An Analysis of the Suitability of Test-based Patch Acceptance Criteria

***Abstract*—Automated program repair techniques attempt to fix programs by looking for patches within a search space of fix candidates. These techniques require a specification of the program to be repaired, used as an acceptance criterion for fix candidates, that many times also plays an important role in guiding some search processes. Most tools use *tests as specifications* for this task. This constitutes a risk, since the incompleteness of tests as specifications may lead one to obtain spurious repairs, that pass all tests but are in fact incorrect. This problem has been identified by various researchers, raising concerns on the quality and validity of program fixes. Still, more thorough studies have been proposed using different sets of tests for fix validation, and resorting to manual inspections, showing that while tools reduce their program fixing rate, they are still able to repair a significant number of cases.**

In this paper, we perform a different analysis of the suitability of tests as acceptance criteria for automated program fixes, by checking patches produced by automated repair tools using a bug-finding tool, as opposed to previous works that used tests or manual inspections. We develop a number of experiments in which faulty programs from a known benchmark are fed to the program repair tools GenProg, Angelix, AutoFix and Nopol, using test suites of varying quality and extension, including those accompanying the benchmark. We then check the produced patches against formal specifications using a bug-finding tool. Our results show that, in general, automated program repair tools are significantly more likely to accept a spurious program fix than producing an actual one, in the studied scenarios.

I. Introduction

The significant advances in automated analysis have led to the development of powerful tools to assist software engineers in software development, greatly contributing to software quality. Indeed, tools based on model checking, constraint solving, evolutionary computation and other automated approaches, are being successfully applied to various aspects of software development, from requirements specification to verification and bug finding. Despite the great effort that is put in software development to detect software problems, in particular through the use of the above mentioned techniques, many bugs reach and make it through the deployment phases. This makes effective software maintenance greatly relevant to the quality of the software that is produced. Thus, the traditional emphasis of software analysis techniques, that concentrated in detecting the existence of defects in software and specifications, has recently started to broaden up to be applied to automatically *repair* software [1], [4], [12], [24].

While the idea of automatic program repair is certainly appealing, automatically fixing arbitrary program defects is known to be infeasible. Firstly, many techniques for automatically repairing programs need to produce repair candidates, often consisting of syntactical modifications on the original program. Clearly, *all* program repair candidates cannot be exhaustively considered, and thus the search space of repair candidates to consider needs to be somehow limited. Secondly, for every repair candidate, checking whether the produced candidate constitutes indeed a repair is an undecidable problem on its own, and solving it fully automatically is then necessarily incomplete. Moreover, this latter problem requires a specification of the expected behavior of the program to be fixed that can be subject to automated analysis, if one wants the whole repair process to remain automatic. Most automated repair techniques use *partial* specifications, given in terms of a validation test suite, that in some cases is also used to guide the search for patches.

There is a risk in using tests as specifications, since as it is well known, their incompleteness makes it possible to obtain *spurious* repairs, i.e., programs that seem to solve the problems of faulty code, but are incorrect despite the fact that the validation suite is not able to expose such incorrectness. Nevertheless, various tools report significant success in repairing code using tests as criteria for accepting patches [8]. More recently, various researchers have observed that automatically produced patches are likely to *overfit* test suites used for their validation, leading tools to produce invalid program fixes [20], [15]. Then, further checks have been performed, to analyze more precisely the quality of automatically produced patches, and consequently the ability of automated program repair techniques in producing actual fixes. However, these further checks have usually been performed through manual inspection, or using extended alternative test suites, leaving room for still undetected flaws.

In this paper, we perform a study of the suitability of tests as acceptance criteria for automated program fixes, by checking patches produced by automated repair tools using a bug-finding tool, as opposed to previous works that used tests or manual inspections. We develop a number of experiments using *IntroClass*, a known benchmark for program repair techniques. Faulty programs from this benchmark are used to feed three search-based and one semantics-based program repair tools, using test suites of varying quality and extension, including those accompanying the benchmark. Produced patches are then complemented with corresponding formal specifications, given as pre- and post-conditions, and checked using Pex [22], an automated test generation tool that attempts to exhaustively cover bounded symbolic paths for the patches. Our results show that, in general, automated program repair tools are significantly more likely to accept a spurious program

fix than producing an actual one, in the studied scenarios.

II. Automatic Program Repair

Automated program repair techniques aim at fixing faulty programs through the application of transformations that modify the program's code. Search-based techniques for automated program repair receive a faulty program to repair, a specification of the program's expected behavior, and attempt to generate a patch through the application of syntactic transformations on the original program that satisfies the provided specification [1]. Different techniques and tools have been devised for automated program repair, which can be distinguished on various aspects such as the programming language or kind of system they apply to, the syntactic modifications that can be applied to programs (or, similarly, the fault model a tool aims to repair), the process to produce the fix candidates or program patches, how program specifications are captured (and how these are contrasted against fix candidates), and how the explosion of fix candidates is tamed.

A crucial aspect of program repair is how program specifications are captured and provided to the tools. Some approaches, notably some of the initial ones (e.g., [1], [21]), require *formal specifications* in the form of pre- and post-conditions, or logical descriptions provided in some suitable logical formalism. Many of the latest mainstream approaches, however, use *tests as specifications*. These approaches relieve techniques from the requirement of providing a formal specification accompanying the faulty program, arguing that such specifications are costly to produce, and are seldom found in software projects. Tests, on the other hand, are significantly more commonly used as part of development processes, and thus requiring tests is clearly less demanding [25], [9].

The partial nature of tests as specifications immediately leads to validity issues regarding the fixes provided by automated program repair tools, since a program patch may be accepted because it passes *all* tests in the validation suite but still not be a true program fix (there might still be other test cases, not present in the validation suite, for which the program patch fails). This problem, known as *overfitting* [20], has been previously identified by various researchers [20], [15], and several tools are known to produce spurious patches as a result of their program repair techniques. This problem is handled differently by different techniques. Some resign the challenge of producing fixes and aim at producing hints [9]. Others take into account a notion of *quality*, and manually compare the produced patches with fixes provided by human developers or by other tools [15], [17]. Notice that, even after manual inspection, subtle defects may be still present in the repairs, thus leading to accepting a fix that is invalid. We partly study this issue in this paper.

III. Analysis

In this paper we evaluate 4 tools that use tests as their patch acceptance criterion. The evaluation is performed on the *IntroClass* dataset, which is described in detail in Section III-A. The dataset contains student-developed solutions for 6 simple problems. The correctness of the student's solutions (which usually take under 30 LOC), can be evaluated using instructor-prepared test suites. Each of the provided solutions is faulty: at least one test in the corresponding suite fails.

Since our aim is to evaluate the suitability of test-based patch acceptance criteria, we will introduce some terminology that will help us better understand the following sections. Given a faulty routine m, and a test suite T employed as an acceptance criterion for automated program fixing, a tool-synthesized version m' of m that passes all tests in T is called a *patch*. A patch may overfit and be correct with respect to the provided suite, yet be faulty with respect to a different suite, or more precisely, with respect to its actual expected program behavior. We may then have *correct* and *incorrect* patches; a correct patch, i.e., one that meets the program's expected behavior, will be called a *fix*. This gives rise to our first research question.

RQ1: When applying a given program repair tool/technique on a faulty program, how likely is the tool/technique to provide a patch, and if a patch is found, how likely is it to be a proper *fix*?

Patch correctness is typically determined by manual inspection. Since manual inspections are error-prone (in fact, the faulty routines that constitute *introClass* were all manually inspected by their corresponding developers, yet they are faulty), we will resort to automated verification of patches, in order to determine if they are indeed fixes. We will use *concrete/symbolic execution* combined with *constraint solving*, to automatically verify produced patches, against their corresponding specifications captured as *contracts*. More precisely, we will translate patches into C#, and equip these with pre- and post-conditions captured using Code Contracts [7]; we will then search for inputs that violate these assertions via concrete/symbolic execution and SMT solving, using Pex [22].

To assess the above research question, we need to run automatic repair tools on faulty programs. As we mentioned, we consider the *IntroClass* dataset, so whatever conclusion we obtain will, in principle, be tied to this specific dataset, and its characteristics (we further discuss this issue in the threats to validity section). By focusing on this dataset, we will definitely get more certainty regarding the following issues:

- overfitting produced by repair tools on the *IntroClass* dataset, and
- experimental data on the limitations of manual inspections in the context of automated program repair (especially because this benchmark has been used previously to evaluate various program repair tools).

Notice that when a patch that is not a fix is produced, one may rightfully consider the problem being on the quality of the test suite used for patch generation, not necessarily a limitation of test-based acceptance criteria as a whole, or the program fixing technique in particular: by providing more/better tests one may prevent the acceptance of incorrect patches. That is, overfitting may be considered a limitation of the particular test suites rather than a limitation of test-based acceptance criteria. To take into account this issue, for instance, [20]

enriches the test suites provided with the benchmark with white-box tests that guarantee branch coverage of a correct variant of the buggy programs. Then, as shown in [20, Fig. 3], between 40% and 50% of the patches that are produced with the original suite, are discarded when the additional white-box suite ensuring branch coverage is considered. Yet the analysis does not address the following two issues:

- Are the patches passing the additional white-box tests indeed fixes? And, equally important,
- Would the tool reject more patches by choosing larger suites?

This leads to our second research question:

RQ2: How does overfitting relate to the thoroughness of the validation test suites, in program repair techniques?

Thoroughness can be defined in many ways, typically via testing criteria. Given the vast amount of testing criteria, an exhaustive analysis is infeasible. Our approach will be to enrich the validation test suites, those provided with the dataset, by adding bounded-exhaustive suites for different bounds. The rationale here is to attempt to be as thorough as possible, to avoid overfitting. For each case study, we obtain suites with approximately 100 tests, and with approximately 1,000 tests (with two different bounds), for each routine. These suites can then be assessed according to measures for different testing criteria. Notice that, as the size of test suites is increased, some tools and techniques may see their performance affected.

A. *The* IntroClass *Dataset*

The *IntroClass* benchmark is thoroughly discussed in [13]. It contains student-developed C programs for solving 6 simple problems (that we will describe below) as well as instructor-provided test suites to assess their correctness. *IntroClass* has been used to evaluate a number of automated repair tools [11], [19], [20], and its simplicity reduces the requirements on tool scalability. The benchmark is composed of the following:

- **Checksum**: Given an input string $S = c_0, \ldots c_k$, this method computes a checksum character c following the formula $c = \left(\sum_{0 \leq i < S.length()} S.charAt(i)\right) \% \ 64 \ + \ ' \ '$.
- **Digits**: Convert an input integer number into a string holding the number's digits.
- **Grade**: Receives 5 floats f_1, f_2, f_3, f_4 and $score$ as inputs. The first four are given in decreasing order ($f_1 > f_2 > f_3 > f_4$). These 4 values induce 5 intervals $(\infty, f_1]$, $(f_1, f_2]$, $(f_2, f_3]$, $(f_3, f_4]$, and $(f_4, -\infty]$. A grade A, B, C, D or F is returned according to the interval $score$ belongs to.
- **Median**: Compute the median among 3 integer input values.
- **Smallest**: Compute the smallest value among 4 integer input values.
- **Syllables**: Compute the number of syllables into which an input string can be split according to English grammar (vowels 'a', 'e', 'i', 'o' and 'u', as well as the character 'y', are considered as syllable dividers).

There are two versions of the dataset, the original one described in [13], whose methods are given in C, and a Java translation of the original dataset described in [5]. Some of the programs that result of the translation from C to Java were not syntactically correct and consequently did not compile. Other programs saw significant changes in their behavior. Interestingly, for some programs, the transformation itself repaired the bug (the C program fails on some inputs, but the Java version is correct). The latter situation is mostly due to the different behavior of the non-initialized variables in C versus Java [5]. These abnormal cases were removed from the resulting Java dataset, which thus has fewer methods than the C one.

Because of the automated program repair tools that we evaluate, that include AutoFix, we need to consider yet another version of the *IntroClass* dataset. This new version is the result of translating the original C dataset into Eiffel. For the translation, we employed the C2Eiffel tool [23]; moreover, since AutoFix requires contracts for program fixing, we replaced the input/output sentences in the original *IntroClass*, which received inputs and produced outputs from/to standard input/output, to programs that received inputs as parameters, and produced outputs as return values. We equipped the resulting programs with the correct contracts for pre- and post-conditions of each case study. As in the translation from C to Java, several faulty programs became "correct" as a result of the translation. These cases have to do with default values for variables, as for Java, and with how input is required and output is produced; for instance, faulty cases that reported output values with accompanying messages in lowercase, when they were expected to be upper case, are disregarded since in Eiffel translated programs outputs are produced as return values. Table I describes, for each dataset, the number of faulty versions for each method. The size of their corresponding test suites are only relevant for C and Java, since in the case of AutoFix, tests are automatically produced using AutoTest [14] (the tool does not receive user-provided test suites).

	chcksm	digits	grade	med.	small.	syll.	Total
C	69	236	268	232	177	161	1,143
Java	11	75	89	57	52	13	297
Eiffel	69	236	115	195	166	161	942
Suite size	16	16	18	13	16	16	95

TABLE I
DESCRIPTION OF THE *IntroClass* C, JAVA AND EIFFEL DATASETS.

B. *Experimental Setup*

In this section we will describe the software and hardware infrastructure we employed to run the experiments whose results we will report in Section IV. We also describe the criteria used to generate the bounded-exhaustive test-suites, as well as the automated repair tools we will evaluate and their configurations.

In order to evaluate the subjects from the *IntroClass* dataset we consider, besides the instructor-provided suite delivered within the dataset, two new bounded-exhaustive suites.

Bounded-exhaustive suites contain all the inputs that can be generated within user-provided bounds. We chose bounds so that the resulting suites have approximately 100 tests and 1,000 tests for each method under analysis. This gives origin to two new suites that we will call S100 and S1,000, whose test inputs for each problem are characterized below:

- $S100_{\text{checksum}} = \{c_0, \ldots, c_k \mid 0 \leq k < 4 \wedge \forall_{0 \leq i \leq k} c_i \in \{'a', 'b', 'c'\}\}$ (120 tests).
- $S100_{\text{digits}} = \{k \mid -64 \leq k \leq 63\}$ (128 tests).
- $S100_{\text{grade}} = \{(f_1, \ldots, f_4, score) \mid (\forall_{1 \leq i \leq 4} f_i \in \{30, 40, 50, 60, 70, 80\}) \wedge (f_1 > f_2 > f_3 > f_4) \wedge score \in \{5, 10, 15, 20, \ldots, 90\}\}$ (285 tests).
- $S100_{\text{median}} = \{(k_1, k_2, k_3) | \forall_{1 \leq i \leq 3} -2 \leq k_i \leq 2\}$ (125 tests).
- $S100_{\text{smallest}} = \{(k_1, k_2, k_3, k_4) | \forall_{1 \leq i \leq 4} -2 \leq k_i \leq 1\}$ (256 tests).
- $S100_{\text{syllables}} = \{c_0, \ldots c_k | 0 \leq k < 4 \wedge \forall_{0 \leq i \leq k} c_i \in \{'a', 'b', 'c'\}\}$ (120 tests).
- $S1,000_{checksum} = \{c_0, \ldots, c_k \mid 0 \leq k < 5 \wedge \forall_{0 \leq i \leq k} c_i \in \{'a', 'b', 'c', 'e'\}\}$ (1,364 tests).
- $S1,000_{digits} = \{k \mid -512 \leq k \leq 511\}$ (1,024 tests).
- $S1,000_{grade} = \{(f_1, \ldots, f_4, score) | (\forall_{1 \leq i \leq 4} f_i \in \{10, 20, 30, 40, 50, 60, 70, 80, 90\}) \wedge (f_1 > f_2 > f_3 > f_4) \wedge score \in \{0, 5, 10, 15, 20, \ldots, 100\}\}$ (2,646 tests).
- $S1,000_{median} = \{(k_1, k_2, k_3) | \forall_{1 \leq i \leq 3} -5 \leq k_i \leq 4\}$ (1,000 tests).
- $S1,000_{smallest} = \{(k_1, k_2, k_3, k_4) | \forall_{1 \leq i \leq 4} -3 \leq k_i \leq 2\}$ (1,296 tests).
- $S1,000_{syllables} = \{c_0, \ldots, c_k | 0 \leq k < 5 \wedge \forall_{0 \leq i \leq k} c_i \in \{'a', 'b', 'c', 'e'\}\}$ (1,364 tests).

Notice that from these inputs, actual tests are built using reference implementations of the methods under repair as an oracle. Notice also that all the tests in S100 also belong to S1,000.

Along the experiments we report in this section, we used PCs with Intel(R) Core(TM) i7-2600 CPU, running at 3.40Ghz and holding 8GB of RAM. We used GNU/Linux 3.2.0 as the OS. We set a timeout of 1 hour.

C. Reproducibility

The empirical study we present in this paper involves a large set of different experiments. These involve 3 different datasets (versions of *IntroClass*, as described in the previous section), configurations for 4 different repair tools accross 3 different languages, 3 different sets of tests for the tools that receive test suites. Also, all case studies have been equipped with contracts, translated into C# and verified using Pex. We make available all these elements for the interested reader to reproduce our experiments, which can be found in [removed link due to anonymization]. Instructions to reproduce each experiment are provided therein.

IV. Experimental Results

In this section we present the evaluation of each of the repair tools on the generated suites, and from the collected data we will discuss research questions RQ1–RQ2 in Sections IV-A–IV-B. Tables II–V summarize the experimental data. Notice that the number of versions (#V) of methods from *IntroClass* on which tools are evaluated do not always match. We will discuss the reasons in the following paragraphs.

Angelix does not include all the versions in the dataset due to the following:

- Since Angelix only supports integers and chars as the return type for methods under repair, versions of the digits routine cannot be analyzed. Notice that for the remaining methods, the outputs can be reduced to a single integer (this is clearly the case for median, smallest and syllables), or to a single char (clearly the case for grade and checksum).
- In order to run experiments with Angelix the source code of each variation has to be instrumented and adapted to include calls to some macro functions, and to print the output in a single integer or char. Since in several variants the errors consist on modifications of the input/output Strings, which are stripped-out by the instrumentation, the bugs "fixed themselves". For instance, if the original C variant of smallest produces the output "The smallest number is 5 " (notice the extra space at the end of the string), while the expected output was "The smallest number is 5", the instrumented method will return the number 5 (which removes the discrepancy).
- Since *IntroClass* consists not only of different students implementations but also different commits/versions of the implementation of each student, in several cases the instrumentation resulted in duplicate files.

A similar situation holds for AutoFix. Since string-based input/output via standard input output is replaced by parameters and return values, various bugs that have to do with how input/output is reported also "fixed themselves" as a result of the translation (see previous description of our IntroClass Eiffel dataset).

Method	#V.	Suite	#Patch.	#Fix.	%Patch.	%Fix.
checksum	11	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
digits	75	O	4	0	5.3%	0%
		O ∪ S100	2	0	2.6%	0%
		O ∪ S1,000	2	0	2.6%	0%
grade	89	O	2	2	2.2%	2.2%
		O ∪ S100	2	2	2.2%	2.2%
		O ∪ S1,000	2	2	2.2%	2.2%
median	57	O	11	4	19%	7%
		O ∪ S100	4	4	7%	7%
		O ∪ S1,000	4	4	7%	7%
smallest	52	O	12	0	23%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
syllables	13	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%

TABLE II
Repair statistics for the Nopol automated repair tool.

A. Research Question 1

This research question addresses overfitting, a well-known limitation of automatic program repair approaches that use test suites as the validation mechanism. The use of patch validation techniques based on human inspections or comparisons with developer patches (or even accepting patches as fixes without further discussion), has not allowed the community the identify the whole extent of this problem. For example, paper [10]

Method	#V.	Suite	#Patch.	#Fix.	%Patch.	%Fix.
checksum	69	O	2	0	2.90%	0.00%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
digits	236	O	33	0	13.98%	0.00%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
grade	268	O	4	2	1.49%	0.75%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
median	232	O	124	14	53.45%	6.03%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
smallest	177	O	115	2	64.97%	1.13%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
syllables	161	O	3	0	1.81%	0.00%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%

TABLE III
REPAIR STATISTICS FOR THE GENPROG AUTOMATIC REPAIR TOOL.

Method	#V.	Suite	#Patch.	#Fix.	%Patch.	%Fix.
checksum	69	5 min	4	0	5.79%	0%
		10 min	4	0	5.79%	0%
		20 min	4	0	5.79%	0%
digits	236	5 min	0	0	0%	0%
		10 min	0	0	0%	0%
		20 min	0	0	0%	0%
grade	115	5 min	0	0	0%	0%
		10 min	0	0	0%	0%
		20 min	0	0	0%	0%
median	195	5 min	0	0	0%	0%
		10 min	0	0	0%	0%
		20 min	0	0	0%	0%
smallest	166	5 min	0	0	0%	0%
		10 min	0	0	0%	0%
		20 min	0	0	0%	0%
syllables	161	5 min	0	0	0%	0%
		10 min	0	0	0%	0%
		20 min	0	0	0%	0%

TABLE IV
REPAIR STATISTICS FOR THE AUTOFIX AUTOMATIC REPAIR TOOL.

Method	#V.	Suite	#Patch.	#Fix.	%Patch.	%Fix.
checksum	34	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
grade	47	O	11	10	23.40%	21.27%
		O ∪ S100	11	8	23.40%	17.02%
		O ∪ S1,000	0	0	0%	0%
median	57	O	39	7	68.42%	12.28%
		O ∪ S100	8	6	14.03%	10.52%
		O ∪ S1,000	2	1	3.50%	1.75%
smallest	48	O	31	0	64.58%	0%
		O ∪ S100	9	9	18.75%	18.75%
		O ∪ S1,000	0	0	0%	0%
syllables	46	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%

TABLE V
REPAIR STATISTICS FOR THE ANGELIX AUTOMATIC REPAIR TOOL.

includes the table we reproduce in Fig. 1. The table gives

program	SearchRepair	AE	GenProg	TrpAutoRepair	total
checksum	0	0	8	0	29
digits	0	17	30	19	91
grade	5	2	2	2	226
median	68	58	108	93	168
smallest	73	71	120	119	155
syllables	4	11	19	14	109
total repaired	150	159	287	247	778

Fig. 7: Number of defects repaired by each technique. The total *column* specifies the total number of defects, and the total *row* specifies the total number of repaired defects.

Fig. 1. Performance of GenProg (and other tools) on the IntroClass dataset, as reported in [10].

the erroneous impression that 287 out of 778 bugs were fixed (36.8%). The paper actually analyzes this in more detail and by using independent test suites to validate the generated patches, it claims GenProg's patches pass 68.7% of independent tests, giving the non-expert reader the impression the produced patches were of good quality. Actually, as our experiments reported in Table III show, only 18 out of 1,143 faults were correctly fixed (which gives a fixing ratio of 1.57%), well below the results presented in [10]. Strikingly, of the few fixes produced by GenProg, an important percentage (8 out of 18) fix errors on the Strings storing the inputs and outputs (10 out of 18 repair faults on a program's logic).

We have obtained similar results for the other tools under analysis. Angelix patches 111 faults out of 232 program variants (a ratio of 47.84%), yet only 41 patches are fixes (the ratio reduces to 17.67%). The remaining patches were discarded with the aid of Pex. Nopol patched 29 out of 297 versions (9%), using the evaluation test suite. Upon verification with Pex, the number of fixes is 6 (2%). AutoFix uses contracts (which we provided) in order to automatically (and randomly) generate the evaluation suite. When a patch is produced, AutoFix validates the adequacy of the patch by with a randomly generated suite. AutoFix then produced patches for the great majority of faulty routines, but itself showed that most of these were inadequate, and overall reported only 2 patches (which were invalid fixes).

As previously discussed in the beginning of Section III, these unsatisfactory results might be due to the low quality of the validation test suite. Yet, it is worth emphasizing, that the IntroClass dataset was developed to be used in program repair, and the community has vouched for its quality by publishing the benchmark and using the benchmark in their research.

B. Research Question 2

This research question relates to the impact of more thorough validation suites on overfitting, as well as on the quality of the produced patches. Table II shows that Nopol profits from larger suites in order to reduce overfitting significantly.

It suffices to consider suite O ∪ S100 to substantially reduce overfitting. The number of spurious patches is reduced from 29 to 8, out of which 6 are fixes. Unfortunately, the number of fixes remains low. This shows that Nopol, when fed with a good quality evaluation suite, is able to produce (a few) good quality fixes. GenProg, on the other hand, cannot cope with suites S100 and S1,000, and exceeds the allotted time in all cases (see Table VI). This is an important limitation: small validation suites produce significant overfitting, and large validation suites deem GenProg useless. Further experiments are required in order to determine validation suite size limits within which GenProg may show progress. Angelix (see Table V) sees its overfitting reduced. The reduction can be seen for instance in methods median and smallest, where overfitting is significantly reduced. Unfortunately, this happens at the expense of the effectiveness of Angelix: significantly fewer fixes are produced. Since AutoFix generates the evaluation suites, rather than providing larger suites we extend the test generation time. AutoFix does not have a good performance on this dataset.

	Nopol	GenProg	Angelix
O	8	52	16
O ∪ S100	6	1,143	48
O ∪ S1,000	6	1,143	122

TABLE VI
NUMBER OF TIMEOUTS REACHED BY TOOL AND VALIDATION SUITE.

V. THREATS TO VALIDITY

In this paper we focused on the *IntroClass* dataset. Therefore, the conclusions we draw only apply to this dataset and, more precisely, to the way in which the selected automated repair tools are able to handle *IntroClass*. Nevertheless, we believe this dataset is particularly adequate to stress some of the points we make in the paper. Particularly, considering small methods that can be easily specified in formal behavioral specification languages such as Code Contracts [6] or JML [2], allow us to determine if patches are indeed fixes or are spurious fix candidates. This is a problem that is usually overlooked in the literature: either patches are accepted as fixes (no further study on the quality of patches is made) [20], or they are subject to human inspection (which we consider severely error-prone), or are compared against developer fixes retrieved from the project repository [18] (which, as pointed out in [20], may show that automated repair tools and developers overfit in a similar way).

Also, we used the repair tools to the best of our possibilities. This is complex in itself because research tools usually have usability limitations and are not robust enough. In all cases we consulted corresponding tool developers in order to make sure we were using the tools with the right parameters, and reported a number of bugs that in some cases were fixed in time for us to run experiments with the fixed versions. The reproducibility package includes all the settings we used.

The results reported only apply to the studied tools. Other tools might behave in a substantially different way. We attempted to conduct this study on a wider class of tools, yet some tools were not available even for academic use (for instance PAR [12]), while other tools had usability limitations that prevented us from running them even on this simple dataset (this was the case for instance with SPR [15]).

VI. RELATED WORK

Various tools for program repair that employ testing as acceptance criteria for program fixes have been shown to produce spurious (incorrect) repairs. Paper [19] shows that GenProg and other tools overfit patches to the provided acceptance suites. They do so by showing that third-party generated suites reject the produced patches. Since several tools (particularly GenProg) use suites to guide the patch generation process, [19] actually shows that the original suites are not good enough. We go one step further and show that even considering more comprehensive suites the performance of the repair tools is only partially improved: fewer overfits are produced, but no new fixes. This supports the experience by the authors of [19], and generalizes it to other tools as well:

> "Our analysis substantially changed our understanding of the capabilities of the analyzed automatic patch generation systems. It is now clear that GenProg, RSRepair, and AE are overwhelmingly less capable of generating meaningful patches than we initially understood from reading the relevant papers."

The overfitting problem is also addressed in [20], where the original test suite is extended with a white-box one, automatically generated using the symbolic execution engine KLEE [3]. Research question 2 in [20] analyzes the relationship between test suite coverage and overfitting, a problem we also study in this paper. Their analysis proceeds by considering subsets of the given suite, and showing this leads to even more overfitted patches. Rather than taking subsets of the original suite, we go the other way around and extend the original suite with a substantial amount of new tests. This allows us to reach to conclusions that exceed [20], as for instance the fact that, while overfitting decreases, the fixing ratio remains very low. Also, we analyze the impact of larger suites on tool performance, which cannot be correctly addressed by using small suites.

Long and Rinard [16] also study the overfitting problem but from the perspective of the tools search space. They conclude that many tools show poor performance because their search space contains significantly fewer fixes than patches, and in some cases, the patch generation process employed produces a search space that does not contain any fixes.

VII. CONCLUSIONS AND FURTHER WORK

The significant advances in automated program analysis have enabled the development of powerful tools for assisting developers in various tasks, such as test case generation, program verification, and fault localization. The great amount of

effort that software maintenance demands is turning the focus of automated analysis into automatically *fixing* programs, and a wide variety of tools for automated program repair have been developed in the last few years. The mainstream of these tools, as we have analyzed in this paper, concentrate in using *tests as specifications*, since tests are more often found in software projects, compared to more sophisticated formal specifications, and their evaluation scales better than the analysis of formal specifications using more thorough techniques. While several researchers have acknowledged the problem of using inherently partial specifications based on tests to capture expected program behavior, the more detailed analyses that have been proposed consisted in using larger test suites, or perform manual inspections, in order to assess more precisely the effectiveness of automated program repair techniques, and the severity of the so called test suite overfitting patches [20].

Our approach in this paper has been to empirically studying the suitability of tests as fix acceptance criteria in the context of automated program repair, by checking produced patches using an automatic bug-finding tool, as opposed to previous works that used tests or manual inspections. We believe that previous approaches to analyze overfitting have failed to demonstrate the criticality of invalid patches overfitting test suites. Our results show that the percentage of valid fixes that state-of-the-art program repair tools, that use tests as acceptance criteria, are able to provide is significantly lower than the estimations of previous assessments, e.g., [20], even in simple examples such as the ones analyzed in this paper. Moreover, increasing the number of tests reduces the number of spurious fixes but does not contribute to generating more fixes, i.e, it does not improve these tools' effectiveness; instead, such increases make tools most often exhaust resources without producing patches.

Some conclusions can be drawn from these results. While weaker or lighter weight specifications, e.g., based on tests, have been successful in improving the applicability of automated analyses, as it has been shown in the contexts of test generation, bug finding, fault localization and other techniques, this does not seem to be the case in the context of automated program repair. Indeed, as our results show, using tests as specifications makes it significantly more likely to obtain invalid patches (that pass all tests) than actual fixes. This result may imply that automated program repair calls for stronger specifications, or at least significantly larger sets of tests cases, than those typically used by repair tools, with the implied necessity for novel techniques that are able to handle such large test sets.

Finally, this work opens various lines for further work. An obvious one consists in auditing patches reported in the literature, by performing an automated evaluation as the one performed in this paper. This is not a simple task in many cases, since it demands understanding the contexts of the repairs, and formally capturing the expected behavior of repaired programs.

REFERENCES

[1] A. Arcuri and X. Yao, *A Novel Co-evolutionary Approach to Automatic Software Bug Fixing*, in CEC 2008.

[2] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K. Rustan M. Leino and E. Poll, *An overview of JML tools and applications*, in STTT 7(3), Springer, 2005.

[3] C. Cadar, D. Dunbar, and D. Engler, *KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs*. In OSDI, San Diego, CA, USA, 2008.

[4] V. Debroy and W.E. Wong, *Using Mutation to Automatically Suggest Fixes to Faulty Programs*. ICST 2010, pp. 65–74.

[5] T. Durieux, M. Monperrus, *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*, Research Report, Universite Lille 1, 2016.

[6] M. Fähndrich, *Static Verification for Code Contracts*. SAS 2010: 2-5.

[7] M. Fähndrich, M. Barnett, D. Leijen, F. Logozzo, *Integrating a set of contract checking tools into visual studio*, in Proc. of TOPI 2012, IEEE, 2012.

[8] C. L. Goues, T. Nguyen, S. Forrest, W. Weimer, *GenProg: a Generic Method for Automatic Software Repair*, IEEE Transactions on Software Engineering 38, IEEE, 2012.

[9] S. Kaleeswaran, V. Tulsian, A. Kanade, A. Orso, *MintHint: Automated Synthesis of Repair Hints*, in Proc. of ICSE 2014, 2014.

[10] Y. Ke, K.T. Stolee, C. Le Goues, and Y. Brun, *Repairing Programs with Semantic Code Search*, in Proc. of ASE 2013, 2013.

[11] Y. Ke, *An automated approach to program repair with semantic code search*, Graduate Theses and Dissertations, Iowa State University, 2015.

[12] D. Kim, J. Nam, J. Song, and S. Kim, *Automatic Patch Generation Learned from Human-written Patches*, in Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), San Francisco, May 18-26, 2013, pp. 802?811.

[13] C. Le Goues, N. Holtschulte, E.K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, *The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs*, IEEE Transactions on Software Engineering (TSE), 2013.

[14] A. Leitner, I. Ciupa, B. Meyer, M. Howard, *Reconciling Manual and Automated Testing: the AutoTest Experience*, in Proceedings of the 40th Hawaii International Conference on System Sciences, 2007.

[15] F. Long and M. C. Rinard, *Staged Program Repair with Condition Synthesis*, in Symposium on the Foundations of Software Engineering (FSE). 2015.

[16] F. Long and M. C. Rinard, *Analysis of the Search Spaces for Generate and Validate Patch Generation Systems*, International Conference on Software Engineering (ICSE), 2016.

[17] S. Mechtaev, J. Yi, and A. Roychoudhury. *Directfix: Looking for simple program repairs*. In ICSE, 2015.

[18] S. Mechtaev, J. Yi, and A. Roychoudhury, *Angelix, Scalable Multiline Program Patch Synthesis via Symbolic Analysis*, International Conference on Software Engineering (ICSE), 2016.

[19] 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, pages 24?36, 2015.

[20] E.K. Smith, E. Barr, C. Le Goues, and Y. Brun, *Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair*, Symposium on the Foundations of Software Engineering (FSE), 2015.

[21] S. Staber, B. Jobstmann, R. Bloem, *Finding and Fixing Faults*, in Proceedings of Conference on Correct Hardware Design and Verification Methods, 2005.

[22] N. Tillmann and J. de Halleux, *Pex: White Box Test Generation for .NET*, in Proceedings of the Second International Conference on Tests and Proofs TAP 2008, LNCS, Springer, 2008.

[23] M. Trudel, C. Furia, M. Nordio, *Automatic C to O-O Translation with C2Eiffel*, in Proceedings of the 2012 19th Working Conference on Reverse Engineering WCRE 2012, IEEE, 2012.

[24] Weimer W., Nguyen T., Le Goues C. and Forrest S., *Automatically finding patches using genetic programming*. ICSE 2009: pp. 364–374.

[25] W. Weimer, S. Forrest, C. Le Goues, T. Nguyen, *Automatic Program Repair with Evolutionary Computation*, Communications of the ACM 53:5, ACM, 2010.