Empir Software Eng (2018) 23:2901–2947
https://doi.org/10.1007/s10664-017-9550-0

Do automated program repair techniques repair hard and important bugs?

Manish Motwani[1] · **Sandhya Sankaranarayanan**[1] · **René Just**[1] · **Yuriy Brun**[1]

Published online: 18 November 2017

Abstract Existing evaluations of automated repair techniques focus on the fraction of the defects for which the technique can produce a patch, the time needed to produce patches, and how well patches generalize to the intended specification. However, these evaluations have not focused on the applicability of repair techniques and the characteristics of the defects that these techniques can repair. Questions such as "Can automated repair techniques repair defects that are hard for developers to repair?" and "Are automated repair techniques less likely to repair defects that involve loops?" have not, as of yet, been answered. To address such questions, we annotate two large benchmarks totaling 409 C and Java defects in real-world software, ranging from 22K to 2.8M lines of code, with measures of the defect's importance, the developer-written patch's complexity, and the quality of the test suite. We then analyze relationships between these measures and the ability to produce patches for the defects of seven automated repair techniques — AE, GenProg, Kali, Nopol, Prophet, SPR, and TrpAutoRepair. We find that automated repair techniques are less likely to produce patches for defects that required developers to write a lot of code or edit many files, or that have many tests relevant to the defect. Java techniques are more likely to produce patches for high-priority defects. Neither the time it took developers to fix a defect nor the test suite's coverage correlate with the automated repair techniques' ability to produce

Communicated by: Martin Monperrus and Westley Weimer

✉ Manish Motwani
mmotwani@cs.umass.edu

Sandhya Sankaranarayanan
ssankar@cs.umass.edu

René Just
rjust@cs.umass.edu

Yuriy Brun
brun@cs.umass.edu

[1] College of Information and Computer Science, University of Massachusetts, Amherst, MA 01003-9264, USA

patches. Finally, automated repair techniques are less capable of fixing defects that require developers to add loops and new function calls, or to change method signatures. These findings identify strengths and shortcomings of the state-of-the-art of automated program repair along new dimensions. The presented methodology can drive research toward improving the applicability of automated repair techniques to hard and important bugs.

Keywords Automated program repair · Repairability

1 Introduction

Automated program repair research has produced dozens of repair techniques that use a buggy program and a specification of that program (without the bug) to produce a program variant that satisfies the specification (Weimer et al. 2013; Jin et al. 2011; Bradbury and Jalbert 2010; Arcuri and Yao 2008; Carzaniga et al. 2013; Wei et al. 2010; Pei et al. 2014; Liu and Zhang 2012; Jeffrey et al. 2009; Wilkerson et al. 2012; Perkins et al. 2009; Coker and Hafiz 2013; Debroy and Wong 2010; Demsky et al. 2006; Orlov and Sipper 2011; Weimer et al. 2009; Weimer et al. 2009; Le Goues et al. 2012a, b; Gopinath et al. 2011; Carbin et al. 2011; Elkarablieh and Khurshid 2008; Qi et al. 2015; Dallmeier et al. 2009; Kim et al. 2013; Tan and Roychoudhury 2015; Nguyen et al. 2013; Sidiroglou and Keromytis 2005; Qi et al. 2013; Mechtaev et al. 2015, 2016; Ke et al. 2015; Sidiroglou-Douskos et al. 2015; Long and Rinard 2015, 2016b; DeMarco et al. 2014; Jiang et al. 2016; Weiss et al. 2017; Le et al. 2017). This specification is typically partial, e.g., a set of tests describing the desired behavior. Due to the partial nature of the specification, patches that satisfy the written specification may not satisfy the intended, unwritten, full specification, and recent work has evaluated the quality of the repairs produced by these techniques, finding that repairs often break existing functionality (Brun et al. 2013; Smith et al. 2015; Qi et al. 2015; Martinez et al. 2017). More recent techniques have produced higher-quality repairs that generalize to the intended specification (Long and Rinard 2015, 2016b; Ke et al. 2015).

However, while research has evaluated whether patches can be produced and the quality of those patches, prior work has not studied the defect characteristics that make automated repair more applicable. This paper evaluates the kinds of defects for which automated repair techniques produce patches, answering the question of whether existing techniques can repair important defects, or defects that are hard for developers to repair. The same way work on evaluating patch quality (Brun et al. 2013; Smith et al. 2015; Qi et al. 2015; Martinez et al. 2017; Pei et al. 2014) has led to work on improving repair quality (Long and Rinard 2015, 2016b; Ke et al. 2015), our intent is for this paper to lead to work on improving the applicability of automated repair to hard-to-fix and critical defects. Repair techniques have been evaluated in terms of how many defects they produce a patch for (e.g., Le Goues et al. (2012a), Ke et al. (2015), Qi et al. (2015), and Kim et al. (2013)), how quickly they produce patches (e.g., Le Goues et al. (2012a) and Weimer et al. (2013)), and the quality of the patches they produce, such as developer-judged correctness (Qi et al. 2015; Long and Rinard 2016b; Martinez et al. 2017), how many independent tests the patched programs pass (Brun et al. 2013; Smith et al. 2015; Jiang et al. 2016), how maintainable the patches are Fry et al. (2012), and how likely developers are to accept them (Kim et al. 2013). Researchers have stressed the need to identify the classes of defects for which repair techniques work well (Monperrus 2014); however, to the best of our knowledge, this is the first

attempt to relate the importance or difficulty of a developer fixing a defect to the success of automated repair.

The largest research challenge in determining how defect characteristics correlate with automated repair technique success is obtaining a large dataset of real-world defects, annotated with characteristics of defect importance and complexity. To that end, we annotate two widely used defect datasets — ManyBugs (Le Goues et al. 2015) consisting of 185 real-world defects in nine large C projects up to 2.8MLOC, and Defects4J (Just et al. 2014) consisting of 357 real-world defects in five large Java projects up to 96KLOC — and make these annotated datasets available for future evaluations.

Our study considers nine automated repair tools that implement seven automated repair techniques (two pairs of tools implement the same technique for different languages). Six of the techniques repair C programs: GenProg (Weimer et al. 2009; Le Goues et al. 2012a, b), TrpAutoRepair (Qi et al. 2013), AE (Weimer et al. 2013), Kali (Qi et al. 2015), SPR (Long and Rinard 2015), and Prophet (Long and Rinard 2016b). Three of the techniques repair Java programs: Nopol (DeMarco et al. 2014), a Java reimplementation of GenProg (Martinez et al. 2017), and a Java reimplementation of Kali (Martinez et al. 2017). We use results from prior evaluations of these techniques on the ManyBugs and Defects4J datasets. GenProg, TrpAutoRepair, and AE have been applied to all 185 ManyBugs defects (Le Goues et al. 2015). SPR, Prophet, and Kali have each been applied to 105 (a strict subset of the 185) ManyBugs defects (Long and Rinard 2015, 2016b; Qi et al. 2015). Nopol, and the Java versions of GenProg and Kali have been applied to 224 Defects4J defects (Martinez et al. 2017).

We identify and compute eleven unique abstract parameters recorded in most bug tracking systems and source code repositories that relate to five defect characteristics: importance, independence, complexity, test effectiveness, and characteristics of the developer-written patch. These parameters and characteristics form the basis of our evaluation, comprising the dataset and methodology that creators of new Java and C automated repair tools can use to evaluate their tools. Our evaluation answers six research questions:

RQ1 *Importance:* Is an automated repair technique's ability to produce a patch for a defect correlated with that defect's importance?

RQ2 *Complexity:* Is an automated repair technique's ability to produce a patch for a defect correlated with that defect's complexity?

RQ3 *Test effectiveness:* Is an automated repair technique's ability to produce a patch for a defect correlated with the effectiveness of the test suite used to repair that defect?

RQ4 *Independence:* Is an automated repair technique's ability to produce a patch for a defect correlated with that defect's dependence on other defects?

RQ5 *Characteristics of developer-written patches:* What characteristics of the developer-written patch for a defect are significantly associated with an automated repair technique's ability to produce a patch for that defect?

RQ6 *Patch quality:* What defect characteristics are significantly associated with an automated repair technique's ability to produce a high-quality patch for that defect?

We find that:

RA1 *Importance*: Java repair techniques are moderately more likely to produce patches for defects of a higher priority, while C repair techniques' ability to produce a patch for a defect does not correlate with defect priority. Further, Java and C repair techniques' ability to produce a patch for a defect has little to no consistent correlation

with the time taken by developer(s) to fix that defect and the number of software versions affected by that defect. This suggests that automated repair is as likely to produce a patch for a defect that takes developers a long time to fix as for a defect that developers fix quickly.

RA2 *Complexity*: C repair techniques are less likely to produce patches for defects that required developers to write more lines of code and edit more files to patch. However, the observed negative correlations are not consistently strong for all techniques, suggesting that automated repair can still produce patches for some complex defects. Further, for Java repair techniques, we do not observe a statistically significant relationship of this kind.

RA3 *Test effectiveness*: Java repair techniques are less likely to produce patches for defects with more triggering or more relevant tests, while C repair techniques' ability to produce a patch for a defect does not correlate with the number of triggering or relevant tests. Further, Java and C repair techniques' ability to produce a patch for a defect has little to no consistent correlation with the statement coverage of the test suite used to repair that defect.

RA4 *Independence*: Java repair techniques' ability to produce a patch for a defect does not correlate with that defect's dependence on other defects. For the C repair techniques, the data does not provide sufficient diversity to study the relationship between repairability and defect independence.

RA5 *Characteristics of developer-written patches*: Java and C repair techniques struggle to produce patches for defects that required developers to insert loops or new function calls, or change method signatures.

RA6 *Patch quality*: Only two of the considered repair techniques, Prophet and SPR, produce a sufficient number of high-quality patches to evaluate. These techniques were less likely to produce patches for more complex defects, and they were even less likely to produce correct patches.

The main contributions of this paper are:

- A methodology for identifying eleven abstract parameters that map onto five defect characteristics for evaluating automated repair applicability.
- The publicly-released, partially manual and partially automated annotation of 409 defects within ManyBugs and Defects4J, two large benchmarks widely used for evaluation of automated program repair.
- A methodology for evaluating the applicability of automated program repair techniques using the ManyBugs and Defects4J defect annotations to encourage research to develop techniques that repair more defects that are considered important and difficult for developers to repair.
- The evaluation of seven automated program repair techniques on 409 ManyBugs and Defects4J defects, identifying how defect characteristics correlate with the techniques' ability to produce patches, and high-quality patches.

Our extensions to the ManyBugs and Defects4J benchmarks and the scripts we have used to automate deriving the data for those benchmarks are available at https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/.

The rest of the paper is structured as follows. Section 2 describes the repair techniques and defect benchmarks we use in our evaluations. Section 3 outlines our experimental methodology. Section 4 discusses our results and their implications. Section 5 presents the threats to the validity of our analysis, and Section 6 places our work in the context of related

research. Finally, Section 7 summarizes our contributions and future research directions enabled by this work.

2 Subjects of Investigation

The goal of this paper is to evaluate if state-of-the-art automated repair techniques can generate patches for defects that are important or difficult for developers to fix. This section details the subjects our study investigated, in terms of the automated repair techniques, real-world defects, and the corresponding patch information. Section 3 explains the experimental methodology that uses these subjects.

2.1 Automated Repair Techniques

Our study considers seven state-of-the-art automated repair techniques.

GenProg (Weimer et al. 2009; Le Goues et al. 2012a, b) is a generate-and-validate repair technique. Such techniques create candidate patches, often using search-based approaches (Harman 2007), and then validate them, typically through testing. GenProg uses a genetic programming heuristic (Koza 1992) to search through the space of possible patches, mutating lines executed by failing test cases, either deleting them, inserting lines of code from elsewhere in the program, or both to create new potential patches, and crossover operators to combine patches. GenProg uses the test suite to select the best-fit patch candidates and continues evolving them until it either finds a patch that passes all tests, or until a specified timeout. GenProg targets general defects without focusing on a specific class. GenProg was originally designed for C programs (Weimer et al. 2009; Le Goues et al. 2012a, b), but has been reimplemented for Java (Martinez et al. 2017). This paper differentiates these two implementations as GenProgC and GenProgJ.

TrpAutoRepair (Qi et al. 2013) (also published under the name RSRepair in "The strength of random search on automated program repair" by Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang in the 2014 International Conference on Software Engineering; we refer to the original name in this paper) uses random search instead of GenProg's genetic programming to traverse the search space of candidate patches for C programs. It uses heuristics to select the most informative test cases first, and stops running the suite once a test fails. TrpAutoRepair limits its patches to a single edit. It is more efficient than GenProg in terms of time and test case evaluations (Qi et al. 2013). Similarly to GenProg, TrpAutoRepair targets general defects without focusing on a specific class.

AE (Weimer et al. 2013) is a deterministic program repair technique that uses heuristic computation of program equivalence to prune the space of possible repairs, selectively choosing which tests to use to validate intermediate patch candidates. AE uses the same change operators as GenProg and TrpAutoRepair, but rather than using a genetic or randomized search algorithm, AE exhaustively searches through the space of all non-equivalent k-distance edits. AE targets C programs, and, again, targets general defects without focusing on a specific class.

Kali (Qi et al. 2015) is a simple generate-and-validate repair technique that only deletes lines of code. It was originally designed to show that even this simple approach can sometimes produce patches that pass the available tests, but it has been shown that at times, these patches are of high quality (Qi et al. 2015; Martinez et al. 2017). Kali was originally designed for C programs (Qi et al. 2015), but has been reimplemented for Java (Martinez

et al. 2017). This paper differentiates these two implementations as KaliC and KaliJ. Kali targets defects that can be patched strictly by removing functionality.

SPR (staged program repair) (Long and Rinard 2015) is a generate-and-validate repair technique that uses a set of predefined, parameterized transformation schemas designed to generate repairs for specific defect classes. SPR targets defects that can be repaired by inserting or modifying conditional statements, initializing variables, replacing one variable with another, replacing one invoked function with another, replacing one constant with another, or inserts a statement from elsewhere in the program. Many of the schemas generate conditions by first computing constraints over specific variable values needed for a repair and then synthesizing logical expressions to satisfy those constraints. SPR uses the test suite to validate the patches, targets C programs, and has been shown to find higher-quality patches than GenProg (Long and Rinard 2015).

Prophet (Long and Rinard 2016b) explores single-edit potential patches, similarly to TrpAutoRepair, and SPR's transformation schemas. It prioritizes the schemas using models inferred from successful developer-written patches from open-source development. The class of defects Prophet targets is the same as SPR's. Prophet targets C programs and has been shown to find higher-quality patches than GenProgC, AE, SPR, and KaliC (Long and Rinard 2016b).

Nopol (DeMarco et al. 2014) is a synthesis-based technique that targets repairing conditional statements in Java programs. Using the test cases, Nopol generates Satisfiability Modulo Theory (SMT) constraints that describe the desired behavior on those test cases and uses an SMT solver to generate a conditional. Nopol fixes defects such as forgotten null pointer checks.

2.2 Defect benchmarks

We use two benchmarks of defects for our study, ManyBugs and Defects4J.

The **ManyBugs** benchmark (Le Goues et al. 2015) consists of 185 defect scenarios, summarized in the top of Fig. 1. Each scenario consists of a version of source code from one of nine large, open-source software systems, a set of project tests that fail on that version, a set of tests that pass on that version, and another version from a later point in the repository that passes all the tests. These defect properties allow for some defects to actually be features that modify system behavior. Out of 185 defects in ManyBugs, 29 defects were features and the remaining 156 defects were bugs. For 122 of the defects, ManyBugs includes accurate links to the project's bug tracking system or forums (though 8 of those links are no longer accessible), describing the defect.

The **Defects4J** benchmark (Just et al. 2014) consists of 357 defect scenarios, summarized in the bottom of Fig. 1. Similarly to ManyBugs, each scenario includes a version of source code with a defect, and a version with that defect repaired by a developer. The benchmark also includes, for each defect, a developer-written test suite that includes at least one triggering test. As Section 2.3 describes, 224 of these defects have been used for automated repair, and so we consider that 224-defect subset. Of these, 205 have links to the project's bug tracking system. We found that 4 out of the 224 defects were features and the remaining 220 defects were bugs.

Benchmark Extensions As part of this work, we augmented ManyBugs and Defects4J with extra information. We annotated every defect with the number of lines of code in the minimized, developer-written patch, the number of files that patch touches, the number of relevant test cases (test cases that execute at least one statement in at least one file edited

program	kLoC	defects	program description
ManyBugs			
fbc	97	3	legacy language compiler
gmp	145	2	multi-precision math library
gzip	491	5	data compression utility
libtiff	77	24	image processing library
lighttpd	62	9	web server
php	1,099	104	web programming language
python	407	15	general-purpose language
valgrind	793	15	dynamic debugging tool
wireshark	2,814	8	network packet analyzer
total	**5,985**	**185**	
Defects4J			
JFreeChart	96	26	chart drawing library
Closure	90	133	compiler
Commons Lang	22	65	Apache core library
Commons Math	85	106	Apache math and stat library
Joda-Time	28	27	date and time library
total	**321**	**357**	

Fig. 1 Our study uses the ManyBugs and Defects4J benchmarks. The 185 ManyBugs defects come from nine open-source software systems (Le Goues et al. 2015), and the 357 Defects4J defects come from five open-source software systems (Just et al. 2014). The `Closure` defects are excluded from our study because prior studies have not used them to evaluate automated repair techniques (Martinez et al. 2017)

by the developer-written patch) as well as test cases that trigger the defect and, the test suite coverage. We have also annotated the 114 ManyBugs and 205 Defects4J defects with links to their projects' bug tracking systems or forums with how much time passed between when the defect was reported and when it was resolved, the priority of the defect, the number of project versions impacted by the defect, and the number of dependent defects. Section 3.2 will describe in more detail how we collected these data. These annotations enable our evaluation, and can enable others to evaluate how defect characteristics correlate with the applicability of their repair techniques. The annotations are available at https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/.

2.3 Repairability Information

Both ManyBugs and Defects4J have been used to evaluate automated program repair in the past (Martinez et al. 2017; Le Goues et al. 2012b, 2015; Long and Rinard 2015, 2016b; Qi et al. 2015). GenProgC, TrpAutoRepair, and AE have been applied to all 185 ManyBugs defects (Le Goues et al. 2015). SPR, Prophet, and KaliC have each been applied to a 105-defect subset of the 185-defect ManyBugs benchmark (Long and Rinard 2015, 2016b; Qi et al. 2015). Nopol, GenProgJ, and KaliJ have been applied to 224 Defects4J defects (Martinez et al. 2017). In our evaluation, we use these results for the 409 defects, in terms of which techniques can produce a patch for which defects. Figure 2 summarizes these results. Combined, the techniques repair 56% and 49% of C defects from the two sets of C defects and 21% of the Java defects, denoted by the "$\bigcup$C" and "$\bigcup$Java" rows. We believe the difference in these numbers is due in part to the fact that we study more techniques that target C. Additionally, the first C-targeting techniques (Weimer et al. 2009) predate the Java-targeting ones, and the first versions of ManyBugs (Le Goues et al. 2012a) predate the first version of Defects4J (Just et al. 2014), so researchers have had more time to improve their techniques for ManyBugs than for Defects4J.

technique	patched	unpatched	patched %
ManyBugs			
GenProgC	87	98	47%
TrpAutoRepair	97	88	52%
AE	86	99	46%
⋃C	**104**	**81**	**56%**
ManyBugs (105-defect subset)			
KaliC	27	78	26%
SPR	46	59	44%
Prophet	43	62	41%
⋃C	**52**	**53**	**49%**
Defects4J			
GenProgJ	27	197	12%
KaliJ	22	202	10%
Nopol	35	189	16%
⋃Java	**47**	**177**	**21%**

Fig. 2 The automated repair techniques we consider that have been evaluated on the entire 185-defect ManyBugs patched 56% of those C defects; the techniques that have been evaluated on the 105-defect subset of ManyBugs patched 49% of those C defects; the techniques that have been evaluated on 224-defect Defects4J patched 21% of those Java defects. The "⋃C" and "⋃Java" rows give the numbers and ratios of defects for which at least one of the C- and Java-targeting techniques could generate a patch

Further, research studying the quality of repair has identified, via manual analysis and judgment, which subset of the defects have correct patches (and which are "plausible but incorrect") for GenProgC, TrpAutoRepair, and AE (Qi et al. 2015), SPR, Prophet, and KaliC (Long and Rinard 2016b), and GenProgJ, Nopol, and KaliJ (Martinez et al. 2017). We use these data in answering RQ6 on patch quality. Because far fewer of the defects are automatically repaired correctly than plausibly, we expect the statistical power of these measurements to be less significant.

3 Methodology

This section explains our data collection procedure and the experimental methodology we use to answer the research questions posed in Section 1. We first use grounded theory to identify and classify relevant information available in bug tracking systems, open-source project hosting platforms, and defect benchmarks (Section 3.1). We use this process to identify five defect characteristics: *defect importance*, *defect complexity*, *test effectiveness*, *defect independence*, and *characteristics of the developer-written patch*. Our study analyzes the relationship between *repairability* — an automated repair technique's ability to generate a patch for a given defect — and these five defect characteristics. We further analyze the relationship between an automated repair technique's ability to produce correct patches and each of these characteristics.

Our study of relevant data available in bug tracking systems, open-source project hosting platforms, and defect benchmarks is somewhat idealistic. It relies on all data that could be available from these sources. Of course, in the real world, not all data sources are available for each bug. For the defects we study (recall Section 2.2), only partial information is available for some of the defects, e.g., only some of the defects contain links to issues in bug tracking systems. Section 3.2 describes how we annotated defects of ManyBugs and Defects4J with the identified defect characteristics.

We describe the methodology here with the assumption that the characteristics are independent. In Section 4.8.3, we relax this assumption, present a methodology to check for correlations and confounding factors among the characteristics, and apply that methodology to our evaluation.

3.1 Identifying importance and difficulty data

Classifying how difficult a defect is to repair, and how important repairing a defect is to a project is a complex and subjective task. There is neither a single measure of difficulty nor of importance. To identify aspects of defects related to difficulty and importance, we first analyzed eight popular bug tracking systems (softwaretestinghelp.com 2015), three popular open-source project hosting platforms with bug tracking systems, and two benchmarks of software defects (that include source code, test suites, and developer-written patches) (Le Goues et al. 2015; Just et al. 2014).

We used constructivist grounded theory (Bryant and Charmaz 2007) with coding and constant comparison specifically designed for reasoning about and categorizing concepts without preconceived abstractions of the involved data (Charmaz 2006). In other words, we started out without having strong preconceived notions of what data found in bug tracking systems, open-source project hosting platforms, and defect benchmarks are likely to be relevant to defect difficulty and importance, and we used the appropriate grounded theory for identifying such data and classifying them into abstractions. This methodology has been previously recommended for use in information systems research (Matavire and Brown 2013). Two of the authors, called coders, independently analyzed all *concrete parameters* available in the bug tracking systems, open-source project hosting platforms, and the defect benchmarks (specifically focusing on the test suites and developer-written patches available in these benchmarks). The coders selected which pieces of data may be relevant to how difficult or important the defect is to repair. For example, concrete parameter *priority* was associated with the importance of defect while *# of lines in the minimized patch for a defect* was associated with difficulty. Coders also identified other data such as number of triggering and relevant tests available for a defect and number of project versions affected by a defect that they felt may be interesting to correlate with automated repair techniques' ability to repair the defect. The two coders then compared their coding and reconciled the differences. Reconciling sometimes required looking at example defects to gain a further insight into the semantics of the concrete parameters.

Next, the coders, again independently, grouped similar concrete parameters (e.g., identical parameters for which different bug tracking systems use different names, or closely related concrete parameters) into *abstract parameters*. For example concrete parameters such as *components*, *linked entities*, *affects versions* and *fix versions* were grouped together to form an abstract parameter *versions*. Again, the coders reconciled their coding. The coders iterated between identifying concrete and abstract parameters until their findings saturated and no more parameters were identified. At the end of analysis, we had created eleven abstract parameters.

Finally, the coders, again independently, categorized the abstract parameters by grouping closely related parameters, and then reconciled their coding. We call these categories *defect characteristics*. For example, abstract parameters *File count*, *Line Count* and *Reproducibility* were grouped together to form a defect characteristic *Complexity*. Similarly abstract parameters *Statement coverage*, *Triggering test count* and *Relevant test count* were grouped

together to form the defect characteristic *Test-Effectiveness*. We came up with five such defect characteristics using eleven abstract parameters.

The eight popular bug tracking systems we used are Bugzilla, JIRA, IBM Rational ClearQuest, Mantis, Trac, Redmine, HP ALM Quality Center, and FogBugz (softwaretestinghelp.com 2015). The three popular open-source project hosting platforms with bug tracking systems we used are Sourceforge, GitHub, and Google code (although the latter is no longer active). Finally, the two benchmarks of software defects we used are a 185-C-defect ManyBugs (Le Goues et al. 2015) and a 357-Java-defect Defects4J (Just et al. 2014) benchmarks (recall Section 2.2). For completeness and reproducibility of our research, we include a complete list and description of the concrete parameters we found for each of the bug tracking systems, project-hosting platforms, and defect benchmarks in Table 1 in Appendix A. Note that the names of the parameters bug tracking systems use are not always intuitive, and sometimes inconsistent between systems. For example, Google code uses the terms "open" and "closed" for timestamps of when an issue was open or closed. GitHub uses these terms as binary labels. Google code uses the parameter "status" to encode these labels. We do not include a detailed description of what information each parameter encodes and how it encodes it in this paper, but this information is available from the underlying bug tracking systems and project-hosting platforms.

The coders grouped these concrete parameters into eleven abstract parameters, and then, categorized the abstract parameters into the five defect characteristics. Again, for completeness and reproducibility, we include the complete mapping of concrete parameters to abstract parameters, and, in turn, to the defect characteristics in Table 2 in Appendix A.

Sections 3.1.2–3.1.6 describe the five defect characteristics and the eleven abstract parameters that map onto them. But first, Section 3.1.1 describes the statistical tests we use to determine whether the abstract parameters correlate with the automated repair techniques' ability to produce patches.

3.1.1 Statistical tests

Ten out of the eleven abstract parameters are numerical. For most parameters, the number of unique values is small and the magnitude of their difference may not be indicative. Therefore, we rely on nonparametric statistics and do not assume that the underlying values of a parameter should be interpreted as equidistant from one another. Specifically, for each technique (including "$\bigcup$C" and "$\bigcup$Java," as described in Section 2.3), for each of the ten numerical abstract parameters, we split the distribution of that parameter's values into two distribution samples: (1) the distribution of the parameter's values for the defects for which the technique produces a patch, and (2) the distribution of the parameter's values for the defects for which the technique does not produce a patch. We use the nonparametric Mann-Whitney U test to determine if the two distribution samples are statistically significantly different. That is, the test computes a p value that indicates whether the null hypothesis that the two distribution samples are statistically indistinguishable (i.e., they are drawn from the same distribution) should be rejected. We use the standard convention of $p \leq 0.01$ to mean the difference is strongly statistically significant, $0.01 < p \leq 0.05$ to mean the difference is statistically significant, $0.05 < p \leq 0.1$ to mean the difference is weakly statistically significant, and $p > 0.1$ to mean the difference is not statistically significant. We measure the strength of the association between the abstract parameter and the technique's ability to produce a patch using the rank-biserial correlation coefficient, a special case of

Somers' d. A defect's repairability with respect to a technique — whether the technique produces a patch for this defect — is a dichotomous variable, and in such situations, Somers' d is a recommended measure of the nonparametric effect size for ordinal data; Somers' d is also asymmetric with the presumed cause and effect variables, which is the case in our study (Ferguson 2009; Newson 2002). We use the standard mapping from Somers' d (which can take on values between -1 and 1) to the adjectives very weak ($|d| < 0.1$), weak ($0.1 \leq |d| < 0.2$), moderate ($0.2 \leq |d| < 0.3$), and strong ($0.3 \leq |d|$) (Le Roy 2009). We further compute the 95% confidence interval for Somers' d (referred to as 95% CI). We consider an association to be statistically and practically significant if it is at least weakly statistically significant and if the 95% confidence interval for Somers' d is entirely positive or entirely negative. For the eleventh abstract parameter, developer-written patch characteristics, we use a logistic regression to fit a model for repairability and determine which characteristics have the strongest effect on repairability (Section 3.1.6).

3.1.2 Defect Importance

Our analysis of the eleven issue tracking systems (eight bug trackers and three project-hosting platforms) identified three common abstract parameters related to importance: priority, project versions affected, and time to fix the defect.

Priority of the Defect The priority of a defect is included in nine out of the eleven issue tracking systems. Different issue tracking systems use different ordinal scales to measure the priority. We mapped these different scales to our own scale that varies from 1 (lowest priority) to 5 (highest priority). Our study determines if there is a significant association between priority and repairability using the nonparametric Mann-Whitney U test and measures the strength of the association using a rank-biserial correlation coefficient Somers' d. In other words, as described in Section 3.1.1, we compare the distribution of priorities of defects patched by an automated repair technique to the distribution of priorities of defects not patched by an automated repair technique. We use the Mann-Whitney U test to test if the distributions are statistically significantly different, and we measure the magnitude of that difference using Somers' d.

Does the Defect Affect More Than One Project Version? The number of project versions a defect affects is included in three out of the eleven issue tracking systems. Our study uses the Mann-Whitney U test to check for a significant association with repairability, and it measures the strength of the association using Somers' d.

Time Taken to Fix the Defect The time between when a defect was reported and when it was resolved is included in eight out of the eleven issue tracking systems. For those systems that did not have any concrete parameters to indicate the time when defect was resolved, we approximated it by computing the time between timestamps of when the issue was entered into an issue tracking system, and the last commit for the issue. Our study determines if there is a significant association between time to fix and repairability using the Mann-Whitney U test, and it measures the strength of the association using Somers' d. Note that time to

fix a defect could be considered as a parameter for importance or difficulty, but based on our analysis of defects and experience of commits made by developers, associating this parameter to the importance characteristic seem more accurate.

3.1.3 Defect Complexity

Our study considers two defect complexity measures: number of files edited by the developer-written patch, and number of non-blank, non-comment lines of code in that patch. These patches are manually minimized in the Defects4J benchmark to remove all changes that do not contribute to the patch's goal (Just et al. 2014), but are not in the ManyBugs benchmark. We partially minimized the ManyBugs patches by removing all blank and comment lines to reduce the potential bias due to over-approximating the number of files and lines of code. We employed the `diffstat` tool to automatically compute the number of source code lines affected by the partially minimized patches.

Number of Source Files Edited by the Developer-Written Patch A defect that requires editing multiple source files might be harder to localize and generally more difficult to repair. Our study investigates the effect of the number of source files edited by a patch on a defect's repairability. It determines if there is a significant association between the number of files edited and repairability using the Mann-Whitney U test. It measures the strength of the association using Somers' d.

Number of Non-Blank, Non-Comment Lines of Code in Developer-Written Patch Our study also investigates if defects with larger developer-written patches, in terms of lines of code, are more difficult for automated repair techniques to repair. Our study determines whether the number of lines of code has a significant association with repairability using the Mann-Whitney U test, and it measures the strength of the association using Somers' d.

3.1.4 Test Effectiveness

Prior work (Smith et al. 2015) suggests that test effectiveness might have an effect on an automated repair technique's ability to generate a patch. Our study identified three parameters related to test effectiveness: (1) the fraction of the lines in the files edited by the developer-written patches that are executed by the test suite, (2) the number of defect-triggering test cases, and (3) the number of relevant test cases (test cases that execute at least one line of the developer-written patch). Our study determines, for each parameter, if it has a significant association with repairability using the Mann-Whitney U test, and it measures the strength of the association using Somers' d.

3.1.5 Defect Independence

Our analysis of eleven issue tracking systems identified dependents as a common abstract parameter related to difficulty: a defect whose repair depends on another issue in the issue tracking system might be more difficult to repair than a defect that can be repaired independently. The information about defect dependents is included in five out of the eleven issue tracking systems. Our study uses the Mann-Whitney U test to check for a significant association with repairability, and it measures the strength of the association using Somers' d.

3.1.6 Developer-Written Patch Characteristics

The ManyBugs benchmark provides annotations for each defect, including characteristics of its developer-written patch (Le Goues et al. 2015). The nine characteristics describe if the developer-written patch:

C1 changes one or more data structures or types
C2 changes one or more method signatures
C3 changes one or more arguments to one or more functions
C4 adds one or more function calls
C5 changes one or more conditionals
C6 adds one or more new variables
C7 adds one or more if statements
C8 adds one or more loops
C9 adds one or more new functions

We manually annotated the Defects4J defects using the same characterization of the developer-written patches.

More than one patch characteristic might apply to a single defect. Our study considers each patch characteristic as a dichotomous variable and uses a logistic regression to fit a model for repairability. It furthermore determines the patch characteristics that have the strongest effect on repairability.

3.2 Characterizing the ManyBugs and Defects4J Data

The defect benchmarks we study (recall Section 2.2), provide partial information for some of the defects, e.g., only some of the defects contain a link to an issue in the issue tracking system. This section describes the information we were able to obtain for these defects, to approximate the idealized methodology described in Section 3.1.

Recall that each defect in the ManyBugs and Defects4J benchmarks corresponds to a pair of commits in a version control system, but not necessarily to an issue in an issue tracking system. For each defect in Defects4J, we tried to manually determine the corresponding issue in the issue tracking system by cross-referencing the commit logs and commit IDs with the commit information in the issue tracking system. For ManyBugs, the information about the issues in the issue tracking system which are associated with a defect was available for 122 out of the 185 defects (though 8 are no longer accessible because either the URL did not resolve or the issue was private). For Defects4J, this information was available for 205 out of the 224 defects. We annotated each defect with a link to the issue tracking system, with the abstract parameters recorded in the issue tracking system. The abstract parameters recorded were obtained from different concrete parameters depending on the issue tracking system used by a given project. Also, information about some of the abstract parameters was not found in some of the issue tracking systems. Hence we couldn't annotate all the defects with all the abstract parameters. Table 3 in Appendix B shows the details about information available for annotating the defects with parameters obtained from issue tracking systems.

An issue in an issue tracking system might apply to more than one defect: a fault reported in the issue tracking system might be related to multiple defects. Our study treats all defects related to the same issue in the issue tracking system as equally important. Section 5 discusses this experimental design choice and its implications. For the ManyBugs and Defects4J benchmarks, 312 out of 327 issues are related to exactly one defect.

For the abstract parameters that were obtained from the two defect benchmarks, we were able to annotate all defects with *line count*, *file count*, *triggering test count*, and *relevant test count* as this information was available with the benchmarks. The triggering test count is the number of negative tests for a defect provided in ManyBugs and number of triggering tests for a defect provided in Defects4J. The relevant test count in Defects4J is the number of test cases that execute at least one statement in at least one file edited by the developer-written patch. These are provided as relevant tests for each defect in Defects4J. ManyBugs provides all test cases that are relevant for the project, but these may not be specific to patched file(s). The relevant test count for ManyBugs is the number of all tests relevant for the project.

We annotated each defect in Defects4J with the statement coverage ratio of the test suite on the file(s) edited by the developer-written patch, using the `coverage` utility provided by the Defects4J framework. For ManyBugs, we used the `gcov` tool to compute this information for all the defects except for 52 defects that we could not compile. Figure 16 in Appendix B shows the number of defects that could be annotated with each abstract parameter.

While analyzing the defects in ManyBugs and Defects4J, we found that some of the defects were actually features. We classified all the defects and found that 29 out of the 185 ManyBugs defects were features while the remaining 156 were bugs, and that 4 out of the 224 Defects4J defects were features while the remaining 220 were bugs. To make this classification, we manually analyzed the issue description and discussion in the issue tracking system (when this information was available) to identify if the issue directly related to implementing new functionality, extending or enhancing functionality, or conforming to a standard. For those issues that satisfied that criterion, we then checked if the issue related to an unexpected output; if it did, we classified it as a bug. We then analyzed the minimized, developer-written source code changes made to resolve the issue for the issues not already classified as bugs to verify that the changes were consistent with the issue description and discussion, leading to the final classification as a feature. For issues without links to an issue tracking system, we followed the same procedure using the developer-written log messages and source code changes.

We considered two potential confounding factors that could affect repairability: (1) the defect type (if the defect relates to a bug report or a feature request), and (2) whether a defect links to an issue in an issue tracking system. The purpose of this analysis was to determine if our study needs to control for these factors. We used Fisher's exact test to test for independence. Section 4.8.3 details our inter-correlation analyzes for confounding factors that result from potential correlations between defect characteristics. Figure 3 shows that the repairability results are not independent of the defect type for ManyBugs, and hence our study controls for this factor by analyzing bug reports and feature requests separately. Fisher's exact test confirmed that there is no significant association between repairability and whether a defect links to an issue in an issue tracking system ($p = 0.64$ for ManyBugs; $p = 1.0$ for the Defects4J subject Chart, the only subject in Defects4J with some missing issue links). Therefore, our study does not control for this factor.

Our study uses the complexity of the developer-written patches as a proxy for defect complexity, instead of inferring complexity from the issue tracking systems for two reasons. First, a developer-written patch is available for every defect in the ManyBugs and Defects4J benchmarks — by contrast, only 327 out of 409 defects have a corresponding issue in an issue tracking system. Second, the defect complexity recorded in an issue tracking system may be subjective or specific to the project. Note that while a defect might, in theory, have an unbound number of valid fixes, we assume that the developer-written fix is indicative of the complexity of the defect it fixes. Section 5 discusses this assumption in greater detail.

ManyBugs			
defect type	patched	unpatched	total
bug	105	51	156
feature	14	15	29
column total	119	66	185
Fisher's exact test: $p = 0.05$			

Defects4J			
defect type	patched	unpatched	total
bug	46	174	220
feature	1	3	4
column total	47	177	224
Fisher's exact test: $p = 1.00$			

Fig. 3 The effect of a defect's type (bug or feature) on its repairability

4 Results

This section answers the six research questions from Section 1 for the automated repair techniques and defects from Section 2, using the methodology from Section 3. As we have already described, we consider the defects classified into bugs and features separately, so we answer the research questions for these two sets of defects separately.

Sections 4.1–4.6 present results for our six research questions, Section 4.7 comments on feature synthesis, and Section 4.8 discusses the implications of our results, relates them to prior findings, and considers confounding factors.

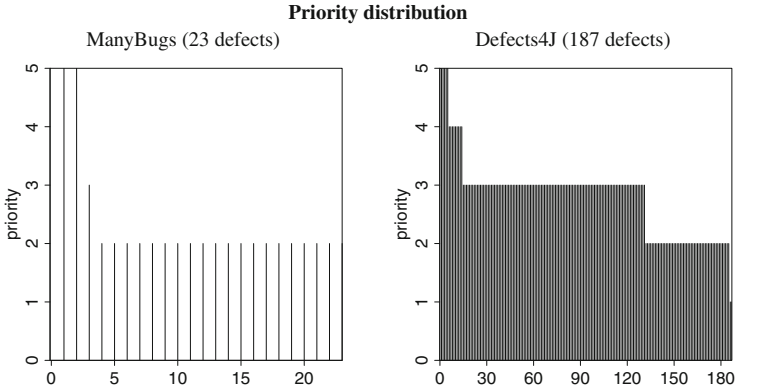
4.1 Defect Importance

RQ1: Is an automated repair techniques ability to produce a patch for a defect correlated with that defects importance?

We measure a defect's importance using the defect's *priority* value in the project's issue tracking system, the number of project versions the defect affects, and the time from when the defect was reported until it was resolved (recall Section 3).

Priority of the Defect For 156 defects in ManyBugs that were classified as bugs, 23 defects had priority values: 20 have priority two, 1 has priority three, and 2 have priority five. For 220 defects in Defects4J that were classified as bugs, 187 defects had priority values: 2 have priority one, 54 have priority two, 117 have priority three, 9 have priority four, and 5 have priority five. Top of Fig. 4 shows the distribution of the priority values.

Figure 4 shows the results of the Somers' d and the Mann-Whitney U tests comparing the priority distributions of defects for which techniques do, and do not produce patches. For the Java techniques, Somers' d indicates moderate to strong positive correlations between priority and repairability. For Nopol and $\bigcup$Java, the Mann-Whitney U test indicates a statistically significant difference between the distributions ($p \leq 0.05$) and the Somers' d 95% confidence interval is entirely positive. While we observed a weakly significant, moderate positive correlation ($p \leq 0.1$) for GenProgJ and KaliJ, our confounding factor analysis (Section 4.8.3) suggests that this observation was due to a correlation between the priority and the number of relevant test cases.



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	0.100	[−0.299, 0.499]	0.856	5	18
TrpAutoRepair	0.167	[−0.147, 0.480]	0.348	9	14
AE	−0.176	[−0.341, −0.012]	0.539	6	17
⋃C	0.114	[−0.165, 0.392]	0.342	11	12
ManyBugs (105-defect subset)					
SPR	−0.111	[−0.541, 0.319]	0.753	9	5
KaliC	−0.111	[−0.541, 0.319]	0.753	9	5
Prophet	−0.222	[−0.472, 0.027]	0.505	5	9
⋃C	−0.175	[−0.679, 0.329]	0.780	10	4
Defects4J					
GenProgJ	0.206	[−0.037, 0.449]	0.088	20	167
Nopol	**0.307**	[**0.133**, **0.481**]	**0.002**	**28**	**159**
KaliJ	0.223	[−0.023, 0.468]	0.080	16	171
⋃Java	**0.216**	[**0.065**, **0.367**]	**0.017**	**37**	**150**

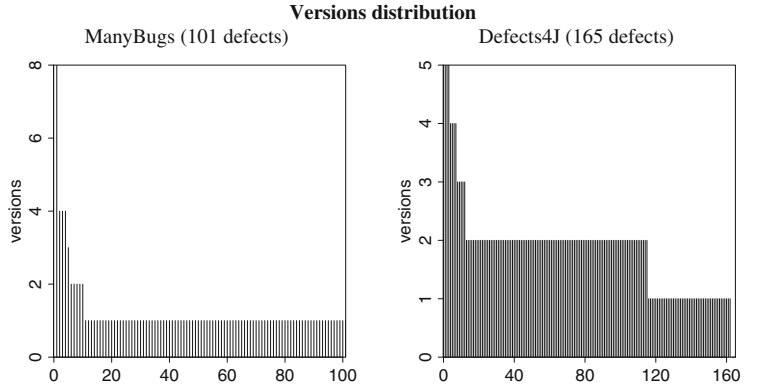
Fig. 4 Priority data are available for 23 ManyBugs and 187 Defects4J defects classified as bugs. Java repair techniques are more likely to produce a patch for defects with a higher priority. Insufficient data for C defects prevent a statistically significant conclusion. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for defects classified as bugs and with known priority values

For C techniques, we observe weak to moderate, both positive and negative correlations. Because relatively few (23) of the C defects have priority values, the Mann-Whitney U test does not find any statistically significant differences between the priority distributions of defects for which techniques do, and do not produce patches. Therefore we make no claims about a correlation between priority and C techniques' ability to produce patches.

Does the Defect Affect more than one Project Version? For 156 defects in ManyBugs that were classified as bugs, 101 defects had information on how many versions they affect. Of these, 91 affected a single version, 5 affected two versions, 1 affected 3 versions, 3 affected four versions, and 1 affected eight versions. For 220 defects in Defects4J that were classified as bugs, 165 defects had this information. Of these, 50 affected a single version, 103 affected two versions, 5 affected three versions, 4 affected four versions, and 3 affected five versions. Top of Fig. 5 shows the distribution of the versions values.

Figure 5 shows the results of the Somers' d and the Mann-Whitney U tests on the versions distributions of defects for which techniques do, and do not produce patches. We found no evidence of a relationship between a defect's repairability and the number of versions it depends on except for GenProgC and AE, which showed a significant ($p \leq 0.05$) and weakly significant ($p \leq 0.1$), respectively, negative correlation with the number of versions affected by a defect. For all other techniques, the results were insignificant ($p > 0.1$ or the 95% confidence interval spanned zero). We conclude that the number of versions affected by a defect likely has negligible effect on automated repair's effectiveness producing a patch for that defect.

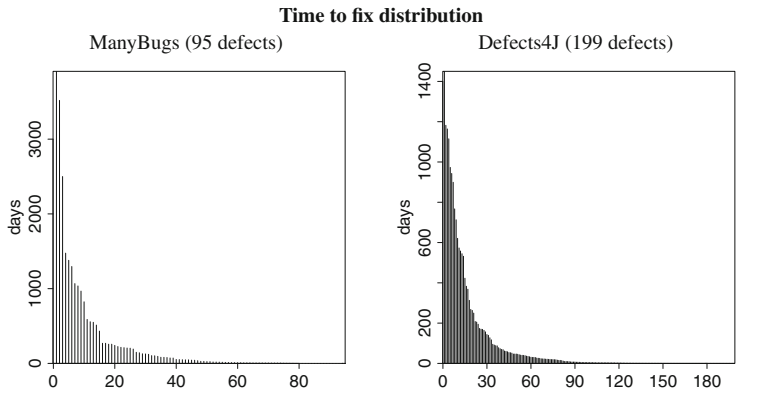
Time Taken to Fix the Defect For 156 defects in ManyBugs that were classified as bugs, 95 defects had information about the time frame that passed between when the defect was reported and when it was resolved. This time to fix the defect varied from 43 minutes to 10.7



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	**−0.143**	**[−0.250, −0.037]**	**0.017**	**47**	**54**
TrpAutoRepair	−0.095	[−0.215, 0.025]	0.134	54	47
AE	−0.113	[−0.225, −0.001]	0.078	49	52
⋃C	−0.075	[−0.201, 0.051]	0.262	59	42
ManyBugs (105-defect subset)					
SPR	−0.120	[−0 366, 0.126]	0.338	25	17
KaliC	0.021	[−0 205, 0.246]	0.882	23	19
Prophet	0.003	[−0 250, 0.255]	1.000	12	30
⋃C	−0.029	[−0 266, 0.209]	0.965	26	16
Defects4J					
GenProgJ	0.022	[−0.208, 0.252]	0.871	18	147
Nopol	0.175	[−0.009, 0.358]	0.096	27	138
KaliJ	−0.032	[−0.264, 0.199]	0.852	14	151
⋃Java	0.105	[−0.071, 0.282]	0.267	35	130

Fig. 5 Versions data are available for 101 ManyBugs and 165 Defects4J defects classified as bugs. Number of versions affected by a defect likely has little effect on automated repair's effectiveness producing a patch for that defect. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for those defects classified as bugs and with known versions values

years. Out of these 95 defects, 48 have time to fix of less than one month, 32 from one month to one year, and 15 greater than one year. For 220 defects in Defects4J that were classified as bugs, 199 had this information. The time to fix varied from 1 minute 21 seconds to 4.0 years. Out of 220 defects, 140 have time to fix of less than one month, 42 from one month to one year, and 17 greater than one year. The defects with a low time to fix may exemplify a source of potential noise in our data. While it is rare for the developer(s) to repair a defect in a minute and a half, they will sometimes discover a defect, think about the correct way to repair it, and even write relevant code before reporting the defect to the issue tracking system. In such cases, our methodology for measuring the time to fix will not capture the time the developer(s) spent thinking about the defect prior to reporting it. Unfortunately, this time is not recorded in the various surviving artifacts. However, this situation most likely pertains only to defects that the developer(s) fix quickly, thus correctly capturing the rank of the defects' time to fix measure. Top of Fig. 6 shows the distribution of the time to fix values.



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	−0.231	[−0.459, −0.003]	0.053	45	50
TrpAutoRepair	−0.067	[−0.304, 0.170]	0.576	51	44
AE	−0.128	[−0.360, 0.105]	0.287	45	50
⋃C	−0.112	[−0.349, 0.126]	0.357	55	40
ManyBugs (105-defect subset)					
SPR	−0.175	[−0.552, 0.202]	0.382	20	16
KaliC	−0.152	[−0.532, 0.229]	0.447	19	17
Prophet	−0.095	[−0.524, 0.335]	0.688	9	27
⋃C	−0.130	[−0.513, 0.253]	0.521	21	15
Defects4J					
GenProgJ	−0.294	[−0.590, 0.002]	0.024	22	177
Nopol	−0.173	[−0.390, 0.043]	0.137	29	170
KaliJ	−0.131	[−0.381, 0.119]	0.364	18	181
⋃Java	**−0.226**	**[−0.411, −0.040]**	**0.029**	**39**	**160**

Fig. 6 Time to fix data are available for 95 ManyBugs and 199 Defects4J defects classified as bugs. Time taken by developer(s) to fix a defect likely has little effect on automated repair's effectiveness producing a patch for that defect. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for those defects classified as bugs and with known time to fix values

Figure 6 shows the results of the Somers' d and the Mann-Whitney U tests on the time taken to fix distributions of defects for which techniques do, and do not produce patches. For every technique, Somers' d indicates a negative correlation: the longer it took for developers to repair a defect, the harder it is for automated repair techniques to produce a patch. However, these results are not statistically significant as indicated by the Mann-Whitney U test, except for GenProgC ($p \leq 0.1$) and GenProgJ ($p \leq 0.05$). For all other techniques, $p > 0.1$. We conclude that the time taken by the developer(s) to fix a defect likely has little effect on automated repair's effectiveness producing a patch for that defect.

> These results indicate that Java automated repair techniques are moderately more likely to produce patches for defects of a higher priority, while C techniques do not correlate with defect priority. The time taken by the developer(s) to fix a defect and number of software versions affected by the defect had little to no correlation with the ability to produce a patch. Overall, there is evidence that automated repair is as likely to repair more important defects, as it is to repair less important ones, which is an encouraging finding.

4.2 Defect Complexity

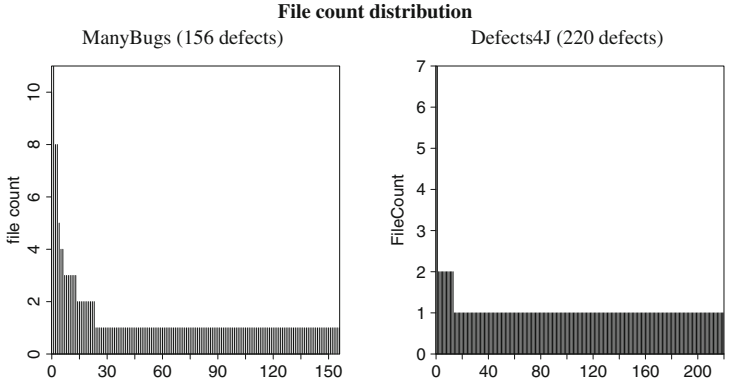
RQ2: Is an automated repair techniques ability to produce a patch for a defect correlated with that defects complexity?

We measure a defect's complexity using two parameters, the number of files containing non-comment, non-blank-line edits in the developer-written fix, and the total number of non-comment, non-blank lines of code in the developer-written fix (recall Section 3).

Number of Source Files Edited by the Developer-Written Patch The information on the number of files edited by the developer patch was available for all 185 defects in ManyBugs. The number of files varied from 1 to 11. The distribution of the 156 ManyBugs defects that were classified as bugs also varied from 1 to 11: 133 edited a single file, 10 two files, 7 three files, 2 four files, 1 five files, 2 eight files, and 1 eleven files. For Defects4J too, all 224 defects had this information. The number of files varied from 1 to 7. Of the 220 defects classified as bugs, 205 edited a single file, 12 two files, 1 three files, 1 four files, and 1 seven files. Top of Fig. 7 shows the distribution of the number of files edited values.

For C techniques, Somers' d showed a weak to moderate negative correlation between the number of files the developer-written patch edited, and the techniques' ability to produce a patch (Fig. 7). The Mann-Whitney U test showed this relationship to be statistically significant ($p \leq 0.05$) for all C techniques. The correlation was also negative for Java repair techniques, although this relationship was very weak and statistically insignificant ($p > 0.1$). We suspect the relatively weaker correlation for Java programs is due to the lower variability in Defects4J in the number of files edited by the developer patch.

Number of Non-Blank, Non-Comment Lines of Code in Developer-Written Patch For the 185 ManyBugs defects, the number of non-comment, non-blank lines in the developer-written patches varied from 1 to 1,887. In the subset of ManyBugs consisting of 156 bugs, the number varied from 1 to 1,341. For Defects4J, for both the 224 defects and the subset consisting of 220 bugs, the number of non-comment, non-blank lines in the developer-written patches varied from 1 to 49. Top of Fig. 8 shows the distribution of the number of lines edited values.

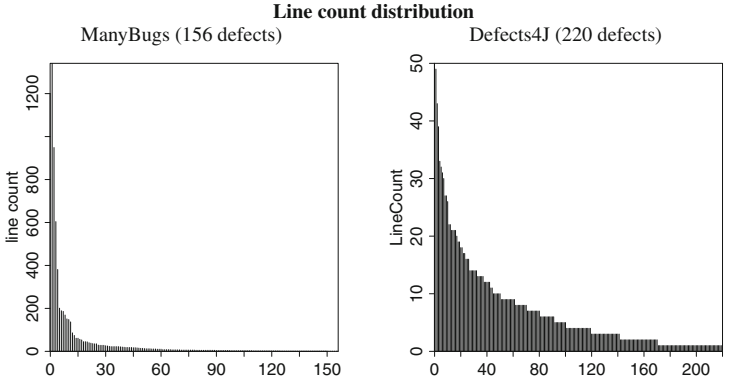


technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	**−0.134**	[**−0.241**, **−0.026**]	**0.016**	**75**	**81**
TrpAutoRepair	**−0.117**	[**−0.231**, **−0.002**]	**0.042**	**84**	**72**
AE	**−0.134**	[**−0.241**, **−0.026**]	**0.016**	**75**	**81**
⋃C	**−0.147**	[**−0.268**, **−0.025**]	**0.011**	**91**	**65**
ManyBugs (105-defect subset)					
SPR	**−0.256**	[**−0.407**, **−0.105**]	**0.002**	**41**	**43**
KaliC	**−0.191**	[**−0.343**, **−0.039**]	**0.028**	**39**	**45**
Prophet	−0.186	[−0.297, −0.075]	0.062	22	62
⋃C	**−0.233**	[**−0.396**, **−0.070**]	**0.007**	**44**	**40**
Defects4J					
GenProgJ	−0.067	[−0.090, −0.045]	0.232	27	193
Nopol	−0.035	[−0.098, 0.027]	0.521	34	186
KaliJ	−0.066	[−0.085, −0.047]	0.372	22	198
⋃Java	−0.047	[−0.100, 0.005]	0.298	46	174

Fig. 7 Number of files in the developer-written patch data are available for all 156 ManyBugs and 220 Defects4J defects classified as bugs. Automated program repair is less likely to produce patches for defects whose developer-written patches edit more files. This result is strongly statistically significant for C repair techniques, but is statistically insignificant for Java repair techniques. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for those defects classified as bugs

For C techniques, Somers' d showed a weak to strong negative correlation: the larger the developer-written patch, the less likely automated repair is to produce a patch. The Mann-Whitney U test showed this relationship to be significant ($p \leq 0.05$) for all C techniques (see Fig. 8). For Java techniques, the results were insignificant ($p > 0.1$ or the 95% confidence interval spanned zero). Thus, we cannot conclude that the number of non-comment, non-blank lines in the developer-written patches is significantly associated with repairability for Java techniques.

These results indicate that C repair techniques are less likely to produce patches for defects that required developers to write more lines of code and edit more files to patch. This suggests that automated repair is more likely to patch easy defects than hard ones, reducing its utility. However, the correlation is not strong for all the techniques meaning that automated repair could still produce patches for some hard-to-repair-manually defects.



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	**−0.228**	**[−0.403, −0.054]**	**0.013**	**75**	**81**
TrpAutoRepair	**−0.182**	**[−0.359, −0.005]**	**0.049**	**84**	**72**
AE	**−0.221**	**[−0.396, −0.046]**	**0.016**	**75**	**81**
⋃C	**−0.249**	**[−0.426, −0.072]**	**0.008**	**91**	**65**
ManyBugs (105-defect subset)					
SPR	**−0.405**	**[−0.625, −0.185]**	**0.001**	**41**	**43**
KaliC	**−0.342**	**[−0.569, −0.115]**	**0.006**	**39**	**45**
Prophet	**−0.363**	**[−0.624, −0.102]**	**0.011**	**22**	**62**
⋃C	**−0.388**	**[−0.614, −0.161]**	**0.002**	**44**	**40**
Defects4J					
GenProgJ	−0.083	[−0.318, 0.152]	0.485	27	193
Nopol	0.209	[−0.018, 0.437]	0.050	34	186
KaliJ	−0.072	[−0.315, 0.170]	0.578	22	198
⋃Java	0.060	[−0.128, 0.247]	0.533	46	174

Fig. 8 The number of non-comment, non-blank lines of files in the developer-written patch data are available for all 185 ManyBugs and 220 Defects4J defects classified as bugs. This number is strongly correlated with automated repair techniques' ability to produce patches. This result is strongly statistically significant for C repair techniques and Nopol. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for those defects classified as bugs

4.3 Test Effectiveness

RQ3: Is an automated repair techniques ability to produce a patch for a defect correlated with the effectiveness of the test suite used to repair that defect?

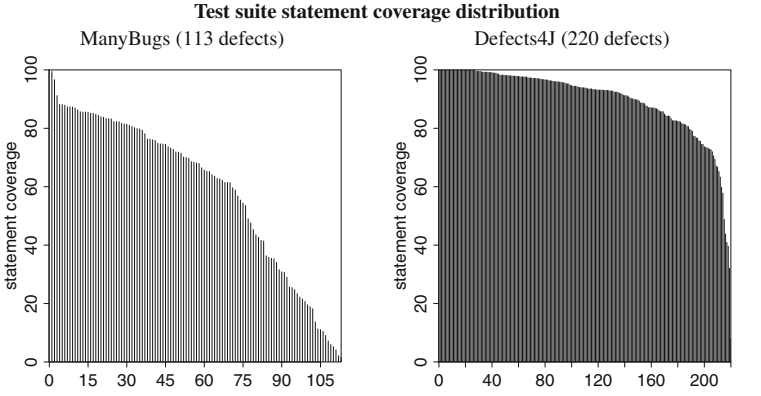
We measure a test suite's quality using three parameters, statement coverage, the number of defect-triggering test cases, and the number of relevant test cases (recall Section 3).

The Fraction of the Lines in the Files Edited by the Developer-Written Patches that are Executed by the Test Suite. For ManyBugs, we were able to compute test suite statement coverage for 113 out of 156 defects classified as bugs. This measure — the fraction of the lines in the files edited by the developer-written patches that are executed by the test suite — varied from 1.6% to 99.4% uniformly across the 113 defects. For Defects4J, we were able to compute test suite statement coverage for all 220 defects classified as bugs. The fraction varied from 7.9% to 100%; for 214 out of 220 defects, the fraction was above

50%, for 5 defects, the fraction was between 30% and 50%, and for 1 defect, the fraction was 7.9%. Top of Fig. 9 shows the distribution of test suite statement coverage values.

For C and Java techniques, the results were insignificant. Somers' d showed a very weak to weak correlation between the coverage of the test suite used to repair the defect and the automated repair's ability to produce a patch for that defect (Fig. 9); the 95% confidence interval for Somers' d consistently spanned zero and the Mann-Whitney U test showed that the differences between the distributions are not statistically significant ($p > 0.1$ for all techniques).

The Number of Defect-Triggering Test Cases For ManyBugs, all 156 defects classified as bugs had information on the number of test cases that trigger the defect. This number of tests varied from 1 to 52. Of these 156 defects, 111 had only a single triggering test case. For Defects4J, all 220 defects had this information, varying from 1 to 28. Of the 220 defects,



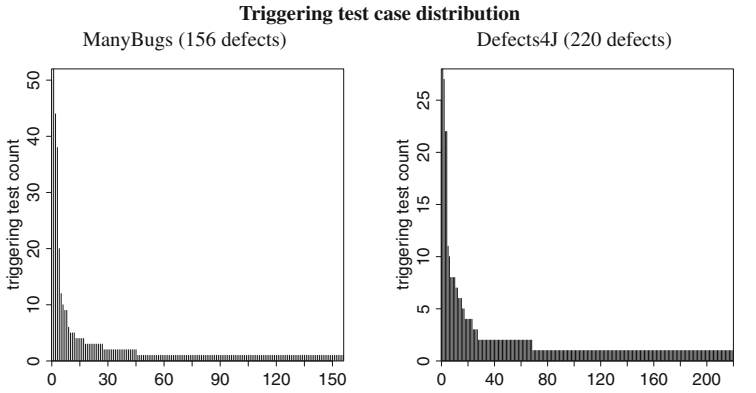
technique	Somers' d	95% CI		p	patched	unpatched
ManyBugs						
GenProgC	−0.094	[−0.312,	0.124]	0.390	55	58
TrpAutoRepair	−0.088	[−0.304,	0.127]	0.421	60	53
AE	−0.159	[−0.374,	0.057]	0.146	57	56
⋃C	−0.166	[−0.378,	0.046]	0.134	66	47
ManyBugs (105-defect subset)						
SPR	−0.073	[−0.365,	0.219]	0.640	34	25
KaliC	−0.090	[−0.383,	0.203]	0.558	32	27
Prophet	−0.109	[−0.484,	0.266]	0.538	15	44
⋃C	−0.094	[−0.387,	0.198]	0.550	36	23
Defects4J						
GenProgJ	−0.097	[−0.354,	0.160]	0.416	27	193
Nopol	−0.064	[−0.278,	0.149]	0.555	34	186
KaliJ	−0.158	[−0.468,	0.152]	0.226	22	198
⋃Java	−0.011	[−0.204,	0.181]	0.907	46	174

Fig. 9 The statement coverage of the test suite are available for 113 ManyBugs and 220 Defects4J defects classified as bugs. There is a weak, statistically insignificant, negative correlation for all techniques, between the coverage of the test suite used to repair the defect, and the technique's ability to produce a patch for the defect. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. The data shown are only for those defects classified as bugs for which we could compute coverage information

152 had only a single triggering test. Top of Fig. 10 shows the distribution of triggering test counts.

For C techniques, the results were insignificant ($p > 0.1$ or the 95% confidence interval spanned zero). For Java techniques, Somers' d showed a weak to moderate negative correlation between the number of triggering test cases and the ability to produce a patch (Fig. 10). The Mann-Whitney U test indicated this result to be statistically significant ($p \leq 0.05$) for all Java techniques except KaliJ, for which $p \leq 0.1$. We conclude that the number of triggering tests negatively affects a Java technique's ability to produce a patch.

The Number of Relevant Test Cases (Test Cases that Execute at Least one Line of the Developer-Written Patch) For ManyBugs, we annotated the same 156 defects with the total number of positive and negative test cases provided for each defect in ManyBugs benchmark. The number of relevant test cases varied from 3 to 7,951. For Defects4J, we

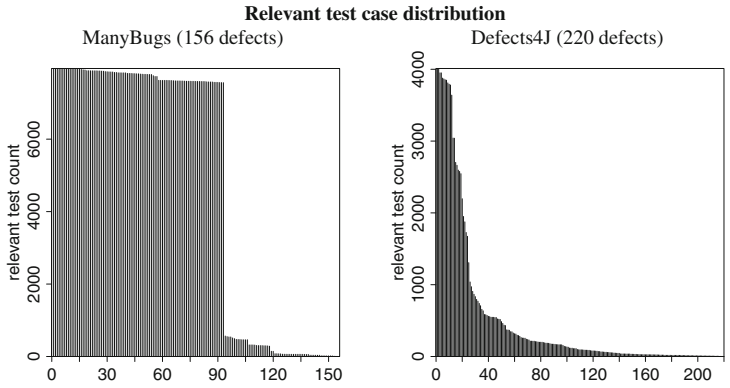


technique	Somers' d	95% CI		p	patched	unpatched
ManyBugs						
GenProgC	0.098	[−0.047,	0.243]	0.187	75	81
TrpAutoRepair	0.137	[−0.004,	0.278]	0.065	84	72
AE	0.067	[−0.079,	0.212]	0.370	75	81
⋃C	0.087	[−0.056,	0.229]	0.249	91	65
ManyBugs (105-defect subset)						
SPR	0.018	[−0.182,	0.217]	0.866	41	43
KaliC	−0.013	[−0.212,	0.186]	0.905	39	45
Prophet	−0.063	[−0.266,	0.140]	0.589	22	62
⋃C	−0.025	[−0.225,	0.175]	0.810	44	40
Defects4J						
GenProgJ	**−0.231**	**[−0.348,**	**−0.113]**	**0.017**	**27**	**193**
Nopol	**−0.221**	**[−0.337,**	**−0.104]**	**0.012**	**34**	**186**
KaliJ	−0.177	[−0.342,	−0.013]	0.095	22	198
⋃Java	**−0.252**	**[−0.348,**	**−0.156]**	**0.001**	**46**	**174**

Fig. 10 The number of triggering test cases is available for all 156 ManyBugs and 220 Defects4J defects classified as bugs. There is a negative correlation between a defect's number of triggering test cases and the ability to produce a patch, but this relationship is only statistically significant for the Java repair techniques. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bolf. The data shown are only for those defects classified as bugs

annotated the 220 defects with the number of relevant tests provided for each defect in Defects4J benchmark and the number of relevant test cases varied from 1 to 4011. Top of Fig. 11 shows the distribution of relevant test counts.

For C techniques, Somers' d showed a very weak positive correlation for TrpAutoRepair. The Mann-Whitney U test indicated this result to be statistically significant ($p \leq 0.05$). However, our confounding factor analysis (Section 4.8.3) found that this correlation was due to a correlation between the number of relevant test cases and the number of files edited by the developer-written patch. For all other C techniques, the results were insignificant ($p > 0.1$ or the 95% confidence interval spanned zero). For Java techniques, Somers' d showed a moderate to strong negative correlation between the number of relevant test cases and the ability to produce a patch for all techniques (Fig. 11). The correlation was statistically significant ($p \leq 0.05$) for Nopol and for $\bigcup$Java and weakly statistically significant ($p \leq 0.1$) for GenProgJ and KaliJ.



technique	Somers' d	95% CI		p	patched	unpatched
ManyBugs						
GenProgC	0.166	[−0.017,	0.349]	0.074	75	81
TrpAutoRepair	**0.188**	[**0.008**,	**0.368**]	**0.043**	**84**	**72**
AE	0.122	[−0.062,	0.307]	0.188	75	81
$\bigcup$C	0.134	[−0.048,	0.316]	0.154	91	65
ManyBugs (105-defect subset)						
SPR	0.087	[−0.162,	0.335]	0.497	41	43
KaliC	0.085	[−0.164,	0.334]	0.507	39	45
Prophet	−0.096	[−0.396,	0.204]	0.509	22	62
$\bigcup$C	0.050	[−0.199,	0.299]	0.697	44	40
Defects4J						
GenProgJ	−0.204	[−0.432,	0.024]	0.086	27	193
Nopol	**−0.317**	[**−0.549**,	**−0.084**]	**0.003**	**34**	**186**
KaliJ	−0.223	[−0.505,	0.059]	0.087	22	198
$\bigcup$Java	**−0.313**	[**−0.500**,	**−0.125**]	**0.001**	**46**	**174**

Fig. 11 The number of relevant test cases is available for all 156 ManyBugs and 220 Defects4J defects classified as bugs. For Java repair techniques, there is a weak to moderate significant negative correlation between a defect's number of relevant test cases and the ability to produce a patch. For C repair techniques, the correlation is weakly positive. These correlations are statistically significant for a subset of techniques. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for those defects classified as bugs

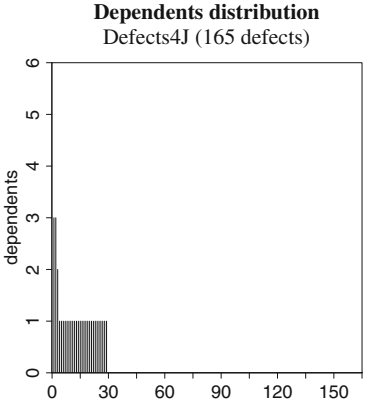
These results indicate that Java repair techniques are less likely to produce patches for defects with more triggering or more relevant tests. Test suite coverage does not significantly correlate with the ability to produce a patch. These findings are concerning, as they show it is harder to produce patches in situations that prior work has shown to lead to higher-quality patches (Smith et al. 2015).

4.4 Defect Independence

RQ4: Is an automated repair techniques ability to produce a patch for a defect correlated with that defects dependence on other defects?

Our dataset turned out to be insufficient to draw conclusions on a relationship between independence and repairability. For ManyBugs, 76 out of 156 defects classified as bugs had information on how many other defects they depended on, but none of them depended on other defects. For Defects4J, 165 out of 220 defects classified as bugs had this information. Of these, 136 did not depend on other defects, 26 depended on a one other defect, 1 on two other defects, and 2 on three other defects.

For C techniques, the lack of variability in the benchmark defects with respect to defect independence prevented us from drawing any conclusions. For Java techniques, the results were insignificant ($p > 0.1$ or the 95% confidence interval spanned zero), as Fig. 12 shows. This suggests that the number of other defects a defect depends on does not affect repairability of Java repair techniques.



	Defects4J					
technique	Somers' d	95% CI		p	patched	unpatched
GenProgJ	−0.075	[−0.233,	0.083]	0.494	18	147
Nopol	0.058	[−0.114,	0.230]	0.501	27	138
KaliJ	−0.039	[−0.230,	0.153]	0.767	14	151
⋃Java	0.032	[−0.117,	0.181]	0.725	35	130

Fig. 12 For Java repair techniques, there is a weak statistically insignificant correlation between a defect's dependence on other defects and the ability to produce a patch. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. The data shown are only for those defects classified as bugs

While we have developed a methodology that can be applied to other defect benchmarks, ManyBugs did not contain enough variability in defect independence to identify a relationship between independence and repairability. For Defects4J, an insignificant correlation is observed for all the techniques.

4.5 Developer-Written Patch Characteristics

RQ5: What characteristics of the developer-written patch for a defect are significantly associated with an automated repair techniques ability to produce a patch for that defect?

Investigating which characteristics of the developer-written patches are significantly associated with defect repairability allows us to reason about automated repair's ability to fix defects in terms of what the developers did. This may, in turn, lead to actionable advice about which kinds of defects the developer(s) should fix manually, and which can be trusted to automated repair. Of course, to use this information, the developer(s) must have a sense of the characteristics of the patch before it is written, which may sometimes be possible. However, the main goal of studying this research question is to help guide future research into automated program repair techniques by identifying the characteristics of the defects, in terms of the patches that repair them, that existing techniques struggle to produce patches for. Research into future repair tools may, for example, target modifying or inserting loops, just as, for example, Nopol targets conditional statements.

Figure 13 shows the distributions of the nine patch characteristics for the two benchmarks, and the results of a logistic regression using these characteristics. For each repair technique, Fig. 13 shows which patch characteristics are significantly associated with repairability and how much variance in repairability is explained by all defect characteristics.

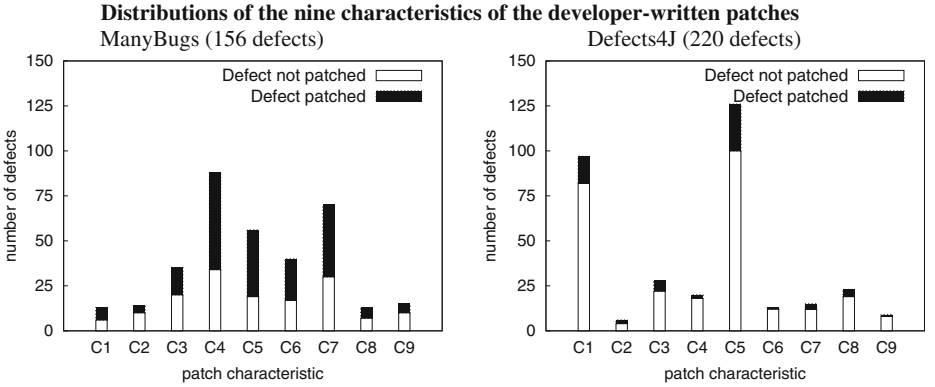
The data suggest that some characteristics of developer-written patches are significantly associated with repairability for C repair techniques, but not for Java repair techniques. In particular, for C repair techniques, changing a data structure or type, a function argument, or a conditional, or adding a new variable, an if statement, or a new function are significantly associated with repairability, whereas changing a method signature, or adding a function call, or a loop is not.

These results suggest that defects that required developers to insert a loop or a new function call, or change a method signature are challenging for automated repair techniques to patch. More patch characteristics are significantly associated with repairability for C repair techniques than for Java repair techniques.

4.6 Patch Quality

RQ6: What defect characteristics are significantly associated with an automated repair techniques ability to produce a high-quality patch for that defect?

Recent work has begun evaluating the quality of patches produced by automated repair (Brun et al. 2013; Smith et al. 2015; Qi et al. 2015; Martinez et al. 2017; Durieux et al. 2015; Long and Rinard 2015, 2016b; Pei et al. 2014). Until now, our analysis remained quality agnostic, focusing on whether techniques can produce patches, as opposed to whether



technique	model quality p	model quality R^2	patch characteristic characteristic #	patch characteristic p
GenProgC	**0.001**	**0.129**	**C3**	**0.036**
			C4	0.088
			C5	0.066
			C7	0.055
			C9	0.051
TrpAutoRepair	**0.005**	**0.109**	C3	0.058
			C6	**0.046**
			C7	**0.040**
			C9	**0.049**
AE	**0.042**	**0.081**	C2	0.057
			C7	0.068
SPR	**0.001**	**0.184**	C1	0.059
			C3	**0.014**
Prophet	**0.004**	**0.169**	**C3**	**0.003**
KaliC	0.213	0.100	C3	0.088
UC	**0.000**	**0.162**	**C3**	**0.001**
			C7	0.075
			C9	**0.030**
GenProgJ	0.555	0.047	C1	0.057
			C5	**0.029**
Nopol	0.572	0.040	none	
KaliJ	0.543	0.055	**C1**	**0.031**
UJava	0.395	0.041	**C1**	**0.039**

Fig. 13 The distribution of the nine patch characteristics for the developer-written patches in the ManyBugs and Defects4J benchmarks. A logistic regression reports that characteristics C1, C3, C5, C6, C7 and C9 — changing a data structure or type, a function argument, or a conditional, or adding a new variable, an if statement or a new function — are significantly associated with repairability. Data for which the p value is below 0.05 are bold

techniques can produce high-quality patches. Quality and applicability are orthogonal aspects of program repair: one can work on improving the quality of the produced patches, the applicability of the repair techniques to a wider range of defects, or both. However, it is important to also study how the two interact. At the present time, the quality of the patches produced by most techniques is fairly low. According to a manual analysis, on the 105-defect subset of ManyBugs, GenProgC could only produce 2 correct patches, TrpAutoRepair 3 correct patches, and AE 2 correct patches (Qi et al. 2015). On an 84-defect subset

of Defects4J, GenProgJ could only produce 5 correct patches, Nopol 5 correct patches, and KaliJ 1 correct patch (Martinez et al. 2017). The number of correct patches is too small for us to make statistically significant claims for these techniques. Inspired by the findings of the low quality of repairs, SPR and Prophet were designed to specifically improve repair quality. SPR produces 13 and Prophet 15 correct patches on the 105-defect subset of ManyBugs (Long and Rinard 2016b). Other techniques that claim to produce high-quality patches, e.g., SearchRepair (Ke et al. 2015), fail to scale to the size and complexity of real-world defects we consider. We use the SPR and Prophet data to begin studying the defect characteristics' effect on the ability to produce high-quality repairs. There are eight abstract parameters that have sufficient data to perform such analysis. Figure 14 shows the Somers' d and Mann-whitney U test results testing for an association between each of these eight abstract parameters and the ability to produce high-quality patches. Only the abstract parameters related to defect complexity and test suite effectiveness exhibit statistically significant associations.

Defect Complexity For Prophet, the number of non-comment, non-blank lines in the developer-written patch correlated negatively with the ability to produce a correct patch. This negative correlation was stronger compared to the negative correlation with the ability to produce a patch at all ($d = -0.564$ for correct patches, vs. $d = -0.342$ for all patches). For SPR, the correlations with the ability to produce a correct patch and a patch all were the same ($d = -0.405$). The Mann-Whitney U test confirmed this distribution difference to be statistically significant ($p \leq 0.05$) for both the techniques. However, for SPR, the 95% confidence interval for Somers' d spans zero for producing *correct* patches.

For Prophet, there was a weakly significant ($p \leq 0.1$) negative correlation between the number of files edited by the developer-written patch and the ability to produce a correct patch. For SPR, the correlation was insignificant.

Test Suite Effectiveness SPR showed a weakly significant ($p < 0.1$) positive correlation for producing correct patches when using higher-coverage test suites ($d = 0.312$ for correct patches, vs. $d = -0.073$ for all patches). This is consistent with prior results showing that higher-coverage test suites lead to higher-quality patches (Smith et al. 2015). The result for Prophet was not statistically significant ($p > 0.1$). Also, correlations with the number of triggering tests and relevant tests were either the same for the correct patches as all patches, or not statistically significant.

Developer-Written Patch Characteristics A logistic regression using the nine patch characteristics showed that characteristics C1 and C3 associated with SPR's ability to produce patches, and characteristic C3 associated with Prophet's ability to produce patches (recall Fig. 13). A logistic regression of *correct* patches generated by SPR and Prophet identified the same characteristics associating with producing correct patches, and also identified C7 (patch adds an if statement) as statistically significantly associating with producing correct patches (SPR $p = 0.044$, Prophet $p = 0.086$). Both SPR and Prophet target defects that can be repaired by inserting or modifying conditional statements, explaining the observation that adding if statements associates with producing correct patches.

None of the statistical tests revealed statistically significant results for the correct patches for the defect importance and defect independence characteristics.

Only two of the considered repair techniques, Prophet and SPR, produce a sufficient number of high-quality patches to evaluate. These techniques were less likely to produce patches for more complex defects, and they were even less likely to produce correct patches.

4.7 Feature Synthesis

We wanted to conduct the same statistical tests to measure correlation between defect characteristics with the ability to synthesize features using those defects in our benchmarks that are features, not bugs. Unfortunately, too few of the defects were features: Features make up 29 of the 185 defects in ManyBugs (21 of the 105-defect subset), and only 4 of the 224 in Defects4J. GenProgJ and KaliJ synthesize none of the features and Nopol synthesizes 1. Meanwhile GenProgC synthesized 11, TrpAutoRepair synthesized 12 and AE synthesized 10 out of 29 features from 185-subset of ManyBugs and SPR synthesized 5, Prophet synthesized 4 and KaliC synthesized 5 out of 21 features from 105-subset of ManyBugs. These sample sizes are too small and none of our experimentsv revealed statistically significant results.

abstract parameter	technique	ManyBugs (105-defect subset) Somers' *d*	95% CI	*p*	patched	unpatched
line count	**SPR (*produces patch*)**	**−0.405**	**[−0.655, −0.155]**	**0.001**	**41**	**43**
	SPR (*correct patch*)	−0.405	[−0.826, 0.016]	0.023	12	72
	Prophet (*produces patch*)	**−0.342**	**[−0.594, −0.090]**	**0.006**	**39**	**45**
	Prophet (*correct patch*)	**−0.564**	**[−0.963, −0.165]**	**0.001**	**14**	**70**
file count	**SPR (*produces patch*)**	**−0.256**	**[−0.402, −0.111]**	**0.002**	**41**	**43**
	SPR (*correct patch*)	−0.120	[−0.264, 0.024]	0.328	12	72
	Prophet (*produces patch*)	**−0.191**	**[−0.339, −0.042]**	**0.028**	**39**	**45**
	Prophet (*correct patch*)	−0.214	[−0.284, −0.144]	0.093	14	70
statement coverage	SPR (*produces patch*)	−0.073	[−0.366, 0.220]	0.640	34	25
	SPR (*correct patch*)	0.312	[−0.071, 0.695]	0.099	12	47
	Prophet (*produces patch*)	−0.090	[−0.385, 0.204]	0.558	32	27
	Prophet (*correct patch*)	0.284	[−0.101, 0.668]	0.135	12	47
triggering test count	SPR (*produces patch*)	0.018	[−0.182, 0.217]	0.866	41	43
	SPR (*correct patch*)	0.109	[−0.208, 0.426]	0.470	12	72
	Prophet (*produces patch*)	−0.013	[−0.212, 0.186]	0.905	39	45
	Prophet (*correct patch*)	0.118	[−0.179, 0.416]	0.390	14	70
relevant test count	SPR (*produces patch*)	0.087	[−0.159, 0.332]	0.497	41	43
	SPR (*correct patch*)	0.319	[−0.100, 0.739]	0.078	12	72
	Prophet (*produces patch*)	0.085	[−0.162, 0.332]	0.507	39	45
	Prophet (*correct patch*)	0.272	[−0.093, 0.638]	0.110	14	70
priority	SPR (*produces patch*)	−0.111	[−0.510, 0.288]	0.753	9	5
	SPR (*correct patch*)	−0.154	[−0.366, 0.058]	1.000	1	13
	Prophet (*produces patch*)	−0.111	[−0.510, 0.288]	0.753	9	5
	Prophet (*correct patch*)	*NA*	*NA*	*NA*	0	14
versions	SPR (*produces patch*)	−0.120	[−0.358, 0.118]	0.338	25	17
	SPR (*correct patch*)	−0.206	[−0.307, −0.104]	0.312	8	34
	Prophet (*produces patch*)	0.021	[−0.205, 0.246]	0.882	23	19
	Prophet (*correct patch*)	−0.206	[−0.307, −0.104]	0.312	8	34
time to fix	SPR (*produces patch*)	−0.175	[−0.555, 0.205]	0.382	20	16
	SPR (*correct patch*)	−0.172	[−0.681, 0.337]	0.501	7	29
	Prophet (*produces patch*)	−0.152	[−0.537, 0.233]	0.447	19	17
	Prophet (*correct patch*)	−0.192	[−0.710, 0.326]	0.452	7	29

Fig. 14 Defect complexity and test suite effectiveness exhibit statistically significant associations with the techniques' ability to produce high-quality patches. SPR and Prophet are the only two techniques that produce a sufficient number of high-quality patches for this analysis. The 95% CI (confidence interval) column shows the range in which Somers' *d* lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a *p* value below 0.05 and the 95% CI does not span zero are bold. The data shown are only for those defects classified as bugs and with known respective parameter values

4.8 Discussion

This section discusses the implications of our findings (Section 4.8.1), makes observations about our dataset and the use of the methodology that produced it (Section 4.8.2), and analyzes potential confounding factors within our evaluation (Section 4.8.3).

4.8.1 Implications

Our data suggest several encouraging conclusions. First, automated repair techniques are slightly more likely to produce patches for defects of a higher priority, and are equally likely to produce patches for defects regardless of how long developer(s) took to fix them. The former finding may suggest differences between low- and high-priority defects, from the point of view of automated program repair. Second, while overall, automated repair techniques were more likely to produce patches for defects that required fewer edits by the developers to fix, the correlations were not strong for all techniques, and the techniques were able to produce patches for some hard-to-repair-manually defects. Producing larger patches requires search-based automated repair techniques to explore more of the search space, which requires longer execution time. As repair techniques typically operate with a time limit, finding such patches may be more difficult than smaller ones. The fact that techniques were able to find patches for some defects that were hard to repair manually suggests that either the techniques are able to sometimes successfully traverse the large search space, or that smaller patches exist than the manually written ones.

At the same time, our data suggest that Java repair techniques had a harder time producing patches for defects with more triggering or more relevant tests. This finding is intuitive because each test executing code related to the defect represents constraints on the patch. To produce a patch, the automated repair techniques have to modify the code to satisfy all the constraints. The more constraints there are, the harder it is to find a satisfying patch. Prior studies have found that higher-coverage test suites can lead to higher-quality patches (Smith et al. 2015) and that larger search spaces lead to a higher fraction of incorrect patches (Long and Rinard 2016a). As a result, we find that test suites that make it easier to produce a patch reduce, in expectation, the quality of the produced patch. This identifies a research challenge of creating techniques that are capable of either finding patches effectively even when constrained by high-quality test suites, or discriminating between low-quality and high-quality patches despite using test suites that provide few guiding constraints.

We identified some evidence that targeting repair techniques to specific defects is worthwhile, as defects that required a developer to write new if statements were more likely to be correctly repaired by SPR and Prophet, two techniques designed to insert or modify conditional statements. This provides preliminary evidence that perhaps when automated repair techniques are applied to defects that developers patched using the kinds of changes the techniques are designed to make, the techniques are capable of making higher-quality changes.

4.8.2 Dataset Observations

The ManyBugs and Defects4J datasets lack certain kinds of data diversity to answer some of our proposed research questions. For example, every defect in ManyBugs that included

ManyBugs								
	file count	line count	relevant test count	triggering test count	statement coverage	time to fix	versions	priority
line count	**0.46**							
relevant test count	**−0.22**	−0.13						
triggering test count	**0.05**	**0.14**	−0.04					
statement coverage	−0.01	−0.14	**0.30**	0.03				
time to fix	0.07	0.10	−0.19	0.11	0.02			
versions	0.11	0.10	**−0.30**	−0.16	−0.12	0.18		
priority	−0.23	−0.50	0.12	−0.14	0.26	0.12	0.06	
dependents	—	—	—	—	—	—	—	—

Defects4J								
	file count	line count	relevant test count	triggering test count	statement coverage	time to fix	versions	priority
line count	0.17							
relevant test count	0.17	0.08						
triggering test count	0.11	0.04	0.22					
statement coverage	0.00	−0.02	−0.01	−0.03				
time to fix	0.05	0.13	**0.22**	0.11	−0.02			
versions	−0.10	0.02	−0.02	−0.02	0.00	−0.15		
priority	0.03	0.11	**−0.13**	−0.05	0.04	−0.19	**0.10**	
dependents	0.04	0.09	−0.05	−0.10	0.05	0.17	−0.11	0.09

Fig. 15 Pairwise Spearman correlation coefficients for the abstract parameters for the ManyBugs and Defects4J defects. The bold coefficients are statistically significant ($p \leq 0.05$) and underlined coefficients are weakly statistically significant ($p \leq 0.1$)

dependence information did not depend on other defects. Similarly, only two of the evaluated repair techniques produced sufficiently many high-quality patches for our analysis to make statistically significant findings about patch quality. Nevertheless, this paper presents a methodology that can be applied to other datasets to derive more data to answer these questions, particularly as the body of defects on which automated program repair techniques are evaluated grows.

One of the goals of our study has been to create a methodology for evaluating the applicability of automated program repair techniques that can be applied to new techniques and help drive research toward improving such applicability. As such, none of the defect characteristics we consider are specific to a repair technique. For example, we define defect complexity in terms of the number of lines and number of files edited by the minimized developer-written patch, and how easy it is to reproduce the defect. We do not take into account that some repair techniques may, for example, find defects that involve control flow more complex than ones that do not. Our study of RQ5 empirically identifies several characteristics of the defects' developer-written patches (such as if the patch changes a conditional or adds a function argument) that associate with the techniques' ability to produce patches for those defects. Studying technique-specific complexity of defects is also a worthwhile effort, but it is beyond the scope of our work on creating a technique-agnostic applicability evaluation methodology.

4.8.3 Confounding Factor Analysis

To consider potential confounding factors in our analyzes, we computed the Spearman correlation coefficients between all pairs of abstract parameters. Figure 15 shows these

coefficients for ManyBugs and Defects4J. The **bold** coefficients are statistically significant ($p \leq 0.05$) and underlined coefficients are weakly statistically significant ($p \leq 0.1$). To be conservative in our analysis, we consider all pairs that correlate at least weakly significantly ($p < 0.1$) to pose potential confounding factors. We found that for ManyBugs, the following pairs of cross-defect-characteristic parameters correlated at least weakly significantly: file count correlates with relevant test count and triggering test count, line count correlates with triggering test count and statement coverage, and versions correlates with relevant test count. For Defects4J, relevant test count correlates with time to fix and priority. (All other correlations of at least weak statistical significance were within defect characteristics, e.g., file count correlated with line count.)

For each correlating cross-characteristic parameter pair $\langle p_1, p_2 \rangle$, we created four logistic regression models for repairability:
Model_1: a model using only p_1,
Model_2: a model using only p_2,
Model_{1+2}: a model using a linear combination of p_1 and p_2 ($p_1 + p_2$), and
Model_{1*2}: a model using all possible interactions between p_1 and p_2 ($p_1 * p_2$).
We then pairwise compare the models' goodness of fit using the area under the curve and determine the statistical significance in the models' quality improvement. We consider improvements that are at least weakly statistically significant to demonstrate confounding factors. If Model_{1+2} shows a significant improvement over model Model_1, we determine that parameter p_2's contribution to the model is significant. Similarly, if Model_{1+2} improves over Model_2 then p_1 contributes. Finally, if Model_{1*2} improves significantly on Model_{1+2}, then there exists an interaction between p_1 and p_2 whose contribution is significant.

Analyzing the correlated pairs of parameters, we find that:

- For all C techniques except TrpAutoRepair, relevant test count does not contribute significantly to model quality beyond file count's contribution. We conclude, for our C analysis, that the observed correlation (weak, significant for TrpAutoRepair) between relevant test count and repairability (Section 4.3) is not due to the confounding factor of relevant test count correlating with file count. The interactions between relevant test count and file count do not significantly contribute to the quality of model.
- For all C techniques, triggering test count does not contribute significantly to model quality beyond file count's contribution; however, for GenProgC, interactions of file count and triggering test count do offer a significant contribution. We conclude, for our C analysis, that the observed correlation (moderate, weakly significant for TrpAutoRepair) between triggering test count and repairability (Section 4.3) is likely largely due to the confounding factor of triggering test count correlating with file count, and file count correlating with repairability, although the combination of the two parameters does add some useful information.
- For GenProgC, TrpAutoRepair, SPR, Prophet, and $\bigcup$C on the full 185-defect ManyBugs, both line count and triggering test count contribute significantly to model quality. For TrpAutoRepair, Prophet, and $\bigcup$C, ManyBugs, interactions between the two parameters significantly contribute more than the two parameters on their own. We conclude, for our C analysis, that the correlation between line count and triggering test count is not a confounding factor.
- For each C technique except AE and $\bigcup$C on 185-defect ManyBugs, both line count and statement coverage contribute significantly to model quality. The interactions between

the two offer no significant contribution. We conclude, for our C analysis, that the correlation between line count and statement coverage is not a confounding factor.
- For each C technique except GenProgC, both relevant test count and versions contribute significantly to model quality. The interactions between the two parameters do not contribute significantly more than the two parameters on their own. We conclude, for our C analysis, that the correlation between relevant test count and versions is not a confounding factor.
- For each Java technique except Nopol, both relevant test count and time to fix contribute significantly to model quality. For Nopol, time to fix does not contribute significantly to model quality beyond relevant test count's contribution. The interactions between the two parameters do not contribute additional information. We conclude, for our Java analysis, that the correlation between relevant test count and time to fix is not a confounding factor.
- For each Java technique except GenProgJ and KaliJ, priority and relevant test count contribute significantly to the model quality. For GenProgJ and KaliJ, priority does not contribute significantly beyond relevant test count's contribution. The interactions between the two parameters do not contribute significantly more than the two parameters on their own. We conclude, for our Java analysis, that the observed correlations (moderate, weakly significant for GenProgJ and KaliJ) between priority and repairability (Section 4.1) is likely largely due to the confounding factor of priority correlating with relevant test count, and relevant test count correlating with repairability.

We conclude that the number of files edited by the developer-written patch is a confounding factor to relevant and triggering test count in the ManyBugs dataset, and that relevant test count is a confounding factor to priority in the Defects4J dataset. All other observed correlations between the parameters do not indicate confounding factors. Our earlier conclusions are not affected by this analysis as only weak or weakly significant observed correlations in only a few cases are affected by the confounding factors, and those observed correlations did not lead to conclusions.

5 Threats to Validity

This paper investigates the relationship between automated repair techniques' ability to produce patches and characteristics of the defects, test suites, and developer-written patches for the defects. However, this paper only begins to explore the relationship between these characteristics and the *quality* of the patches (recall RQ6). Future work needs to address this concern, as today, techniques repair very few of the studied defects correctly, reducing the power of our analysis. The methodology presented in this paper can be applied to other defect benchmarks as they become available, and to other repair techniques that focus on repair quality and applicability.

The goal of our study is to characterize the kinds of defects for which automated program repair is capable of producing patches and high-quality patches. The study is observational. Of course, given a defect, changing its metadata, such as its priority, will not affect the techniques' ability to produce patches, and our findings should be viewed as directing research into improving or creating new automated program repair techniques, not as methods for

making existing repair techniques apply to specific defects. However, some of our findings suggest how altering inputs to automated program repair techniques may affect patch production, such as that increasing the number of tests may make it more difficult to produce a patch.

We took steps to ensure that our study is objective and reproducible. All characteristics derived from source-code repositories and developer-written patches are computed using deterministic scripts, available at https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/. However, some of the judgments with respect to which parameters are relevant to this study are subjective. We addressed this threat by having two authors independently collect parameters in eleven issue tracking systems, and independently measure all subjective parameters for the defects we considered. The authors then merged their findings, paying special attention to any disagreements in initial judgments.

Our study relies on repairability and quality results presented by prior work (Martinez et al. 2017; Le Goues et al. 2012b, 2015; Long and Rinard 2015, 2016b; Qi et al. 2015) and errors or subjective judgment quality in that work affects our findings. As an example, GenProgC makes an assumption that the source code files that must be edited to repair a defect are known (Le Goues et al. 2012b, 2015); this assumption may not hold in practice, and that would threaten the generalizability of those repairability results, and, in turn, our findings. Using results from multiple studies partially mitigates this threat, although using an objective measure of quality, such as number of independent tests a patch passes (Smith et al. 2015), would go farther. Unfortunately, such an evaluation requires multiple, independent, high-quality test suites for real-world defects, and such test suites do not exist for the ManyBugs benchmark, but do for Defects4J.

While GenProgC was designed prior to ManyBugs, the other techniques have been developed in part to compete with GenProgC on the then known (at least partially) ManyBugs benchmark. This may affect the generalizability of the techniques to other defects, which, in turn affects the generalizability of our results. We mitigate this threat by using two defect benchmarks and multiple repair techniques.

Our study treats all parameters related to a characteristic as equally important. This is likely an oversimplification of the real world. For example, defect priority is likely a better indicator of a defect's importance than the number of project versions the defect affects. To mitigate this threat, we perform independent analyzes with each parameter.

Our study treats all defects related to the same issue in an issue tracking system equally. This may contribute to noisy data. For example, a single issue, and commit, may resolve two defects. One of these defects may be critical, while the other is not. Our analysis, due to lack of finer granularity in the source code repository and issue tracking system, considers both defects as critical, potentially affecting our findings. The relatively low number of such defects mitigates this threat, although another study could remove such defects altogether.

We consider a developer-written patch to be a proxy of the complexity of the defect it patches. In theory, there may be many other patches for the same defect, some smaller and simpler and others larger and more complex. We mitigate this threat by considering a large number of defects and using benchmarks of mature software projects that are less likely to accept poorly written patches.

6 Related Work

Prior work has argued the importance of evaluating the types of defects automated repair techniques can repair, and evaluating the produced patches for understandability,

correctness, and completeness (Monperrus 2014). Our work addresses the concern of evaluating how defect characteristics affect automated repair success. Most initial technique presentations evaluate what fraction of a set of defects the technique can produces patches for, e.g., (Le Goues et al. 2012b; Jin et al. 2011; Carzaniga et al. 2013; Weimer et al. 2009; Weimer et al. 2013; Dallmeier et al. 2009). Some research has evaluated techniques for how quickly they produce patches (e.g., (Le Goues et al. 2012a; Weimer et al. 2013)), how maintainable the patches are (Fry et al. 2012), and how likely developers are to accept them (Kim et al. 2013). These evaluations neither considered the quality of the repair, nor the characteristics of the defects that affect the techniques' success. More recent work has focused on evaluating the quality of repair. For example, on 204 Eiffel defects, manual patch inspection showed that AutoFix produced high-quality patches for 51 (25%) of the defects, which corresponded to 59% of the patches it produced (Pei et al. 2014). While AutoFix uses contracts to specify desired behavior, by contrast, the patch quality produced by techniques that use tests has been found to be much lower. Manual inspection of the patches produced by GenProgC, TrpAutoRepair (referred to as RSRepair in that paper), and AE on a 105-defect subset of ManyBugs (Qi et al. 2015), and by GenProgJ, Nopol, and KaliJ on a 224-defect subset of Defects4J (Martinez et al. 2017) showed that patch quality is often lacking in automatically produced patches. An automated evaluation approach that uses a second, independent test suite not used to produce the patch to evaluate the quality of the patch similarly showed that GenProgC, TrpAutoRepair, and AE all produce patches that overfit to the supplied specification and fail to generalize to the intended specification (Smith et al. 2015). This work has led to more research, and new techniques that improve the quality of the patches (Long and Rinard 2015, 2016b; Ke et al. 2015); however, the questions of which defect characteristics affect repair success, and what kinds of defects can be repaired remain unanswered. It is the goal of this paper to begin answering these questions, and to motivate research that improves the applicability of automated repair to important and difficult-to-repair defects.

Several benchmarks of defects have evolved specifically for evaluating automated repair. The ManyBugs benchmark (Le Goues et al. 2015) consists of 185 C defects in real-world software. The IntroClass benchmark (Le Goues et al. 2015) consists of 998 C defects in very small, student-written programs (although not all 998 are unique). Additionally, Defects4J (Just et al. 2014), a benchmark of 357 Java defects in real-world software, while not explicitly designed for automated repair, has been used for this purpose (Martinez et al. 2017). A few other benchmarks have been used, e.g., to evaluate AFix (Jin et al. 2011), but these benchmarks have either not been made publicly available or have not been used as widely for automated repair evaluation. Automated repair evaluations using small-scale benchmarks with artificial defects (Kong et al. 2015), e.g., the Siemens benchmark (Hutchins et al. 1994), are unlikely to generalize to real-world defects. This paper chose to focus on ManyBugs and Defects4J because they are publicly available and well documented, enabling us to collect the necessary data about defect characteristics described in Section 3, as well as the fact that these benchmarks have been used widely for evaluation of automated repair techniques (Le Goues et al. 2012b, 2015; Weimer et al. 2013; Qi et al. 2015; Long and Rinard 2015, 2016b; Martinez et al. 2017; Durieux et al. 2015; DeMarco et al. 2014). The IntroClass defects are also well documented, publicly available, and used in several automated technique evaluations, but the relatively small size of the defects and the projects within which they are contained make them a poor dataset for our purposes. Our

work evaluates techniques that work on C and Java defects, but does not explicitly compare the languages to each other.

Automated repair approaches can be classified broadly into two classes: (1) *Generate-and-validate* approaches create candidate patches (often via search-based software engineering (Harman 2007)) and then validate them, typically through testing (e.g., (Perkins et al. 2009; Weimer et al. 2009; Alkhalaf et al. 2014; Carzaniga et al. 2013; Carzaniga et al. 2010; Coker and Hafiz 2013; Ke et al. 2015; Kim et al. 2013; Weimer et al. 2013; Liu et al. 2014; Sidiroglou-Douskos et al. 2015; Tan and Roychoudhury 2015; Debroy and Wong 2010; Qi et al. 2015; Long and Rinard 2015, 2016b). (2) *Synthesis-based* approaches use constraints to build correct-by-construction patches via formal verification or inferred or programmer-provided contracts or specifications (e.g., Wei et al. (2010), Pei et al. (2014), Mechtaev et al. (2015), Mechtaev et al. (2016), and Jin et al. (2011)).

Generate-and-validate repair works by generating multiple candidate patches that might address a particular bug and then validating the candidates to determine if they constitute a repair. In practice, the most common form of validation is testing. These approaches differ in how they choose which locations to modify, which modifications are permitted, and how the candidates are evaluated. Some approaches use heuristic search over the search space of patches by applying mutation and crossover selection; e.g., GenProg uses a genetic algorithm (Weimer et al. 2009; Le Goues et al. 2012a, b), and TrpAutoRepair (Qi et al. 2013) uses random search. Kali (Qi et al. 2015; Martinez et al. 2017) and Debroy and Wong (Debroy and Wong 2010) use exhaustive search over modification operators, such as statement removal and conditional operations. SPR (Long and Rinard 2015) synthesizes candidate conditionals that can be inserted into programs. MT-APR (Jiang et al. 2016) uses metamorphic testing, eliminating the need for test oracles by instead of checking the correctness of individual test outputs, check testing results through verification of relations among multiple test cases and their outputs. SearchRepair (Ke et al. 2015) uses candidate code selected from other source code based on automatically-generated, desired patch input-output relationship profiles to generate patches. Par (Kim et al. 2013) uses templates from historical developer-written patches as its set of allowed code modifications, although these templates apply to only 15% of manually-written defect patches (Soto et al. 2016). Prophet (Long and Rinard 2016b) similarly learns templates from prior developer-written patches, but in an automated manner. ClearView (Perkins et al. 2009) constructs run-time patches using Daikon-mined data-value pre- and post-conditions (Ernst et al. 2001).

Some testing is concerned with non-functional properties (Ammann and Offutt 2008; Galhotra et al. 2017) and errors outside of the codebase (Muşlu et al. 2013, 2015; Wang et al. 2015). Repair-like approaches can be applied in this context, e.g., (Langdon et al. 2016; Petke et al. 2017; Schulte et al. 2014) for program improvement. Our study did not consider repair of such non-functional properties, but future work could apply a similar methodology in that context.

By contrast to test-based generate-and-validate repair mechanism, synthesis-based approaches generate correct-by-construction patches, still, of course, limited by the quality of the specification. Nopol (DeMarco et al. 2014), SemFix (Nguyen et al. 2013), DirectFix (Mechtaev et al. 2015), and Angelfix (Mechtaev et al. 2016) use SMT or SAT constraints to encode test-based specifications. (SearchRepair (Ke et al. 2015) mentioned earlier also uses SMT-constraints to encode desired input-output relationships.) AutoFix (Wei et al. 2010; Pei et al. 2014) and AFix (Jin et al. 2011) deterministically generate patches using

manually-specified pre- and post-condition contracts, with AFix targeting atomicity violations in concurrent programs. Synthesis-based approaches can repair not only software but also software tests, e.g., SPECTR (Yang et al. 2012). Real-world specifications are often partial, but higher quality specifications, e.g., contracts (Wei et al. 2010; Pei et al. 2014), do tend to produce better quality patches (Smith et al. 2015). This observation has implications for systems that make runtime decisions, such as self-adaptive systems, e.g., Brun and Medvidovic (2007a, 2007b); Brun et al. (2015), suggesting that perhaps high-quality specifications are necessary in that context to improve adaptation quality.

Many of these automated repair techniques evaluate using real-world defects obtained through ad-hoc case studies, manual search through bug databases, industrial partnerships, and informal communication with developers. Our study covers many of these techniques, particularly ones evaluated on publicly available benchmarks, and can be applied to other techniques as well. While some of these approaches are designed to tackle specific types of defects (e.g., AFix), the techniques' evaluations do not focus on which types of defects, and which defect characteristics positively, and negatively, affect repairability.

7 Contributions

Automated program repair has recently become a popular area of research, but most evaluations of repair techniques focus on how many defects a technique can produce a patch for. This paper, for the first time, analyzes how characteristics of the defects, the test suites, and the developer-written patches correlate with the repair techniques' ability to produce patches, and to a smaller degree, produce high-quality patches. The paper studies seven popular repair techniques applied to two large defect benchmarks of real-world defects.

We find that automated repair techniques are less likely to produce patches for defects that required developers to write a lot of code or edit many files, or that have many tests relevant to the defect, and that Java techniques are moderately more likely to produce patches for high-priority defects. The time it took developers to fix a defect does not correlate with automated repair techniques' ability to produce patches. A test suite's coverage also does not correlate with the ability to produce patches, but higher coverage correlated with higher-quality patches. Finally, automated repair techniques had a harder time fixing defects that required developers to add loops or new function calls, or change method signatures.

We produce a methodology and data that extend the ManyBugs and Defects4J benchmarks to enable evaluating new automated repair techniques, answering questions such as "can automated repair techniques repair defects that are hard for developers to repair, or defects that developers consider important?"

Our findings both raise concerns about automated repair's applicability in practice, and also provide promise that, in some situations, automated repair can properly patch important and hard defects. Recent work on evaluating repair quality (Brun et al. 2013; Smith et al. 2015; Qi et al. 2015; Martinez et al. 2017; Durieux et al. 2015) has led to work to improve the quality of patches produced by automated repair (Long and Rinard 2015, 2016b; Ke et al. 2015). Our position is that our work will similarly inspire new research into improving the applicability of automated repair to hard and important defects.

Acknowledgements This work is supported by the National Science Foundation under grants CCF-1453474 and CCF-1564162.

Appendix A: Importance and Difficulty Data

Table 1 describes the relevant concrete parameters for each of the bug tracking systems, project-hosting platforms, and defect benchmarks. We omit the semantics of the specific names the various systems and platforms use. This information is available from the underlying bug tracking systems and project-hosting platforms. Table 2 shows the mapping from concrete parameters to abstract parameters and to the five defect characteristics.

Table 1 We used grounded theory to extract from bug tracking systems, project-hosting platforms, and defect benchmarks the concrete parameters relevant to defect importance and difficulty, as well as several other parameters interesting to correlate with automated repair techniques' ability to repair the defect

Issue tracking system	Concrete parameters relevant to importance or difficulty	Other relevant concrete parameters
Bugzilla	importance (priority and severity), target milestone, dependencies (depends on and blocks), reported, modified, time tracking (orig. est., current est., hours worked, hours left, %complete, gain, deadline), priority, components	hardware (platform and OS), keywords, personal tags
FogBugz	priority, milestones, die, subcases	areas, category
GitHub	–	labels
Google code	open, closed, blockedon, blocking, priority, reproducible, star	summary+labels
HP ALM/Quality Center	severity, closing date, detected on date, priority, reproducible, estimated fix time, view linked entities	—
IBM Rational ClearQuest	severity, priority	keywords
JIRA	component/s, votes, watchers, due, created, updated, resolved, estimate, remaining, logged, priority, severity, affects versions, fix versions	environment, labels
Mantis	reproducibility, date submitted, last update	category, profile, platform, OS, tags
Redmine	priority, updated, related issues, associated revisions, start, due date, estimated time(hours)	category
SourceForge	created, updated, priority, milestone	keywords, milestone
Trac	component, priority, milestone	keywords
Defects4J	# of files in the developer-written patch, # of lines in the developer-written patch, # of relevant tests, # of triggering tests, coverage information of test suit	—
ManyBugs	# of files in the developer-written patch, # of lines in the developer-written patch, # of positive tests, # of negative tests	developer-written patch modifications types, defect types

Table 2 Mapping of the concrete parameters from Table 1 to the eleven abstract parameters and then to the five defect characteristics

Defect characteristic	Abstract parameter	Concrete parameters
Importance	**Time to fix**: the amount of time (days) taken by developer(s) to fix a defect. This is computed as the time difference between when the issue was reported and when the issue was resolved. Depending on the issue tracking system, different concrete parameters are used to obtain these two timestamps.	reported, modified, time tracking (orig. est., current est., hours worked, hours left, %complete, gain, deadline), due, created, updated, resolved, estimate, remaining, logged, date submitted, last update, start, due date, estimated time (hours), closing date, detected on date, estimated fix time, opened, closed, milestone
	Priority: importance of fixing a defect in terms of defect priority. This is obtained using priority assigned to the defect. Different issue tracking systems use different values to denote low, normal, high, critical, blocker defects. We use a scale of 1 to 5 corresponding to these priority values (1 is the lowest priority and 5 is the highest) and map the values used by issue tracking systems to our scale. Significance is measured using the number of watchers for a defect, or the number of votes.	priority, importance (priority and severity), watchers, votes, stars
	Versions: effect of defect on different versions of a project or other project modules and components.	components, linked entities, affects versions, fix versions
Complexity	**File count**:the number of files containing non-comment, non-blank-line edits in the developer-written fix	information available from commits on issue tracking systems and helper scripts provided by Defects4J repository.
	Line count: the total number of non-comment, non-blank lines of code in the developer-written fix	information obtained using diff between buggy and fixed source code files. Helper scripts provided with Defects4J
	Reproducibility: how easy it is to reproduce the defect	reproducible, reproducibility
Test Effectiveness	**Statement coverage**: the fraction of the lines in the files edited by the developer-written patches that are executed by the test suite	provided by Defects4J framework
	Triggering test count: number of defect triggering test cases	information provided in `test.sh` script in ManyBugs and triggering_tests in Defects4J
	Relevant test count: number of test cases that execute at least one statement in at least one file edited by the developer-written patch	information provided in `test.sh` script in ManyBugs and relevant_tests in Defects4J
Independence	**Dependents**: number of defects (also with URLs to issue tracking systems) on which the fixing of a given defect depends	Dependencies (depends on and blocks), blockedon, related issues, subcases
Characteristics of the developer-written patch	**Patch characteristics**: characteristics of the developer-written patch in terms of the type of code modifications done to fix the defect	information about bug type available within the ManyBugs metadata

Appendix B: Availability of Data for Annotating Defects

Table 3 describes information about which abstract parameters were available in different issue tracking systems used by ManyBugs and Defects4J projects and how the corresponding concrete parameters were used to annotate the defects. Figure 16 shows the number of defects annotated for each abstract parameter using concrete parameters from bug trackers and benchmarks.

Table 3 Information about abstract parameters obtained from the issue tracking systems

Project	Issue tracking system	Time to fix	Priority	Versions	Dependents	Reproducibility
ManyBugs						
PHP	php bugs	difference between timestamps of "Modified" and "Submitted"	NA	number of values in "PHP Version"	NA	NA
python	python bugs	difference between timestamps of "Last changed" and "created on"	value of "Priority" scaled to our range of [1,5] as: low → 1; normal → 2; high → 3; critical → 4; deferred blocker → 5; release blocker → 5	number of versions in "Versions"	number of values in "Dependencies"	NA
gzip	mail archive	NA	NA	number of versions in "Version"	NA	NA
	Debian bugs	NA	value of "Severity" scaled to our range of [1,5] as: critical → 5; grave → 5; serious → 4; important → 3; normal → 2; minor → 1	number of versions in "Version"	NA	NA
libtiff	Bugzilla Map Tools	difference in timestamps of "Modified" and "Reported"	NA	number of versions in "Version"	value of "Depends on:"	NA
valgrind	KDE Bug tracking System	difference in timestamps of "Modified" and "Reported"	value of "Importance"	number of versions in "Version" field	information in "Dependency tree/graph"	NA

Table 3 (continued)

Project	Issue tracking system	Time to fix	Priority	Versions	Dependents	Reproducibility
lighttpd	Redmine	NA	value of "Priority:" scaled to our range of [1,5] as: low → 1; normal → 2; high → 3; urgent → 4; immediate → 5	values in "Target version:"	NA	NA
Defects4J						
CommonMath	Apache issues	value of "Open → Resolved" field in "Transitions" tab	value of "Priority"	number of versions in "Affected versions" and "fix versions"	number of "Issue Links"	NA
CommonLang	Apache issues	value of,"Open → Resolved" field in "Transitions" tab	value of "Priority"	number of versions in,"Affected versions" and "fix versions"	number of "Issue Links"	NA
JFreeChart	Sourceforge	difference between timestamps of "Updated" and "Created"	value of "Priority" scaled to our range of [1,5] as: 4 → 1; 5 → 2; 6 → 3,7; 8 → 4;9 → 5	NA	NA	NA
JodaTime	Github	difference between timestamps of "Created" and "Last Commit"	NA	NA	NA	NA
	Sourceforge	difference between timestamps of "Updated" and "Created"	value of "Priority" scaled to our range [1,5] as: 4 → 1; 5 → 2; 6 → 3,7; 8 → 4;9 → 5	NA	NA	NA

ManyBugs							
time to fix:	105	file count:	185	statement coverage:	133	dependents:	88
priority:	25	line count:	185	triggering test count:	185	patch characteristics:	185
versions:	111	reproducibility:	0	relevant test count:	185		
Defects4J							
time to fix:	203	file count:	224	statement coverage:	224	dependents:	169
priority:	191	line count:	224	triggering test count:	224	patch characteristics:	224
versions:	169	reproducibility:	0	relevant test count:	224		

Fig. 16 The number of defects annotated for each abstract parameter using the information described in Table 3 and data available in the ManyBugs and Defects4J benchmarks

References

Alkhalaf M, Aydin A, Bultan T (2014) Semantic differential repair for input validation and sanitization. In: International symposium on software testing and analysis (ISSTA), San Jose, CA, USA, pp 225–236

Ammann P, Offutt J (2008) Introduction to software testing, 1st edn. Cambridge University Press, New York

Arcuri A, Yao X (2008) A novel co-evolutionary approach to automatic software bug fixing. In: Congress on Evolutionary Computation, pp 162–168

Bradbury JS, Jalbert KDi Penta M, Poulding S, Briand L, Clark J (eds) (2010) Automatic repair of concurrency bugs. Benevento, Italy

Brun Y, Bang J, Edwards G, Medvidovic N (2015) Self-adapting reliability in distributed software systems. IEEE Transactions on Software Engineering (TSE) 41(8):764–780. https://doi.org/10.1109/TSE.2015.2412134

Brun Y, Barr E, Xiao M, Le Goues C, Devanbu P (2013) Evolution vs. intelligent design in program patching. Tech. Rep. https://escholarship.org/uc/item/3z8926ks, UC Davis: College of Engineering

Brun Y, Medvidovic N (2007) An architectural style for solving computationally intensive problems on large networks. In: Software engineering for adaptive and self-managing systems (SEAMS). Minneapolis, MN, USA. https://doi.org/10.1109/SEAMS.2007.4

Brun Y, Medvidovic N (2007) Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: International workshop on engineering fault tolerant systems (EFTS). Dubrovnik, Croatia, pp 38–43. https://doi.org/10.1145/1316550.1316557

Bryant A, Charmaz K (2007) The SAGE handbook of grounded theory. SAGE Publications Ltd, New York

Carbin M, Misailovic S, Kling M, Rinard M (2011) Detecting and escaping infinite loops with xJolt. In: European conference on object oriented programming (ECOOP). Lancaster, England, UK

Carzaniga A, Gorla A, Mattavelli A, Perino N, Pezzè M (2013) Automatic recovery from runtime failures. In: ACM/IEEE international conference on software engineering (ICSE). San Francisco, CA, USA, pp 782–791

Carzaniga A, Gorla A, Perino N, Pezzè M (2010) Automatic workarounds for web applications. In: ACM SIGSOFT international symposium on foundations of software engineering (FSE). Santa Fe, New Mexico, USA, pp 237–246. https://doi.org/10.1145/1882291.1882327

Charmaz K (2006) Constructing grounded theory: a practical guide through qualitative analysis. SAGE Publications Ltd, New York

Coker Z, Hafiz M (2013) Program transformations to fix C integers. In: ACM/IEEE international conference on software engineering (ICSE). San Francisco, CA, USA, pp 792–801

Dallmeier V, Zeller A, Meyer B (2009) Generating fixes from object behavior anomalies. In: IEEE/ACM international conference on automated software engineering (ASE) short paper track. Auckland, New Zealand, pp 550–554. https://doi.org/10.1109/ASE.2009.15

Debroy V, Wong W (2010) Using mutation to automatically suggest fixes for faulty programs. In: International conference on software testing, verification, and validation. Paris, France, pp 65–74. https://doi.org/10.1109/ICST.2010.66

DeMarco F, Xuan J, Berre DL, Monperrus M (2014) Automatic repair of buggy if conditions and missing preconditions with SMT. In: International workshop on constraints in software testing, verification, and analysis (CSTVA). Hyderabad, India, pp 30–39. https://doi.org/10.1145/2593735.2593740

Demsky B, Ernst MD, Guo PJ, McCamant S, Perkins JH, Rinard M (2006) Inference and enforcement of data structure consistency specifications. In: International symposium on software testing and analysis (ISSTA). Portland, ME, USA, pp 233–243

Durieux T, Martinez M, Monperrus M, Sommerard R, Xuan J (2015) Automatic repair of real bugs: An experience report on the Defects4J dataset. arXiv:1505.07002

Elkarablieh B, Khurshid S (2008) Juzi: a tool for repairing complex data structures. In: ACM/IEEE international conference on software engineering (ICSE) formal demonstration track. Leipzig, Germany, pp 855–858. https://doi.org/10.1145/1368088.1368222

Ernst MD, Cockrell J, Griswold WG, Notkin D (2001) Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering (TSE) 27(2):99–123

Ferguson CJ (2009) An effect size primer: a guide for clinicians and researchers. Prof Psychol: Res Prac 40(5):532–538. https://doi.org/10.1037/a0015808

Fry ZP, Landau B, Weimer W (2012) A human study of patch maintainability. In: International symposium on software testing and analysis (ISSTA). Minneapolis, MN, USA, pp 177–187

Galhotra S, Brun Y, Meliou A (2017) Fairness testing: testing software for discrimination. In: European software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). Paderborn, Germany, pp 498–510. https://doi.org/10.1145/3106237.3106277

Gopinath D, Malik MZ, Khurshid S (2011) Specification-based program repair using SAT. In: International conference on tools and algorithms for the construction and analysis of systems (TACAS). Saarbrücken, Germany, pp 173–188

Harman M (2007) The current state and future of search based software engineering. In: ACM/IEEE international conference on software engineering (ICSE), pp 342–357. https://doi.org/10.1109/FOSE.2007.29

Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In: ACM/IEEE international conference on software engineering (ICSE). Sorrento, Italy, pp 191–200

Jeffrey D, Feng M, Gupta N, Gupta R (2009) Bugfix: a learning-based tool to assist developers in fixing bugs. In: International conference on program comprehension (ICPC). Vancouver, BC, Canada, pp 70–79. https://doi.org/10.1109/ICPC.2009.5090029

Jiang M, Chena TY, Kuoa FC, Towey D, Ding Z (2016) A metamorphic testing approach for supporting program repair without the need for a test oracle. J Syst Softw (JSS) 126:127–140. https://doi.org/10.1016/j.jss.2016.04.002

Jin G, Song L, Zhang W, Lu S, Liblit B (2011) Automated atomicity-violation fixing. In: ACM SIGPLAN conference on programming language design and implementation (PLDI). San Jose, CA, USA, pp 389–400. https://doi.org/10.1145/1993498.1993544

Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the international symposium on software testing and analysis (ISSTA). San Jose, CA, USA, pp 437–440

Ke Y, Stolee KT, Le Goues C, Brun Y (2015) Repairing programs with semantic code search. In: International conference on automated software engineering (ASE). Lincoln, NE, USA, pp 295–306. https://doi.org/10.1109/ASE.2015.60

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: ACM/IEEE international conference on software engineering (ICSE). San Francisco, CA, USA, pp 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893

Kong X, Zhang L, Wong WE, Li B (2015) Experience report: how do techniques, programs, and tests impact automated program repair? In: IEEE international symposium on software reliability engineering (ISSRE). Gaithersburg, MD, USA, pp 194–204. https://doi.org/10.1109/ISSRE.2015.7381813

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge

Langdon WB, White DR, Harman M, Jia Y, Petke J (2016) API-constrained genetic improvement. In: International symposium on search based software engineering (SSBSE). Raleigh, NC, USA, pp 224–230. https://doi.org/10.1007/978-3-319-47106-8_16

Le XBD, Chu DH, Lo D, Le Goues C, Visser W (2017) S3: syntax- and semantic-guided repair synthesis via programming by examples. In: European software engineering conference and ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE). Paderborn, Germany

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012a) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: AMC/IEEE international conference on software engineering (ICSE). Zurich, Switzerland, pp 3–13

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Transactions on Software Engineering (TSE) 41(12):1236–1256. https://doi.org/10.1109/TSE.2015.2454513

Le Goues C, Nguyen T, Forrest S, Weimer W (2012b) Genprog: a generic method for automatic software repair. IEEE Transactions on Software Engineering (TSE) 38:54–72. https://doi.org/10.1109/TSE.2011.104

Le Roy MK (2009) Research methods in political science: an introduction using MicroCase, 7th edn. Thompson Learning, Wadsworth

Liu P, Tripp O, Zhang C (2014) Grail: context-aware fixing of concurrency bugs. In: ACM SIGSOFT international symposium on foundations of software engineering (FSE). Hong Kong, China, pp 318–329

Liu P, Zhang C (2012) Axis: Automatically fixing atomicity violations through solving control constraints. In: ACM/IEEE international conference on software engineering (ICSE). Zurich, Switzerland, pp 299–309

Long F, Rinard M (2015) Staged program repair with condition synthesis. In: European software engineering conference and ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE). Bergamo, Italy, pp 166–178. https://doi.org/10.1145/2786805.2786811

Long F, Rinard M (2016a) An analysis of the search spaces for generate and validate patch generation systems. In: ACM/IEEE international conference on software engineering (ICSE). Austin, TX, USA, pp 702–713. https://doi.org/10.1145/2884781.2884872

Long F, Rinard M (2016b) Automatic patch generation by learning correct code. In: ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL). St. Petersburg, FL, USA, pp 298–312. https://doi.org/10.1145/2837614.2837617

Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2017) Automatic repair of real bugs in Java: a large-scale experiment on the Defects4J dataset. Empirical Software Engineering (EMSE) 22(4):1936–1964. https://doi.org/10.1007/s10664-016-9470-4

Matavire R, Brown I (2013) Profiling grounded theory approaches in information systems research. Eur J Inf Syst 22(1):119–129. https://doi.org/10.1057/ejis.2011.35

Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: looking for simple program repairs. In: International conference on software engineering (ICSE). Florence, Italy

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: International conference on software engineering (ICSE). Austin, TX, USA

Monperrus M (2014) A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In: ACM/IEEE international conference on software engineering (ICSE). Hyderabad, India, pp 234–242. https://doi.org/10.1145/2568225.2568324

Muşlu K, Brun Y, Meliou A (2013) Data debugging with continuous testing. In: European software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE) NIER Track. Saint Petersburg, Russia, pp 631–634. https://doi.org/10.1145/2491411.2494580

Muşlu K, Brun Y, Meliou A (2015) Preventing data errors with continuous testing. In: International symposium on software testing and analysis (ISSTA). Baltimore, MD, USA, pp 373–384. https://doi.org/10.1145/2771783.2771792

Newson R (2002) Parameters behind nonparametric statistics: Kendall's tau, Somers' D and median differences. Stata J 2(1):45–64

Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: program repair via semantic analysis. In: ACM/IEEE international conference on software engineering (ICSE). San Francisco, CA, USA, pp 772–781

Orlov M, Sipper M (2011) Flight of the FINCH through the Java wilderness. IEEE Trans Evol Comput 15(2):166–182

Pei Y, Furia CA, Nordio M, Wei Y, Meyer B, Zeller A (2014) Automated fixing of programs with contracts. IEEE Transactions on Software Engineering (TSE) 40(5):427–449. https://doi.org/10.1109/TSE.2014.2312918

Perkins JH, Kim S, Larsen S, Amarasinghe S, Bachrach J, Carbin M, Pacheco C, Sherwood F, Sidiroglou S, Sullivan G, Wong WF, Zibin Y, Ernst MD, Rinard M (2009) Automatically patching errors in deployed software. In: ACM symposium on operating systems principles (SOSP). Big Sky, MT, USA, pp 87–102, https://doi.org/10.1145/1629575.1629585

Petke J, Haraldsson SO, Harman M, Langdon WB, White DR, Woodward JR (2017) Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolutionary Computation (TEC). In press. https://doi.org/10.1109/TEVC.2017.2693219

Qi Y, Mao X, Lei Y (2013) Efficient automated program repair through fault-recorded testing prioritization. In: International conference on software maintenance (ICSM). Eindhoven, The Netherlands, pp 180–189. https://doi.org/10.1109/ICSM.2013.29

Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: International symposium on software testing and analysis (ISSTA). Baltimore, MD, USA, pp 24–36. https://doi.org/10.1145/2771783.2771791

Schulte E, Dorn J, Harding S, Forrest S, Weimer W (2014) Post-compiler software optimization for reducing energy. In: International conference on architectural support for programming languages and operating systems (ASPLOS). Salt Lake City, UT, USA, pp 639–652. https://doi.org/10.1145/2541940.2541980

Sidiroglou S, Keromytis AD (2005) Countering network worms through automatic patch generation. IEEE Secur Priv 3(6):41–49

Sidiroglou-Douskos S, Lahtinen E, Long F, Rinard M (2015) Automatic error elimination by horizontal code transfer across multiple applications. In: ACM SIGPLAN conference on programming language design and implementation (PLDI). Portland, OR, USA, pp 43–54. https://doi.org/10.1145/2737924.2737988

Smith EK, Barr E, Le Goues C, Brun Y (2015) Is the cure worse than the disease? Overfitting in automated program repair. In: European software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). Bergamo, Italy, pp 532–543. https://doi.org/10.1145/2786805.2786825

softwaretestinghelp.com (2015) 15 most popular bug tracking software to ease your defect management process. http://www.softwaretestinghelp.com/popular-bug-tracking-software/, accessed December 11 2015

Soto M, Thung F, Wong CP, Goues CL, Lo D (2016) a deeper look into bug fixes: patterns, replacements, deletions, and additions. In: International conference on mining software repositories (MSR) mining challenge track. Austin, TX, USA. https://doi.org/10.1145/2901739.2903495

Tan SH, Roychoudhury A (2015) relifix: automated repair of software regressions. In: International conference on software engineering (ICSE). Florence, Italy

Wang X, Dong XL, Meliou A (2015) Data X-Ray: a diagnostic tool for data errors. In: International conference on management of data (SIGMOD)

Wei Y, Pei Y, Furia CA, Silva LS, Buchholz S, Meyer B, Zeller A (2010) Automated fixing of programs with contracts. In: International symposium on software testing and analysis (ISSTA). Trento, Italy, pp 61–72. https://doi.org/10.1145/1831708.1831716

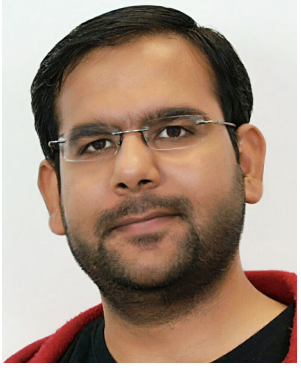
Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: models and first results. In: IEEE/ACM international conference on automated software engineering (ASE). Palo alto, CA, USA

Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: ACM/IEEE international conference on software engineering (ICSE). Vancouver, BC, Canada, pp 364–374. https://doi.org/10.1109/ICSE.2009.5070536

Weiss A, Guha A, Brun Y (2017) Tortoise: interactive system configuration repair. In: International conference on automated software engineering (ASE). Urbana-champaign, IL, USA

Wilkerson JL, Tauritz DR, Bridges JM (2012) Multi-objective coevolutionary automated software correction. In: Conference on genetic and evolutionary computation (GECCO). Philadelphia, PA, USA, pp 1229–1236. https://doi.org/10.1145/2330163.2330333

Yang G, Khurshid S, Kim M (2012) Specification-based test repair using a lightweight formal method. In: International symposium on formal methods (FM). Paris, France, pp 455–470. https://doi.org/10.1007/978-3-642-32759-9_37



Manish Motwani is a PhD student at the University of Massachusetts, Amherst. His primary research area is software engineering and he is interested in improving software engineers' productivity by automating software engineering practices. His research involves analyzing large software repositories to learn interesting phenomena in software development and maintenance, and to use that knowledge to design novel automation techniques, such as testing and program repair. Prior to joining UMass, he worked at the Tata Research Development and Design Center, India where he designed and developed techniques to automate the requirements elicitation and security compliance value chain processes. He earned a BS in Computer Science and Engineering with specialization in Computer Networks from the International Institute of Information Technology, Hyderabad, India.

Sandhya Sankaranarayanan is an engineer in Emerging Solutions and Architecture at VMware. She received her MS from the University of Massachusetts, Amherst in 2017.

René Just is an Assistant Professor at the University of Massachusetts, Amherst. His research interests are in software engineering and software security, in particular static and dynamic program analysis, mobile security, mining software repositories, and applied machine learning. His research in the area of software engineering won three ACM SIGSOFT Distinguished Paper Awards, and he develops research infrastructures and tools (e.g., Defects4J and the Major mutation framework) that are widely used by other researchers.

Yuriy Brun is an Associate Professor with the University of Massachusetts, Amherst. His research interests include software engineering, software fairness, self-adaptive systems, and distributed systems. He received his PhD from the University of Southern California in 2008 and was a Computing Innovation postdoctoral fellow at the University of Washington until 2012. Prof. Brun is a recipient of the NSF CAREER Award in 2015, the IEEE TCSC Young Achiever in Scalable Computing Award in 2013, a Best Paper Award in 2017, two ACM SIGSOFT Distinguished Paper Awards in 2011 and 2017, a Microsoft Research Software Engineering Innovation Foundation Award in 2014, a Google Faculty Research Award in 2015, a Lilly Fellowship for Teaching Excellence in 2017, a College Outstanding Teacher Award in 2017, and an ICSE 2015 Distinguished Reviewer Award.