An Empirical Study on the Suitability of Test-based Patch Acceptance Criteria

LUCIANO ZEMÍN and ARIEL GODIO, Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina
CÉSAR CORNEJO, National Council for Scientific and Technical Research (CONICET) and Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina
RENZO DEGIOVANNI, Luxembourg Institute of Science and Technology, Esch-sur-Alzette, Luxembourg
SIMÓN GUTIÉRREZ BRIDA and GERMÁN REGIS, Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina
NAZARENO AGUIRRE, National Council for Scientific and Technical Research (CONICET) and Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina
MARCELO FABIÁN FRIAS, The University of Texas at El Paso, El Paso, TX, USA

In this article, we empirically study 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 *IntroClass*, a known benchmark for program repair techniques, are fed to the program repair tools GenProg, Angelix, AutoFix, and Nopol, using test suites of varying quality, including those accompanying the benchmark. We then check the produced patches against formal specifications using a bug-finding tool. Our results show that, in the studied scenarios, automated program repair tools are significantly more likely to accept a spurious program fix than producing an actual one. Using bounded-exhaustive suites larger than the originally given ones (with about 100 and 1,000 tests) we verify that overfitting is reduced but (a) few new correct repairs are generated and (b) some tools see their performance reduced by the larger suites and fewer correct repairs are produced. Finally, by comparing with previous work, we show that overfitting is underestimated in semantics-based tools and that patches not discarded using held-out tests may be discarded using a bug-finding tool.

Authors' Contact Information: Luciano Zemín, Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina; e-mail: lzemin@itba.edu.ar; Ariel Godio, Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina; e-mail: agodio@itba.edu.ar; César Cornejo, National Council for Scientific and Technical Research (CONICET) and Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina; e-mail: ccornejo@dc.exa.unrc.edu.ar; Renzo Degiovanni, Luxembourg Institute of Science and Technology, Esch-sur-Alzette, Luxembourg; e-mail: renzo.degiovanni@uni.lu; Simón Gutiérrez Brida, Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina; e-mail: sgutierrez@dc.exa.unrc.edu.ar; Germán Regis, Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina; e-mail: gregis@dc.exa.unrc.edu.ar; Nazareno Aguirre, National Council for Scientific and Technical Research (CONICET) and Department of Computer Science, National University of Río Cuarto, Río Cuarto, Argentina; e-mail: naguirre@dc.exa.unrc.edu.ar; Marcelo Fabián Frias (corresponding author), The University of Texas at El Paso, El Paso, TX, USA; e-mail: mfrias4@utep.edu.

ACM 1557-7392/2025/2-ART57
https://doi.org/10.1145/3702971

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Formal software verification**; **Empirical software validation**;

Additional Key Words and Phrases: automatic program repair, formal specifications, testing, oracle

ACM Reference format:
Luciano Zemín, Ariel Godio, César Cornejo, Renzo Degiovanni, Simón Gutiérrez Brida, Germán Regis, Nazareno Aguirre, and Marcelo Fabián Frias. 2025. An Empirical Study on the Suitability of Test-based Patch Acceptance Criteria. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 57 (February 2025), 20 pages.
https://doi.org/10.1145/3702971

1 Introduction

Software has become ubiquitous, and many of our activities depend directly or indirectly on it. Having adequate software development techniques and methodologies that contribute to producing quality software systems has become essential for many human activities. The significant advances in automated analysis techniques have led, in the last few decades, to the development of powerful tools able to assist software engineers in software development, that have proved to greatly contribute to software quality. Indeed, tools based on model checking [8], constraint solving [42], evolutionary computation [11], and other automated approaches are being successfully applied to various aspects of software development, from requirements specification [3, 15] to verification [23] and bug finding [16, 21]. Despite the great effort that is put in software development to detect software problems (wrong requirements, deficient specifications, design flaws, implementation errors, etc.), e.g., 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, and since software maintenance takes a significant part of the resources of most software projects, also economically relevant to the software development industry. 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, 10, 29, 32, 53].

While the idea of **Automatic Program Repair (APR)** is certainly appealing, automatically fixing arbitrary program defects is known to be infeasible. In the worst-case it reduces to program synthesis which is known to be undecidable for Turing-complete programming languages [7]. Thus, the various techniques that have been proposed to automatically repair programs are intrinsically incomplete, in various respects. Firstly, many techniques for automatically repairing programs need to produce repair candidates, often consisting of syntactical modifications on the original (known to be faulty) program. Clearly, *all* (even bounded) program repair candidates cannot be exhaustively considered, and thus the space of repairs 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, also, necessarily incomplete. Moreover, this latter problem requires a description of the expected behavior of the program to be fixed, i.e., a specification, subject to automated analysis, if one wants the whole repair process to remain automatic. Producing such specifications is costly, and therefore requiring these specifications is believed to undermine the applicability of automatic repair approaches. Most automated repair techniques then use *partial* specifications, given in terms of a validation test suite. Moreover, most techniques heavily rely on these tests as part of the techniques, e.g., for fault localization [53].

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 [18]. 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 [34, 47]. 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 article, we empirically study 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 and/or manual inspections. We develop a number of experiments using *IntroClass*, a known benchmark for program repair techniques, consisting of small programs solving simple assignments. Faulty programs from this benchmark are used to feed four state-of-the-art 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 [49], an automated test generation tool based on concrete/symbolic execution and constraint solving, 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.
—By improving the quality of the test suite by extending it to a sort of bounded-exhaustive suite (whose size is bounded to approximately 100 or 1,000 tests), we show that a few more correct fixes are obtained in those cases where the tool under analysis is able to cope with the suite size.
—Finally, we show that overfitting is more likely to occur in semantics-based tools than previously reported in [56]. The use of the bug-finding tool allows us to detect overfitting patches that remain undetected using held-out tests, tests that are not used during the patch generation process but are instead kept aside for the verification of the produced patches.

Notice that using *IntroClass*, a benchmark built from simple small programs (usually not exceeding 30 lines of code), is not a limitation of our analysis. If state-of-the-art tools fail to distinguish correct fixes from spurious ones on this benchmark, we should not expect them to perform better on larger, or more complex benchmarks. If anything, using *IntroClass* makes the analysis more conclusive.

This article reports research that extends previous work reported in [59]. It poses and answers **Research Questions (RQs)** that were not part of [59]. A deeper discussion of the relationship between these two works is presented in Section 5.

The article is organized as follows. After this Introduction, in Section 2 we introduce automated program repair and the overfitting problem. In Section 3 we evaluate four tools for automated program repair, namely, Angelix [37], AutoFix [52], GenProg [18], and Nopol [55]. We present and answer the RQs. In Section 4 we discuss the threats to the validity of the results presented in the article. In Section 5 we discuss related work. In Section 6 we discuss the results we obtain in the article, to finish in Section 7 with some conclusions and proposals for further work.

2 Automated Program Repair

Automated program repair techniques aim at fixing faulty programs through the application of transformations that modify the program's code. **Generation and Validation (G and V)**

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.

From the point of view of the technology underlying the repair generation techniques, a wide spectrum of approaches exist, including search-based approaches such as those based on evolutionary computation [1, 52, 53], constraint-based automated analyses such as those based on **Satisfiability Modulo Theory (SMT)** and SAT solving [17, 37], and model checking and automated synthesis [48, 51]. A crucial aspect of program repair that concerns this article is how program specifications are captured and provided to the tools. Some approaches, notably some of the initial ones (e.g., [1, 48]), require *formal specifications* in the form of pre- and post-conditions, or logical descriptions provided in some suitable logical formalism. These approaches then vary in the way they *use* these specifications to assess repair candidates; some check repair candidates against specifications using some automated verification technique [48]; some use the specifications to produce tests, which are then used to drive the patch generation and patch assessment activities [52]. Moreover, some of these tools and techniques require strong specifications, capturing the, say, "full" expected behavior of the program [1, 48], while others use contracts present in code, that developers have previously written mainly for runtime checking [52].

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. Weimer et al. mention in [54], for instance, that by requiring tests instead of formal specifications one greatly improves the practical applicability of their proposed technique. Kaleeswaran et al. [25] also mention that approaches requiring specifications suppose the existence of such specifications, a situation that is rarely true in practice; they also acknowledge the limitations of tests as specifications for repairs, and aim at a less ambitious goal than fully automatically repairing code, namely, to generate repair hints.

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* [47], has been previously identified by various researchers [34, 47, 56], 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 (e.g., the already mentioned [25]). Others take into account a notion of *quality*, and manually compare the produced patches with fixes provided by human developers or by other tools [34, 36]. 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 article.

3 Evaluation

In this section we evaluate Angelix, AutoFix, GenProg, and Nopol, four well-regarded tools that use tests as their patch acceptance criterion. The evaluation is performed on the *IntroClass* dataset, which is described in detail in Section 3.1. The dataset contains student-developed solutions for six 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.

The tools were selected because they use different underlying techniques for patch generation, and also due to the existence of mature-enough implementations that would allow to carry on the experiments. Angelix [37] collects semantic information from controlled symbolic executions of the program and uses MaxSMT [43] to synthesize a patch. AutoFix [52] is intended for Eiffel [39] programs repair. It is contract-based, and is based in AutoTest [40] for automated test generation. GenProg [18] uses genetic algorithms to search for a program variant that retains correct functionality yet passes failing tests. Finally, Nopol [55] collects program execution information and uses SMTs [2] to generate patches. Unlike Angelix, Nopol focuses on the repair of buggy conditional statements.

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 repair, 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 RQ.

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 it is for it to be a *fix*?

Patch correctness is typically determined by manual inspection. Since manual inspections are error-prone (in fact, the faulty routines that constitute the *IntroClass* dataset 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* [38]. More precisely, we will translate patches into C#, and equip these with pre- and post-conditions captured using Code Contracts [14]; we will then search for inputs that violate these assertions via concrete/symbolic execution and SMT solving, using the Pex tool [49]. Finally, to prevent any error introduced by the above described process, we run the corresponding test generated by Pex on the original patched method to check that it actually fails.

To assess the above RQ, 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 Section 4). 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). We will show, with the aid of the bug-finding tool that patches that were hinted as correct in [56], are not.

Notice that when a patch is produced, but this patch is not a fix, 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 [47, 56] enrich 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 [47, Figure 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 RQ:

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, with quality suites according to many different of these criteria, is infeasible. Our approach will be to enrich the validation test suites, those provided with the dataset, by adding bounded-exhaustive suites [4] 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. This leads to our third RQ:

RQ3: How does test suite size affect the performance of test-based automated repair tools?

As we mentioned earlier in this section, patches are classified as correct (i.e., fixes) or not using either manual inspections or, as in [56], using held-out tests. We consider these to be error-prone procedures to assess the correctness of a patch. In [56] overfitting of the Angelix APR tool is analyzed over the *IntroClass* dataset. Held-out tests are used to determine non-overfitting patches. Our fourth RQ is then stated as:

RQ4: Can we produce substantial evidence on the fact held-out tests underestimate overfitting patches in semantics-based APR tools such as Angelix?

The remaining part of this section is organized as follows. Section 3.1 describes the *IntroClass* dataset. Section 3.2 describes the experimental setup we used. Section 3.3 describes the reproducibility package we are providing in order to guarantee reproducibility of the performed experiments. Finally, Section 3.5 presents the evaluations performed, and discusses RQ1–RQ4.

3.1 The *IntroClass* Dataset

The *IntroClass* benchmark is thoroughly discussed in [31]. It contains student-developed C programs for solving six 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 [27, 46, 47, 56], and its simplicity reduces the requirements on tool scalability. The benchmark comprises methods to solve the following problems:

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 in reverse order.

Grade: Receives five 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 four values induce five 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 three integer input values.

Smallest: Compute the smallest value among four 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 [31], whose methods are given in the C language, and a Java translation of the original dataset described in [12]. Some of the programs that result from 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 [12]. 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, which include AutoFix and Angelix, we need to consider yet other versions of the *IntroClass* dataset:

— *IntroClass* Eiffel: This new version is the result of translating the original C dataset into Eiffel. For the translation, we employed the C2Eiffel tool [50]; 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.

— *IntroClass* Angelix: 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 return the output in a single integer or char. Since in several variants the errors consist of modifications of the input/output Strings, which are stripped out by the instrumentation, the faulty versions became "correct."

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

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 [26].

Table 1. Description of the *IntroClass* C, Java, and Eiffel Datasets

	Checksum	Digits	Grade	Median	Smallest	Syllables	**Total**
IntroClass C (GenProg)	46	143	136	98	84	63	570
IntroClass C (Angelix)	31	149	36	58	41	44	359
IntroClass Java (Nopol)	11	75	89	57	52	13	297
IntroClass Eiffel (AutoFix)	45	141	72	86	56	77	477
Test suite size	16	16	18	13	16	16	95

resulted in duplicate files, which we removed using the *diff* tool to reduce bias (notice that the *IntroClass* version used in [26] and reported in Figure 1 does not eliminate duplicates and contains 208 extra datapoints). Table 1 describes the four datasets and, for each dataset, the number of faulty versions for each method. The sizes of their corresponding test suites are only relevant for C and Java since, in the case of AutoFix, tests are automatically produced using AutoTest [33] (the tool does not receive user-provided test suites).

3.2 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 3.5. 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 suites. For those programs in which it is feasible, we consider bounded-exhaustive suites. Bounded-exhaustive suites contain all the inputs that can be generated within user-provided bounds. We will use such suites for programs *digits*, *median*, *smallest*, and *syllables*. Program *grade* uses floats and, therefore, even small bounds would produce suites that are too big; therefore, for program *grade* we use tests that are not bounded exhaustive, but that are part of a bounded exhaustive suite. 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 in Tables 2 and 3. 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, i.e., S1,000 extends S100 in all cases.

Table 2. Specification of Test Suites $S100$

		Test inputs specification	Total
$S100_{\text{checksum}}$	=	$\{c_0, \ldots, c_k \mid 0 \leq k \leq 4 \wedge \forall_{0 \leq i \leq k} c_i \in \{'a', 'b', 'c'\}\}$	120
$S100_{\text{digits}}$	=	$\{k \mid -64 \leq k \leq 63\}$	128
$S100_{\text{grade}}$	=	$\{(f_1, \ldots, f_4, score) \mid (\forall_{1 \leq i \leq 4} f_i \in \{30, 40, 50, 60, 70, 80\}) \wedge (f_1 > f_2 > f_3 > f_4) \wedge score \in \{5, 10, 15, 20, \ldots, 90\}\}$	285
$S100_{\text{median}}$	=	$\{(k_1, k_2, k_3) \mid \forall_{1 \leq i \leq 3} -2 \leq k_i \leq 2\}$	125
$S100_{\text{smallest}}$	=	$\{(k_1, k_2, k_3, k_4) \mid \forall_{1 \leq i \leq 4} -2 \leq k_i \leq 1\}$	256
$S100_{\text{syllables}}$	=	$\{c_0, \ldots c_k \mid 0 \leq k < 4 \wedge \forall_{0 \leq i \leq k} c_i \in \{'a', 'b', 'c'\}\}$	120

Table 3. Specification of Test Suites $S1,000$

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

Along the experiments we report in this section, we used a workstation with Intel Core i7 2600, 3.40 GHz, and 8 GB of RAM running Ubuntu 16.04 LTS x86_64. The experiments involving Pex were performed on a virtual machine (VirtualBox) running a fresh install of Windows 7 SP1. The specific version of the software used in the experiments, including that of the APR tools, can be found on the reproducibility package. In general, finding an appropriate timeout depends on the context in which the tool is being used. Perhaps, for mission-critical applications a large timeout is more appropriate, while for other domains, it may be too expensive to devote 1 hour to (most probably failed) repair attempts. We set a 2-hour timeout; enough for the tools to run, and at the same time reasonable for the time running all the experiments will require.

3.3 Reproducibility

The empirical study we present in this article involves a large set of different experiments. These involve four different datasets (versions of *IntroClass*, as described in the previous section), configurations for four different repair tools across three different languages, and three 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 in

`https://sites.google.com/a/dc.exa.unrc.edu.ar/test-specs-program-repair/`

Instructions to reproduce each experiment are provided therein.

3.4 Why Use Bounded-exhaustive Suites

RQs 2 and 3 discuss the impact of using alternative (w.r.t. test suites used as specifications of the programs being developed) test suites in overfitting and tool scalability. It is expected that these held-out tests will allow one to expose those patches generated by an APR tool that overfit to the

Table 4. Repair Statistics for the Nopol Automated Repair Tool

Method	#Versions	Suite	#Patches	#Fixes	%Patches	%Fixes
Checksum	11	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
Digits	75	O	7	1	9.3%	1.3%
		O ∪ S100	2	1	2.7%	1.3%
		O ∪ S1,000	2	1	2.7%	1.3%
Grade	89	O	2	1	2.2%	1.1%
		O ∪ S100	2	1	2.2%	1.1%
		O ∪ S1,000	2	1	2.2%	1.1%
Median	57	O	11	4	19.3%	7%
		O ∪ S100	4	4	7%	7%
		O ∪ S1,000	4	4	7%	7%
Smallest	52	O	12	0	23.1%	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%

specification suite. The first problem, namely, the impact on overfitting of considering alternative suites, has been already discussed in the literature by using, besides the original suite, new white-box suites automatically generated in order to satisfy a coverage criterion (usually, branch coverage) [26, 47, 56]. Using white-box suites as held-out tests is rarely useful in software development. They are obtained from a (usually non-existent) correct implementation whose complexity usually differs greatly from the students-developed version losing both, the property of the suite being white-box and the satisfaction of the coverage criterion. Also, they may not be comprehensive enough. For example, for the *IntroClass* dataset, the white-box suites used in [56] have between 8 and 10 tests, and keeping portions of the suite to check the impact of larger/smaller suites on assessing scalability seems at least risky. As an alternative of using small white-box suites, that suffer from the mentioned methodological limitations, we propose the use of bounded-exhaustive suites whenever possible, or similar ones when bounded-exhaustiveness is not an option. They can easily be made to grow in size as much as necessary, and they capture a similar behavior to formal specifications in a fragment of the program domain. We will be exploring alternatives to bounded-exhaustive suites with similar characteristics as further work.

3.5 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 RQ1–RQ4 in Sections 3.6–3.9. Tables 4–7 summarize the experimental data. On these tables, we report a patch/fix if *any* repair mode/configuration or execution produced a patch/fix. For example, we considered 10 executions of GenProg due to its random behavior. It suffices only one execution to find a patch to be reported.

3.6 RQ1

This RQ addresses overfitting, a well-known limitation of G and V APR approaches that use test suites as the validation mechanism. The use of patch validation techniques based on human

Table 5. Repair Statistics for the GenProg Automatic Repair Tool

Method	#Versions	Suite	#Patches	#Fixes	%Patches	%Fixes
Checksum	46	O	3	2	6.5%	4.3%
		O ∪ S100	23	0	50%	0%
		O ∪ S1,000	22	1	47.8%	2.2%
Digits	143	O	29	12	20.3%	8.4%
		O ∪ S100	17	10	11.9%	7%
		O ∪ S1,000	13	7	9.1%	4.9%
Grade	136	O	2	0	1.5%	0%
		O ∪ S100	50	0	36.8%	0%
		O ∪ S1,000	23	0	16.9%	0%
Median	98	O	37	13	37.8%	13.3%
		O ∪ S100	67	0	68.4%	0%
		O ∪ S1,000	51	3	52%	3.1%
Smallest	84	O	37	3	44%	3.6%
		O ∪ S100	61	0	72.6%	0%
		O ∪ S1,000	47	0	56%	0%
Syllables	63	O	19	0	30.2%	0%
		O ∪ S100	38	0	60.3%	0%
		O ∪ S1,000	37	0	58.7%	0%

Table 6. Repair Statistics for the AutoFix Automatic Repair Tool

Method	#Versions	Suite	#Patches	#Fixes	%Patches	%Fixes
Checksum	45	O	1	0	2.2%	0%
		O ∪ S100	1	0	2.2%	0%
		O ∪ S1,000	1	0	2.2%	0%
Digits	141	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
Grade	72	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
Median	86	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
Smallest	56	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
Syllables	77	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%

Table 7. Repair Statistics for the Angelix Automatic Repair Tool

Method	#Versions	Suite	#Patches	#Fixes	%Patches	%Fixes
Checksum	31	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%
Digits	149	O	20	5	13.4%	3.4%
		O ∪ S100	13	10	8.7%	6.7%
		O ∪ S1,000	8	4	5.4%	2.7%
Grade	36	O	7	7	19.4%	19.4%
		O ∪ S100	7	7	19.4%	19.4%
		O ∪ S1,000	5	5	13.9%	13.9%
Median	58	O	29	10	50.0%	17.2%
		O ∪ S100	15	15	25.9%	25.9%
		O ∪ S1,000	6	6	10.3%	10.3%
Smallest	41	O	33	1	80.5%	2.4%
		O ∪ S100	1	1	2.4%	2.4%
		O ∪ S1,000	1	1	2.4%	2.4%
Syllables	44	O	0	0	0%	0%
		O ∪ S100	0	0	0%	0%
		O ∪ S1,000	0	0	0%	0%

inspections or comparisons with developer patches (or even accepting patches as fixes without further discussion) has not allowed the community to identify the whole extent of this problem. For example, paper [26] includes the table we reproduce in Figure 1. The table gives 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 while, in fact, it might be the case that none of the patches is actually a fix. Actually, as our experiments reported in Table 5 show, only 30 out of 570 faults were correctly fixed (which gives a fixing ratio of 5.3%, well below the 36.8% presented in [26]).

We have obtained similar results for the other tools under analysis. Angelix patches 89 faults out of 360 program variants (a ratio of 24.7%), yet only 23 patches are fixes (the ratio reduces to 6.4%). The remaining patches were discarded with the aid of Pex. Nopol patched 32 out of 297 versions (10.7%), 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 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 one patch (which was an invalid fix).

As previously discussed in the beginning of Section 3, 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.

Table 8. Number of Timeouts Reached by Tool and Validation Suite

	Nopol	GenProg	Angelix
O	0	2	5
O ∪ S100	0	135	47
O ∪ S1,000	0	237	69

3.7 RQ2

This RQ relates to the impact of more thorough validation suites on overfitting, as well as on the quality of the produced patches. Table 4 shows that Nopol profits from larger suites in order to reduce overfitting significantly. It suffices to consider suite O ∪ S100 to notice a reduction in overfitting. The number of patches is reduced from 32 to 8. Unfortunately, the number of fixes remains low. This shows that Nopol, when fed with a better quality evaluation suite, is able to produce (a few) good quality fixes. GenProg, on the other hand, shows an interesting behavior (see Table 5): It doubles the number of patches with suite O ∪ S100, yet the number of fixes is reduced from 30 to 10. With suite O ∪ S1,000 produces around 50% more patches but the number of fixes is reduced from 30 to 11. We believe this is due to its random nature. Angelix (see Table 7) sees its overfitting reduced. Interestingly, suite O ∪ S100 allows Angelix to obtain five more fixes for method `digits` and five more for method `median`. 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.

3.8 RQ3

This RQ relates to the impact larger validation suites may have on repair performance. In most of the evaluated tools this impact can be illustrated by showing the number of timeouts reached during the repair process. Table 8 reports, for each tool able to use suites S100 and S1,000, and for each validation suite, the number of timeouts reached during repair. We report a timeout only when a timeout occurs for *all* repair modes/configurations and executions. For example, we considered two repair modes (condition and pre-condition) for Nopol. Depending on the suite, some of them reached a timeout, however, there is no faulty version for which all of them do. Recall that AutoFix generates its validation suites from user-provided contracts, and is therefore left out of this analysis. While Nopol's mechanism for data collection discards tests that are considered redundant (which as shown in Table 8 makes Nopol resilient to suite size increments), GenProg and Angelix are both sensitive to the size of the evaluation suite.

3.9 RQ4

This RQ addresses our intuition that using held-out tests, as a means to determine if a patch found by an APR tool is indeed correct, is an error-prone procedure. This procedure is widely used and accepted by the community [26, 30, 47, 56, 58]. In [56, Table 2], reproduced in Figure 2, we see the overfitting reported for Angelix on the *IntroClass* benchmark using the originally provided black-box test suites. The table omits methods `syllables` and `checksum` for which no patches were generated. Table 9 compares the number of fixes obtained in [56] (reproduced in Figure 2) and in this article in Table 7 for those methods that are in the intersection of both studies, namely, `median`, `smallest`, and `digits`.

While there is a discrepancy in the results reported in Table 9, this does not imply *per se* that an error has been made. Methods have been instrumented, parameters may include subtle differences

	Black box
Subject	Angelix
smallest	27/37
median	29/38
digits	5/6

Fig. 2. Overfitting patches produced by Angelix as reported in [56, Table 2].

Table 9. Comparing the Number of Reported Fixes with [56]

	Figure 2	Table 7
Median	9	5
Smallest	10	10
Digits	1	1

Table 10. Quality of Patches That Pass the Held-out Tests

	#Patches	#Pass O	#Pass Pex
Median	887 [58]	130 [28]	66 [12]
Smallest	1,155 [51]	283 [26]	11 [3]
Digits	78 [22]	9 [7]	3 [1]

that lead to different results, and so on. These discrepancies, nevertheless, called our attention and led to RQ4. Paper [56] reports a reproducibility package [57] that, in particular, includes all the patches generated by Angelix. It is not clear how the files in [57] match the experiments reported in [56] (the number of files does not match the results reported in the paper). Still, many files are available that allow us to study this RQ in depth. Disregarding their provenance, there are 45,131 patches in the reproducibility package [57]. Since there may be repeated patches, we removed repetitions and 2,120 patches remained. For each of these patches we checked overfitting using the test suite provided in *IntroClass* (suite O in our tables) as held-out tests. Also, we checked overfitting using Pex. Finally, as we did with our experiments, we ran the corresponding test generated by Pex (if any) on the original patched method to check that it actually fails. Table 10 reports the total number of unique patches for each method. Since there might be multiple patches for a specific student version, we also present the number of versions involved in brackets. For example, there are 78 unique patches for the method digits in the available reproducibility package, yet they correspond to 22 different student versions. Table 10 provides strong evidence that using held-out tests to assess the correctness of a patch is unacceptable since more than 80% of them are deemed correct by the held-out tests, yet they are incorrect (the 80% value is obtained from Table 10 by calculating $(\#pass\ O - \#Pass\ PEX)/\#Pass\ O$). Interestingly, the number of student versions fixed reported in Table 10 (after an in-depth analysis of the reproducibility package obtained from [56]) and Table 7 are very similar. This observation suggests that despite the differences in the instrumentation of the dataset and the experimental setup, running the same tool on the same dataset produced the same results, providing evidence against any bias introduced during our experiments.

4 Threats to Validity

In the article 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 article. Particularly, considering small methods that can be easily specified in formal behavioral specification languages such as Code Contracts [13] or JML [5] 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) [47], or they are subject to human inspection (which we consider severely error-prone), or are compared against developer fixes retrieved from the project repository [37] (which, as pointed out in [47], may show that automated repair tools and developers overfit in a similar way). Since these tools work on different languages (GenProg and Angelix work on C code, while Nopol works on Java code and AutoFix repairs Eiffel code), the corresponding datasets are different across languages as explained in Section 3.2. Therefore, the reader is advised to not draw conclusions by comparing between tools working on different datasets.

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.

Notice that we are using Pex as our golden standard, i.e., if a patch is deemed correct by Pex, it is accepted as a fix. This may not always be the case. If that happens, it will count against our hypothesis. For the experimental evaluation we used a 2-hour timeout. We consider this an appropriate timeout, but increasing it may yield added results.

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 [29]), 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 [34]).

5 Related Work

This article extends previous results published by the authors in [59]. The paper addressed the use of specifications and automated bug-finding to assess overfitting in automated program repair tools. Similar ideas were later also published in [44], where OpenJML [9] is used for verification purposes rather than Pex. Only the overfitting problem is analyzed in [44], but on a different benchmark of programs. This new benchmark is a valuable contribution of [44]. Instead, we stick to the *IntroClass* dataset, which has simpler programs, has been used as a benchmark in other papers, and serves as a lower bound for the evaluation of APR tools, i.e., if tools fail to properly fix these simple programs, it is hardly the case they will be successful on more complex ones.

Automatic program fixing has become over the last few years a very active research topic, and various tools for program repair are now available, many of which we have already referred to earlier in this article. These generally differ in their approaches for producing program patches, using several different underlying approaches, including search-based techniques, evolutionary computation, pattern-based program fixing, program mutation, synthesis, and others. Since this article is mainly concerned with how these program fixing approaches evaluate the produced fix candidates, we will concentrate on that aspect of automated program repair techniques. A very small set of program repair techniques use *formal specifications* as acceptance criteria for program

fixes. Gopinath et al. [17] propose a technique to repair programs automatically, by employing SAT solving for various tasks, including the construction of repair values from faulty programs where suspicious statements are parameterized, and checking whether the repair candidates are indeed fixes; they use contracts specified in Alloy [20], and SAT-based bounded verification for checking candidate programs against specifications. Staber et al. [48] apply automated repairs on programs captured as finite state machines, whose intended behavior is captured through linear time temporal logic [8]; repair actions essentially manipulate the state transition relation, using a game-theoretic approach. von Essen and Jobstmann [51] propose a technique to repair reactive systems, specified by formal specifications in linear time temporal logic, resorting to automated synthesis. The approach by Arcuri and Yao [1] applies to sequential programs accompanied by formal specifications in the form of first-order logic pre- and post-conditions, and uses genetic programming to evolve a buggy program in the search for a fix, driven by a set of tests automatically computed from the formal specification and the program. Their use of formal specifications is then weaker than the previously mentioned cases. Wei et al. [52] propose a technique that combines tests for fault localization with specifications in the form of contracts, for automatically repairing Eiffel programs. Their technique may correct both programs and contracts; it uses automatically generated tests to localize faults and to instantiate fix schemas to produce fix candidates; fix candidates are then assessed indirectly against contracts, since they are evaluated on a collected set of failing and passing tests, automatically built using random test generation using the contracts.

All other automated repair tools we are aware of use tests as specifications, mainly as a way of making the corresponding techniques more widely applicable, since tests can be more commonly found in software projects and their use scales more reasonably than other verification approaches. We summarize here a set of known tools and techniques that use tests as specifications. The BugFix tool by Jeffrey et al. [22] applies to C programs and uses *tests as specifications*; the tool employs machine learning techniques to produce bug-fixing suggestions from rules learned from previous bug fixes. Weimer et al. [54] use genetic algorithms for automatically producing program fixes for C programs, using tests as specifications too; moreover, they emphasize the fact that tests as opposed to formal specifications lead to wider applicability of their technique. Kern and Esparza [28] repair Java programs by systematically exploring alternatives to *hotspots* (error prone parts of the code), provided that the developers characterize hotspot constructs and provide suitable syntactic changes for these; they also use tests as specifications, but their experiments tend to use larger test sets compared to the approaches based on evolutionary computation. Debroy and Wong [10] propose a technique that combines fault localization with mutation for program repair; fault localization is a crucial part of their technique, in which a test suite is involved, the same one used as acceptance criterion for produced program patches. Tool SemFix by Nguyen et al. [41] combines symbolic execution with constraint solving and program synthesis to automatically repair programs; this tool uses provided tests both for fault localization and for producing constraints that would lead to program patches that pass all tests. Kaleeswaran et al. [25] propose a technique for identifying, from a faulty program, parts of it that are likely to be part of the repaired code, and suggest expressions on how to change these. Tests are used in their approach both for localizing faults and for capturing the expected behavior of a program to synthesize hints for fixes. Ke [27] proposes an approach to program repair that identifies faulty code fragments and looks for alternative, human-written, pieces of code, that would constitute patches of the faulty program; while their approach uses constraints to capture the expected behavior of fragments and constraint solving to find patches, this behavior is taken from tests, and the produced patches are in the end evaluated against a set of test cases, for acceptance. Long and Rinard [34] propose SPR, a technique based on the use of transformation schemas that target a wide variety of program defects, and are instantiated using a novel condition synthesis algorithm. SPR also uses tests as specifications, not only as acceptance

criterion but also as part of its condition synthesis mechanism. Mechtaev et al. [37] propose Angelix, a tool for program repair based on symbolic execution and constraint solving, supported by a novel notion of angelic forest to capture information regarding (symbolic) executions of the program being repaired; while Angelix uses symbolic execution and constraint solving, the intended behavior of the program to be repaired is in this case also captured through test cases. Finally, Xuan et al. [55] propose Nopol, which also resorts to constraint solving to produce patches from information originating in test executions, encoded as constraints. Again, Nopol uses tests both in the patch generation process, and as acceptance criterion for its produced fixes to produce patches from information originating in test executions, encoded as constraints.

Various tools for program repair that employ testing as acceptance criteria for program fixes have been shown to produce spurious (incorrect) repairs. Paper [46] 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, [46] 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 [46], 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 [47, 56], where the original test suite is extended with a white-box one, automatically generated using the symbolic execution engine KLEE [6]. RQ2 in [47] analyzes the relationship between test suite coverage and overfitting, a problem we also study in this article. 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 [47], 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 [35] also study the overfitting problem but from the perspective of the tools search space. It concludes 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.

Kali [46] was developed with the purpose of generating patches that delete functionality. RSRepair [45] is an adaptation of GenProg that substitutes genetic programming by random search.

6 Discussion

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 article, 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 [47].

Our approach in this article has been to empirically study 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., [47], even in simple examples such as the ones analyzed in this article. 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.

7 Conclusions and Further Work

Some conclusions can be drawn from these results. While weaker or lighterweight 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. We foresee three lines of research in order to overcome this fundamental limitation:

(1) Use of strong formal specifications describing the problem to be solved by the program under analysis. In some domains (for instance when automatically repairing formal models [19, 60, 61]) this is the natural way to go. For programs, more work is necessary in order to assess if partial formal specifications present improvements over test-based specifications.
(2) Use more comprehensive test suites, as for instance bounded-exhaustive suites. These capture a portion of the semantics of the strong formal specification. Since these suites are likely to be large in size, new tools must be prepared to deal with large suites.
(3) Include a human in the loop that assesses if a repair candidate is indeed a fix. If she determines it is not, she may expand the suite with new tests. This iterative process has limitations (the human may make wrong decisions), but has good chances of being more effective than test-based specifications.

This work opens more lines for further work. An obvious one consists of auditing patches reported in the literature, by performing an automated evaluation as the one performed in this article. 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. Also, in the article we used bounded-exhaustive test suites that contain approximately 100 or 1,000 tests. For some tools we saw improvement in the number of fixes, and for others we saw that large suites deem the tools useless. We will study how tools behave when finer granularity is applied in the construction of bounded-exhaustive suites, hoping to find sweet spots that favor the quality of the produced patches.

In this article we are not proposing the use of specifications and verification tools along automated program repair in industrial settings. Specifications are scarce, and producing good-enough specifications is an expensive task. Yet it is essential that APR tool users be aware of the actual limitations APR tools have. Still, in an academic setting we believe that checking tools against

the IntroClass dataset and assessing the quality of patches using formal specifications should be a standard. This article provides all the infrastructure necessary to make this task a simple one.

References

[1] A. Arcuri and X. Yao. 2008. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *CEC '08*, 162–168.

[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. 2009. Satisfiability modulo theories. In *Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications*. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh (Eds.), Vol. 185, IOS Press, 825–885.

[3] R. Bharadwaj and C. Heitmeyer. 1999. Model Checking Complete Requirements Specifications Using Abstraction. *Automated Software Engineering* 6, 1 (1999), 37–68.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *ISSTA '02*, 123–133.

[5] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll. 2005. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer* 7, 3 (2005), 212–232.

[6] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI '08*, 209–224.

[7] A. Church. 1960. Application of Recursive Arithmetic to the Problem of Circuit Synthesis In *Summaries of Talks Presented at the Summer Institute for Symbolic Logic Cornell University*, 2nd edn. Communications Research Division, Institute for Defense Analyses, Princeton, NJ, 3–50.

[8] E. Clarke, O. Grumberg, and D. Peled. 2000. *Model Checking*. MIT Press.

[9] R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods Symposium*. Springer, 472–479.

[10] V. Debroy and W. E. Wong. 2010. Using Mutation to Automatically Suggest Fixes to Faulty Programs. In *ICST '10*, 65–74.

[11] K. A. De Jong. 2006. *Evolutionary Computation: A Unified Approach*. MIT Press.

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

[13] M. Fähndrich. 2010. Static Verification for Code Contracts. In *SAS '10*, 2–5.

[14] M. Fähndrich, M. Barnett, D. Leijen, and F. Logozzo. 2012. Integrating a Set of Contract Checking Tools into Visual Studio. In *TOPI '12*. IEEE, 43–48.

[15] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. 2001. Model Checking Early Requirements Specification in Tropos. In *RE '01*, 174–181.

[16] J. P. Galeotti, N. Rosner, C. López Pombo, and M. F. Frias. 2013. TACO: Efficient SAT-based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1283–1307.

[17] D. Gopinath, M. Z. Malik, and S. Khurshid. 2011. Specification-based Program Repair Using SAT. In *TACAS '11*, 173–188.

[18] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72.

[19] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. 2021. Bounded Exhaustive Search of Alloy Specification Repairs. In *ICSE '21*, 1135–1147.

[20] D. Jackson. 2006. *Software Abstractions: Logic, Language and Analysis*. The MIT Press.

[21] D. Jackson and M. Vaziri. 2000. Finding Bugs with a Constraint Solver. In *ISSTA '00*. ACM, 14–25.

[22] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. 2009. BugFix: A Learning-based Tool to Assist Developers in Fixing Bugs. In *ICPC '09*, 70–79.

[23] R. Jhala and R. Majumdar. 2009. Software Model Checking. *ACM Computing Surveys* 41, 4 (2009), 1–54.

[24] B. Jobstmann, A. Griesmayer, and R. Bloem. 2005. Program Repair as a Game. In *CAV '05*, 226–238.

[25] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. 2014. MintHint: Automated Synthesis of Repair Hints. In *ICSE '14*, 266–276.

[26] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. 2013. Repairing Programs with Semantic Code Search. In *ASE '13*, 295–306.

[27] Y. Ke. 2015. *An Automated Approach to Program Repair with Semantic Code Search*. Graduate Theses and Dissertations. Iowa State University.

[28] C. Kern and J. Esparza. 2010. Automatic Error Correction of Java Programs. In *FMICS '10*, 67–81.

[29] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *ICSE '13*, 802–811.

[30] X. Kong, L. Zhang, W. E. Wong, and B. Li. 2015. Experience Report: How Do Techniques, Programs, and Tests Impact Automated Program Repair? In *ISSRE '15*. IEEE, 194–204.

[31] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2013. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41 (2013), 1236–1256.

[32] C. Le Goues, M. Pradel, and A. Roychoudhury. 2019. Automated Program Repair. *Communications of the ACM* 62, 12 (2019), 56–65.
[33] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. 2007. Reconciling Manual and Automated Testing: The AutoTest Experience. In *HICSS '07*, 261.
[34] F. Long and M. C. Rinard. 2015. Staged Program Repair with Condition Synthesis. In *FSE '15*, 166–178.
[35] F. Long and M. C. Rinard. 2016. Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *ICSE '16*, 702–713.
[36] S. Mechtaev, J. Yi, and A. Roychoudhury. 2015. Directfix: Looking for Simple Program Repairs. In *ICSE '15*, 448–458.
[37] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix, Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE '16*, 691–701.
[38] B. Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25 (1992), 40–51.
[39] B. Meyer. 2013. *A Touch of Class*, 2nd corrected ed. Springer.
[40] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. 2009. Programs That Test Themselves. *IEEE Software* 42 (2009), 22–24.
[41] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *ICSE '13*, 772–781.
[42] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM* 53, 6 (2006), 937–977.
[43] R. Nieuwenhuis and A. Oliveras. 2006. On SAT Modulo Theories and Optimization Problems. In *SAT '06*. Armin Biere and Carla P. Gomes (Eds.), Lecture Notes in Computer Science, Vol. 4121, Springer, 156–169.
[44] A. Nilizadeh, G. T. Leavens, X.-B. D. Le, C. S. Păsăreanu, and D. R. Cok. 2021. Exploring True Test Overfitting in Dynamic Automated Program Repair Using Formal Methods. In *ICST '21*, 229–240.
[45] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. 2014. The Strength of Random Search on Automated Program Repair. In *ICSE '14*, 254–265.
[46] Z. Qi, F. Long, S. Achour, and M. C. Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *ISSTA '15*, 24–36.
[47] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *FSE '15*, 532–543.
[48] S. Staber, B. Jobstmann, and R. Bloem. 2005. Finding and Fixing Faults. In *CHARME '05*, 35–49.
[49] N. Tillmann and J. de Halleux. 2008. Pex: White Box Test Generation for .NET. In *TAP '08*. Springer, 134–153.
[50] M. Trudel, C. Furia, and M. Nordio. Automatic C to O-O Translation with C2Eiffel. In *WCRE '12*. IEEE, 501–502.
[51] C. von Essen and B. Jobstmann. 2013. Program Repair without Regret. In *CAV '13*, 896–911.
[52] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. 2010. Automated Fixing of Programs with Contracts. In *ISSTA '10*, 61–72.
[53] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *ICSE '09*, 364–374.
[54] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. 2010. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM* 53, 5 (2010), 109–116.
[55] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43 (2016), 34–55.
[56] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in Semantics-based Automated Program Repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.
[57] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Paper [56] reproducibility package. Retrieved September 2, 2021 from https://doi.org/10.5281/zenodo.1012686
[58] H. Ye, M. Martinez, and M. Monperrus. 2021. Automated Patch Assessment for Program Repair at Scale. *Empirical Software Engineering* 26, 2 (2021), 1–38.
[59] Luciano Zemín, Simón Gutiérrez Brida, Ariel Godio, César Cornejo, Renzo Degiovanni, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. 2017. An Analysis of the Suitability of Test-based Patch Acceptance Criteria. In *SBST@ICSE '17*, 14–20.
[60] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Nazareno Aguirre, Marcelo F. Frias, and Hamid Bagheri. 2022. ATR: Template-based Repair for Alloy Specifications. In *ISSTA '22*, 666–677.
[61] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F. Frias, Nazareno Aguirre, and Hamid Bagheri. 2021. FLACK: Counterexample-guided Fault Localization for Alloy Models. In *ICSE '21*, 637–648.

Received 25 June 2022; revised 25 July 2024; accepted 30 August 2024