433 *(+5.0%)*
CopyArray	243	140 *(-42.6%)*
Factorial	14	22 *(+64.8%)*
Fibonacci	190	398 *(+109.7%)*
Find_First_In_Sorted	282	71 *(-74.9%)*
Find_In_Sorted	57	100 *(+74.5%)*
FindFirstZero	342	101 *(-70.6%)*
FindInArray	151	91 *(-39.6%)*
GCD	1193	486 *(-59.3%)*
IC_digits	53	58 *(+8.1%)*
IC_grade	34	63 *(+82.2%)*
IC_median	53	93 *(+75.7%)*
IC_smallest	214	338 *(+58.0%)*
IC_syllables	24	58 *(+145.0%)*
Inverse	105	39 *(-63.4%)*
LCM	766	756 *(-1.3%)*
LeapYear	18287	18329 *(+0.2%)*
LinearSearch	32	48 *(+50.0%)*
OddEven	13	14 *(+10.0%)*
Perimeter	44	49 *(+10.8%)*
PrimeCheck	195	101 *(-48.4%)*
PrimeNumbers	550	257 *(-53.3%)*
Smallest	219	113 *(-48.3%)*
StackQueue	169	184 *(+9.0%)*
StrPalindrome	26	63 *(+147.2%)*
StudentEnrollment	510	738 *(+44.8%)*
Time	1099	996 *(-9.4%)*
Average	**885**	**884** ***(-0.2%)***

F. Sensitivity Analysis of Verification Parameters

As stated in Section IV-C, the bitwidth and loop unroll parameters must be chosen consistently to produce meaningful CEs. While the selection of these parameters followed a simple rule, namely low but sufficient to ensure that Iter-T produces a bug-revealing test, one might wonder how sensitive the final outcome is to these parameters.

In this section, we present a sensitivity analysis evaluating the final outcome (as defined in Table I) as a function of these parameters. For these experiments, we use TBar as the underlying APR tool for two reasons: *(i)* TBar does not rely on stochastic search or randomized exploration, which eliminates an important confounding factor, and *(ii)* TBar produces the largest number of successful repairs. Since the goal is to study how verification parameters affect the outcome, we focus on cases where the APR tool is able to generate a patch. We therefore restrict the analysis to bugs for which TBar succeeds when using both the benchmark test suite and the test suite generated by Iter-T, which we use as a proxy for the APR tool's ability to find a fix. Among the bugs that satisfy these criteria, multiple case studies may correspond to the same class. For each class with at least one matching case, we select the case for which TBar requires the largest number of tests to find a fix. This selection strategy allows us to evaluate the impact of verification parameters in the most demanding scenarios for each class, where the ability of Iter-T to generate bug-revealing tests is exercised to the greatest extent possible. As a result, a total of 25 cases are included in this analysis.

Table V presents the final outcome of these experiments as a function of the bitwidth and loop unroll parameters. Some bugs either do not contain loops or require the full Java bitwidth; in these cases, the sensitivity analysis is naturally limited to the remaining parameter. The results are divided into three groups, according to the parameters that were varied. In those cases where it is meaningful to vary both parameters, they were varied together, i.e., (1,1), (2,2), ...

Table V shows that in most cases a fix (✓) is found. Entries marked with "·" correspond either to cases where the corresponding parameter is not applicable i.e., there are constants in the source code that cannot be represented with the corresponding bitwidth, or cases that reached a timeout for previous scopes.

Not surprisingly, for very low parameters TACO becomes unable to find a CE. These cases are marked with **N** (No negative). In some cases, marked with $\mathbf{E^n}$ (Error), TACO does not support some Java constructs produced by TBar. However, we show the actual iteration at which this happened to illustrate that an actual bug-revealing test was generated and enabled TBar to produce a patch. Particularly, $\mathbf{E^0}$ corresponds to cases where the TACO translation capacity was exceeded i.e., no call to TBar was made.

These experiments were not free from overfitting. Such cases are marked with $\mathbf{O^n}$ (Overfit). We also show the iteration at which TACO deemed correct an overfit patch. These cases

TABLE IV
COMPARING THE TIME IN SECONDS ITER-T REQUIRES TO FIX BUGS VERSUS THE TIME WASTED WHEN USING THE PROVIDED TEST SUITE AND FAILING TO FIX THE BUGS

Class	Wasted	Iter-T *(Diff)*
Absolute	1	9.5 *(+849%)*
Alphabet	8864	4615.1 *(-48%)*
BankAccount	67	193.2 *(+190%)*
BinarySearch	2356	1225.9 *(-48%)*
BubbleSort	5	132.7 *(+2554%)*
CombinationPermutation	271	23257.5 *(+8482%)*
Fibonacci	259	282.3 *(+9%)*
Find_First_In_Sorted	10800	2740.5 *(-75%)*
Find_In_Sorted	50	30.3 *(-40%)*
FindInArray	15059	10478 *(-30%)*
GCD	1160	357 *(-69%)*
IC_digits	89	326 *(+265%)*
IC_grade	175	844 *(+382%)*
IC_median	28	228 *(+716%)*
IC_smallest	4	1007 *(+25068%)*
IC_syllables	138	52 *(-62%)*
Inverse	210	395 *(+88%)*
LCM	92	734 *(+698%)*
LinearSearch	2	10 *(+406%)*
Perimeter	9002	4175 *(-54%)*
PrimeCheck	42	141 *(+239%)*
PrimeNumbers	5646	588 *(-90%)*
Smallest	2	11 *(+463%)*
StackQueue	491	235 *(-52%)*
StudentEnrollment	169	558 *(+231%)*
Time	1115	1326 *(+19%)*
Average	**2205**	**1572** ***(-29%)***

correspond to infinite loops not detected by TACO, since we are deliberately truncating loops through the corresponding parameter and loop invariants were removed, as explained in Section III-F.

A one-hour timeout was set for TACO on these experiments. Entries marked with ⏲✓ correspond to cases where TACO exhausted the allotted time verifying an actual fix. On the other hand, ⏲, means a timeout occurred on the first iteration.

Interestingly, for the bug on class `PrimeNumbers`, an overfit patch was generated at scope 4 (which means that TACO was able to find a CE). One might expect that, by increasing both verification parameters by one, TACO would be able to find at least the same CE. However, for scopes 5 and 6 TACO found no CE, and ended up exhausting the allotted time at scope 7. For scope 4, an input that produced an overflow with a restricted bitwidth was found. This input also exposed the bug when translated to Java (without bitwidth restrictions). However, the same input produced no overflow when analyzed with a higher bitwidth, which resulted in no CE being found.

G. Evaluation of a Binomial Heap Bug

We complement our earlier experiments with an evaluation on a buggy binomial heap implementation. Binomial heaps are a classic yet non-trivial data structure, characterized by complex structural invariants. The bug considered in this experiment is particularly difficult to detect: it manifests only for heaps of size at least seven and for specific key configurations. As a result, standard random or small test suites are unlikely to expose the bug.

TABLE V
OUTCOME BY BITWIDTH AND LOOP UNROLL PARAMETERS

	Class	Scope 1	2	3	4	5	6	7	8
Bitwidth	BankAccount	·	·	·	·	✓	✓	✓	✓
	Calculator	·	·	·	·	·	·	✓	✓
	Perimeter	·	·	·	✓	✓	✓	✓	✓
	Time	·	·	·	·	·	·	✓	✓
Loop unroll	AddLoop	✓	✓	✓	✓	E^3	⏲✓	·	·
	CombinationPermutation	✓	✓	✓	✓	✓	⏲✓	·	·
	BubbleSort	✓	✓	✓	✓	✓	⏲✓	·	·
	BinarySearch	✓	✓	✓	O^1	✓	O^1	✓	O^1
	CopyArray	N	✓	✓	✓	✓	O^1	✓	✓
	Factorial	✓	✓	✓	✓	✓	✓	✓	✓
	Find_First_In_Sorted	✓	✓	✓	✓	✓	✓	✓	✓
	Find_In_Sorted	✓	O^2	✓	✓	✓	✓	✓	✓
	FindFirstZero	E^1	E^2	E^1	✓	✓	✓	✓	✓
	FindInArray	✓	✓	✓	✓	✓	✓	✓	✓
	IC_syllables	N	✓	✓	✓	✓	✓	✓	✓
	Inverse	✓	✓	✓	✓	✓	✓	✓	✓
	LinearSearch	✓	✓	✓	✓	✓	✓	✓	✓
	Smallest	✓	✓	✓	✓	✓	✓	✓	✓
	StrPalindrome	✓	✓	✓	✓	✓	✓	✓	✓
	GCD	N	✓	✓	✓	✓	✓	✓	✓
	LCM	✓	✓	✓	✓	✓	✓	✓	✓
Both	IC_digits	·	·	·	·	✓	✓	✓	E^0
	PrimeCheck	·	·	✓	✓	✓	✓	✓	✓
	PrimeNumbers	·	·	·	O^2	N	N	⏲	·
	StackQueue	·	·	·	✓	✓	✓	✓	E^0

✓: Fix; ⏲ / ⏲✓: Timeout finding a CE / checking a fix; N: No negative; E^n / O^n: Error / Overfit on iteration n.

Table VI presents the overall results of these experiments. The times, like those reported in Table III, include all the overhead incurred by calls to TACO to build the test suite and verify the fix, as well as the calls to the TB-APR tools.

As a baseline test suite, we generated 51 inputs (binomial heaps) with sizes ranging from 0 to 50 using the `insert` method with random keys. The tests call the `extractMin` method on these inputs and assert both the invariant of the resulting data structure and the returned value. However, this test suite contained no bug-revealing test, a common requirement to run TB-APR tools, so we added one of the inputs generated by Iter-T in the first iteration that successfully exposed the bug (see Fig. 4).

The only fix generated was obtained using the test suite generated with Iter-T, which consists of only 3 tests, with Cardumen as the underlying TB-APR tool. The time reported in Table V for this experiment includes the verification of the fix (1982 seconds), as well as the calls to Cardumen to generate intermediate patches and TACO to build the test suite (183 seconds). The only patch generated with the benchmark test suite, found by JMutRepair, was immediately discarded by TACO after 9 seconds of analysis using a proper bug-revealing test.

AlphaRepair is, by far, the most time-consuming TB-APR tool in this experiment (see Fig. 2). Although it exhausted the five-hour timeout with both test suites, the candidate patches explored up to that point were stored on disk. After inspecting these patches, we identified valid fixes in both experiments.

While no individual call to ARJA-e exhausted the one-hour timeout, the cumulative execution time did; therefore, we classify this experiment as a timeout. Notably, ARJA-e advanced up to iteration 5, generating five overfitting patches

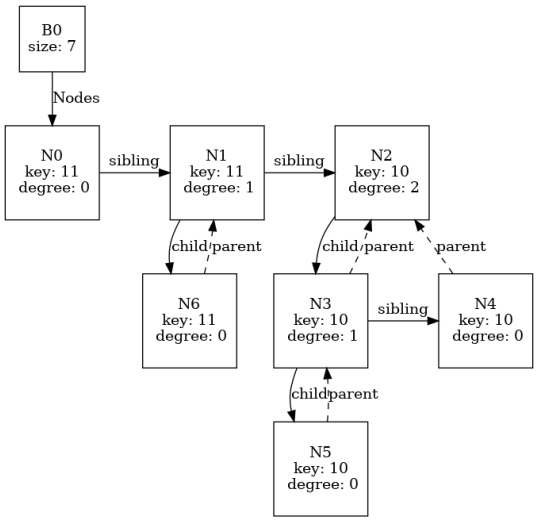

Fig. 4. Bug-revealing binomial heap

for which Iter-T was able to provide bug-revealing tests. In the final iteration, after adding 3 bug-revealing tests, ARJA-e did not find any patches.

TABLE VI
EVALUATION OF BINOMIAL HEAP BUG BY TB-APR: BENCHMARK TEST SUITE VS. ITER-T GENERATED TEST SUITE

TB-APR tool	Benchmark		Iter-T			
	Result	Time	Result	#Tests	#It.	Time
AlphaRepair	Timeout	-	Timeout	1	0	-
ARJA-e	Timeout	-	Timeout	11	5	-
Cardumen	No patch	88	**Fix**	3	2	2165
JGenProg	No patch	1192	No patch	3	0	1094
JKali	No patch	271	No patch	4	1	551
JMutRepair	**Overfit**	5	No patch	6	3	290
KaliA	No patch	77	No patch	4	1	221
Nopol	No patch	1	No patch	3	0	79
TBar	Timeout	-	Timeout	2	1	-

H. Experimental Setup

All experiments were carried out on a Debian GNU/Linux 12 Docker with 32GB RAM, which runs on a workstation equipped with a 12th Gen Intel(R) Core(TM) i9-12900KF processor and 64GB RAM.

I. Data Availability

The reproducibility package is available at [40]. It contains all tools, experimental data, and detailed instructions on how to setup a Docker container to replicate the experiments.

J. Discussion

Automated test generation techniques such as EvoSuite [41] and Randoop [42] are widely used to improve code coverage and support regression testing. However, their suitability for program repair is limited. These techniques typically generate large test suites, which significantly increase patch validation costs and often cause TB-APR tools to scale poorly or time out.

Moreover, coverage-driven test generation lacks the precision provided by formal specifications. As a result, such tests often miss subtle or boundary-case bugs that are explicitly captured by contracts. Recent work [43] empirically supports this limitation: more than 50% of EvoSuite-generated test suites failed to detect existing bugs, and complete repairs using ARJA-e were achieved in only 2.3% of incorrect submissions. These results highlight that increased coverage does not necessarily translate into effective bug exposure or reduced overfitting.

Iter-T currently relies on manually provided JML contracts, reflecting a limitation shared by most specification-based approaches. Importantly, Iter-T does not require full specifications: partial contracts, similar to those used in property-based testing, are sufficient to guide CE generation. While well-specified contracts are still uncommon in practice, recent advances in large language models have demonstrated that inferring formal specifications from comments or documentation is increasingly feasible [44].

At the same time, while LLM-based repair tools have shown strong performance on small and relatively simple programs, their use often introduces a new challenge: the generation of multiple plausible patches rather than a single definitive fix. Iter-T addresses this issue by iteratively filtering candidate patches, providing a principled mechanism for patch selection that is complementary to LLM-based repair approaches.

K. Threats to Validity

While our experimental evaluation supports the conclusion that test suites generated with Iter-T are of high quality for TB-APR, there are several threats to the validity of these results that warrant explicit discussion.

1) Internal Validity: To ensure a fair comparison and mitigate confounding variables, we maintained a strict consistency in tool configuration. When evaluating the nine TB-APR tools, the only variable modified was the test suite provided to them (the benchmark test suite versus the one generated by Iter-T).

Another potential threat is the human factor in evaluating patch correctness. While TACO automatically deems patches as correct based on bounded verification, we manually verified these patches to ensure they were true fixes. This manual process is inherently error-prone, however, we conducted the inspections carefully to mitigate this risk.

Furthermore, we justified our choice of algorithmic cutoffs, specifically the maximum of three tests per iteration, execution timeouts, and bounded verification parameters. The 3-test limit is supported by a clear saturation trend in our data: the frequency of patches found dropped from 95.8% to 3.9% and 0.3% as the number of added tests progressed from 1 to 3. This sharp decline demonstrates that increasing the cutoff would be meaningless. Regarding timeouts, they occurred in less than 1.7% of our experimental runs. Given this low frequency, we consider the impact of timeouts on our overall conclusions to be negligible, representing a necessary trade-off for practical experiment termination. Regarding the verification bounds, these were kept small enough to ensure the feasibility of the process while remaining sufficient to expose the bugs. These limits were applied consistently across all tools to prevent bias.

2) External Validity: Several factors may limit the generalizability of our findings. Our experiments were limited to the IntroclassJava and BuggyJavaJML benchmarks. Although these provide a wide variety of both seeded and real bugs, they may not fully represent the complexity of large-scale codebases or specific bug classes.

We evaluated Iter-T using nine representative TB-APR tools covering several repair paradigms: genetic/evolutionary, constraint-based, large language models, template-based, and simple mutation/replacement. While this broad selection is representative of the state-of-the-art in TB-APR and mitigates the risk of results being biased toward a specific repair algorithm, the observed behavior might still differ when applied to radically different program repair paradigms.

While the technique implemented by Iter-T is theoretically agnostic of the specific language or verification tool, the current implementation is bound to the Java/JML ecosystem and the TACO verifier. Its effectiveness in other environments (e.g., for C using CBMC or Rust using Kani) remains to be empirically validated to assess its generality fully.

3) Construct Validity: This threat concerns whether the experimental measures accurately capture the intended concepts. We operationalize "repair effectiveness" exclusively as the ability to generate functionally correct patches and to expose overfitting via bug-revealing tests. We recognize this as a case of construct underrepresentation, as our metrics do not capture other qualitative aspects of repair, such as code readability, maintainability, or changes in complexity. However, this choice allows for an objective evaluation based on precise correctness specifications as an explicit oracle, ensuring that "effectiveness" is measured strictly in terms of functional correctness.

4) Conclusion Validity: To mitigate these threats, we conducted a large-scale empirical evaluation comprising 12906 experiments, resulting in a total of 1411 and 1713 successful repairs for the baseline and Iter-T techniques, respectively. This extensive sample reduces the risk that observed differences are driven by chance or outliers. Furthermore, to account for structured variability in the data, we employed mixed-effects models when analyzing both repair success and repair time, incorporating random effects for classes, bugs, and APR tools. These analyzes confirm that the observed improvements are statistically significant and robust, and are not driven by any particular tool, benchmark, or defect category.

Finally, to mitigate the risk that conclusions could be distorted by the bounded nature of automated verification, all patches deemed correct by TACO were manually inspected, ensuring that our conclusions regarding the effectiveness of Iter-T are not inflated by false negatives.

V. Related Work

Program repair is a fruitful field with many advanced techniques and tools. In Section IV we have looked at nine representative repair tools, including GenProg, Kali, and Nopol, which use a variety of techniques, such as heuristics, syntactic-based and SMT-based solving. As mentioned throughout the paper, the overfitting problem has long been recognized in program repair. Moreover, the problem of evaluating repair results is further challenging because many research projects do not release their patches and correctness results, and researchers are encouraged to make their repair results publicly available (e.g., see [45], [46]). Typically, patch correctness is assessed through a test suite (e.g., an independent test suite [46]), by manual inspection, or by comparing patches to a ground truth [31], [47]. These approaches have limitations, e.g., human bias when using manual validation, as well as the ineffectiveness of additional tests. In Yang et al. [48], the authors revisit Patch-Correctness Checking (PCC) techniques using 1,988 manually labeled plausible patches for the Defect4J [49] dataset. The study reevaluates nine state-of-the-art PCC methods, focusing primarily on oracle-absent approaches such as static analysis, dynamic testing, and naturalness-based metrics. Among its findings, the authors report that common assumptions, such as correct patches being syntactically closer to the original buggy code, do not hold in many cases. Additionally, learning-based techniques are shown to overfit, while dynamic approaches, although generally reliable, degrade on complex patches. Notably, naturalness-based techniques outperform traditional static methods. In Tian et al. [50], the authors also predict patch correctness with an oracle-absent approach by combining deep learning-based code embeddings with manually engineered features. They construct a manually labeled dataset of 2,147 patches drawn from both APR-generated and human-written sources. Their findings demonstrate that embedding-based models capture valuable semantic information, but combining them with structured, human-understandable features leads to significantly higher accuracy and robustness. Recent work by Le-Cong et al. [51] introduced INVALIDATOR, a technique for automated patch correctness assessment that addresses the persistent challenge of overfitting in APR. Unlike traditional approaches relying on test suite augmentation or costly manual inspection, INVALIDATOR combines semantic reasoning based on dynamically inferred program invariants with syntactic reasoning derived from pre-trained language models. This hybrid strategy allows the system to detect patches that either violate correct specifications or preserve erroneous behaviors, even when additional test generation fails. Their evaluation on Defects4J demonstrated that INVALIDATOR substantially outperforms prior state-of-the-art methods, highlighting the effectiveness of integrating semantic and syntactic perspectives in patch correctness assessment. In this work, we focus on automatic testing using formal specifications and bounded verification, which not only allow for automatic validation of repair candidates but also help us to create good "corner case" tests that can effectively guide the repair process. Interestingly, while the use of formal specifications and bounded verification is limited in traditional program repair (e.g., due to lack of specifications [9]), this type of repair has been shown to be natural and effective for debugging declarative languages (e.g. Alloy) [52], [53]. PROPR is a technique that also attempts to overcome overfitting by profiting from specifications [5]. It differs from ours in that it uses properties, in the sense of property-based testing, rather than contracts. Although properties are expressed as contract-like assertions, they generally apply to

parameterized tests (as metamorphic properties), instead of capturing the intended behavior of methods and classes, as contracts do.

In Nilizadeh et al. [10], the authors present the cornerstone on which Iter-T is built. It introduces the idea of using a CE in order to build a single test that is added to the provided test suite. The article shows that it is enough to reduce overfitting. In this article, we go further by (i) presenting a technique that doesn't require existing test suites, (ii) iterating this process and (iii) providing evidence of the effectiveness of test suites composed solely of verification CEs of plausible patches, as finely tailored suites for APR. These test suites are significantly smaller, which in turn reduces the time required to find a fix. In Yang et al. [6], the authors tackle the same problem we focus on in this paper: reducing overfitting by generating better test suites. The authors propose Opad, a framework for the detection of overfitted patches that uses fuzzing and automatically generates crash and memory-safety oracles. When compared to Iter-T, Opad scales to substantially larger programs due to the use of fuzzing, but the automatically generated oracles are significantly weaker than the contracts used in Iter-T. Also, Opad only detects overfitted patches so that the underlying APR tool can continue traversing the patches search space, but does not enrich the developer-provided test suite. Actually, Opad could benefit by following the Iter-T technique and enriching the provided test suite in order to help the APR tool.

Since we are already using specifications, the question of how Iter-T compares to specification-based repair tools immediately comes to mind. Gopinath et al. [54] do contract-based program repair using SAT. They assume fault location is known and only consider a limited set of fix candidates (expressions of the form $v.f_1.\cdots.f_n$). This work does not fall within the realm of TB-APR, yet it can benefit from our work: Iter-T can contribute a test suite to be used during fault localization, and the extra cost of requiring a contract can be amortized during fault localization and posterior repair using the approach from [54]. SpeciFix [55] also uses specifications. Like Iter-T, it fixes programs equipped with contracts. Unlike Iter-T, which modifies the source code with the aid of the underlying TB-APR tool, SpeciFix modifies the contracts (i.e., a program is considered correct if it complies with its contract, and compliance is achieved by SpeciFix by modifying the latter). A similar approach is followed in Abreu et al. [56] for Dafny [57] programs.

Le et al. [58] leverage formal specifications for APR by combining genetic programming with modular deductive verification. They use CEs to localize faults and extract violated sub-formulas from the specification, which directly guide patch synthesis, following the intuition that concrete code extracted from the specification would help in patching the program. Similar to Iter-T, they rely on specifications as the ultimate correctness oracle. In contrast, Iter-T can be seen as a hybrid approach: specifications are not directly consumed by the repair tools but are instead used to generate fault-revealing tests that refine a test suite. This test suite can then be used by a wide range of mature, off-the-shelf APR tools without modifying their internals. An additional advantage of this approach is that Iter-T produces a reusable artifact, a synthesized regression test suite, alongside the patch itself.

VI. Conclusions

We have presented a technique supported by Iter-T, that allows generating test suites automatically that are of good quality for test-based automated program repair. The generated test suites are built out of tests synthesized from CEs produced while verifying overfitted patches with the aid of a bounded verification tool. These test suites are small and allow one to fix more bugs using less time compared to test-based program repair from developer-generated test suites. The presented technique requires that the methods to be fixed contain precise specifications. We are aware that writing such specifications is a task that requires qualified developers. This may be perceived as a limitation, but at the same time, these specifications enable the generation of test suites that correctly repair substantially more bugs in less time.

Regarding further work, in Section III-D we mentioned we believed the particular structure of the test synthesized by Iter-T might be related to its good performance. We will assess this hypothesis. Moreover, new, more extensive benchmarks will be used in the experimental evaluation of Iter-T.

References

[1] K. Beck, *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] N. J. Wahl, "An overview of regression testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 1, pp. 69–73, 1999. [Online]. Available: https://doi.org/10.1145/308769.308790

[3] X. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, p. 163. [Online]. Available: https://doi.org/10.1145/3180155.3182536

[4] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[5] M. P. Gissurarson, L. Applis, A. Panichella, A. van Deursen, and D. Sands, "PROPR: property-based automatic program repair," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1768–1780. [Online]. Available: https://doi.org/10.1145/3510003.3510620

[6] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 831–841. [Online]. Available: https://doi.org/10.1145/3106237.3106274

[7] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 24–36. [Online]. Available: https://doi.org/10.1145/2771783.2771791

[8] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, M. Odersky and P. Wadler, Eds. ACM, 2000, pp. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.104

[10] A. Nilizadeh, M. Calvo, G. T. Leavens, and X. D. Le, "More reliable test suites for dynamic APR by using counterexamples," in *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, Z. Jin, X. Li, J. Xiang, L. Mariani, T. Liu, X. Yu, and N. Ivaki, Eds. IEEE, 2021, pp. 208–219. [Online]. Available: https://doi.org/10.1109/ISSRE52982.2021.00032

[11] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, 2005. [Online]. Available: https://doi.org/10.1007/s10009-004-0167-4

[12] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser, "JFIX: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 376–379. [Online]. Available: https://doi.org/10.1145/3092703.3098225

[13] R. S. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 390–405. [Online]. Available: https://doi.org/10.1145/3453483.3454051

[14] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias, "TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds," *IEEE Trans. Software Eng.*, vol. 39, no. 9, pp. 1283–1307, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2013.15

[15] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_26

[16] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, F. Rothlauf, Ed. ACM, 2009, pp. 947–954. [Online]. Available: https://doi.org/10.1145/1569901.1570031

[17] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 5:1–5:53, 2020. [Online]. Available: https://doi.org/10.1145/3360004

[18] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Software Eng.*, vol. 43, no. 1, pp. 34–55, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

[19] A. Nilizadeh and G. T. Leavens, "Be realistic: automated program repair is a combination of undecidable problems," in *Proceedings of the Third International Workshop on Automated Program Repair*, ser. APR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 31–32. [Online]. Available: https://doi.org/10.1145/3524459.3527346

[20] D. R. Cok, "Openjml: JML for java 7 by extending openjdk," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 472–479. [Online]. Available: https://doi.org/10.1007/978-3-642-20398-5_35

[21] A. Nilizadeh, M. Calvo, G. T. Leavens, and D. R. Cok, "Generating counterexamples in the form of unit tests from hoare-style verification attempts," in *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022*, A. Hartmanns, I. Schaefer, S. Gnesi, and N. Plat, Eds. ACM, 2022, pp. 124–128. [Online]. Available: https://doi.org/10.1145/3524482.3527656

[22] A. Nilizadeh, G. T. Leavens, X. D. Le, C. S. Pasareanu, and D. R. Cok, "Exploring true test overfitting in dynamic automated program repair using formal methods," in *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 2021, pp. 229–240. [Online]. Available: https://doi.org/10.1109/ICST49551.2021.00033

[23] T. Durieux and M. Monperrus, "Introclassjava: A benchmark of 297 small and buggy java programs," Universite Lille 1, Tech. Rep. hal-01272126, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01272126/document

[24] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707

[25] R. Kersten, K. S. Luckow, and C. S. Pasareanu, "POSTER: afl-based fuzzing for java with kelinci," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2511–2513. [Online]. Available: https://doi.org/10.1145/3133956.3138820

[26] L. Zemín, S. G. Brida, A. Godio, C. Cornejo, R. Degiovanni, G. Regis, N. Aguirre, and M. F. Frias, "An analysis of the suitability of test-based patch acceptance criteria," in *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. IEEE, 2017, pp. 14–20. [Online]. Available: https://doi.org/10.1109/SBST.2017.12

[27] M. Fähndrich, M. Barnett, D. Leijen, and F. Logozzo, "Integrating a set of contract checking tools into visual studio," in *Proceedings of the Second International Workshop on Developing Tools as Plug-Ins, TOPI 2012, Zurich, Switzerland, June 3, 2012*, D. Garbervetsky and S. Kim, Eds. IEEE Computer Society, 2012, pp. 43–48. [Online]. Available: https://doi.org/10.1109/TOPI.2012.6229809

[28] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153. [Online]. Available: https://doi.org/10.1007/978-3-540-79124-9_10

[29] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[30] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 959–971. [Online]. Available: https://doi.org/10.1145/3540250.3549101

[31] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[32] M. Martinez and M. Monperrus, "ASTOR: a program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 441–444. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[33] Y. Yuan and W. Banzhaf, "ARJA: automated repair of java programs via multi-objective genetic programming," *IEEE Trans. Software Eng.*, vol. 46, no. 10, pp. 1040–1067, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2874648

[34] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, ser. Lecture Notes in Computer Science, T. E. Colanzi and P. McMinn, Eds., vol. 11036. Springer, 2018, pp. 65–86. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9_3

[35] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[36] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program

repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 172–184. [Online]. Available: https://doi.org/10.1145/3611643.3616271

[37] W. Wang, Y. Wang, S. Joty, and S. C. H. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 146–158. [Online]. Available: https://doi.org/10.1145/3611643.3616256

[38] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," *CoRR*, vol. abs/2303.10494, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.10494

[39] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *CoRR*, vol. abs/2304.00385, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.00385

[40] Replication package. [Online]. Available: https://mega.nz/folder/K5okzbwR#H87sWI179p4dNYOqqex1IQ

[41] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[42] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[43] R. Gu, J. M. Rojas, and D. Shin, "Can test generation and program repair inform automated assessment of programming projects?" in *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 2025, pp. 699–710. [Online]. Available: https://doi.org/10.1109/ICST62969.2025.10988955

[44] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee, "How effective are large language models in generating software specifications?" in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2025, Montreal, QC, Canada, March 4-7, 2025*. IEEE, 2025, pp. 1–12. [Online]. Available: https://doi.org/10.1109/SANER64311.2025.00014

[45] M. Monperrus, "A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

[46] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.

[47] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.

[48] J. Yang, Y. Wang, Y. Lou, M. Wen, and L. Zhang, "A large-scale empirical review of patch correctness checking approaches," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1203–1215. [Online]. Available: https://doi.org/10.1145/3611643.3616331

[49] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[50] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 92:1–92:34, 2023. [Online]. Available: https://doi.org/10.1145/3576039

[51] T. Le-Cong, D. Luong, X. D. Le, D. Lo, N. Tran, B. Q. Huy, and Q. Huynh, "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *IEEE Trans. Software Eng.*, vol. 49, no. 6, pp. 3411–3429, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3255177

[52] S. Gutiérrez Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. Frias, "Icebar: Feedback-driven iterative repair of alloy specifications," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[53] G. Zheng, T. Nguyen, S. G. Brida, G. Regis, N. Aguirre, M. F. Frias, and H. Bagheri, "Atr: template-based repair for alloy specifications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 666–677.

[54] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *Tools and Algorithms for the Construction and Analysis of Systems*, P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 173–188.

[55] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer, "Automatic program repair by fixing contracts," in *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, ser. Lecture Notes in Computer Science, S. Gnesi and A. Rensink, Eds., vol. 8411. Springer, 2014, pp. 246–260. [Online]. Available: https://doi.org/10.1007/978-3-642-54804-8_17

[56] A. Abreu, N. Macedo, and A. Mendes, "Exploring automatic specification repair in dafny programs," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023 - Workshops, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 105–112. [Online]. Available: https://doi.org/10.1109/ASEW60602.2023.00019

[57] K. R. M. Leino, "Accessible software verification with dafny," *IEEE Software*, vol. 34, no. 6, pp. 94–97, 2017.

[58] X. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 2016, pp. 428–432. [Online]. Available: https://doi.org/10.1109/ICSME.2016.66