# Automatic Software Repair: A Bibliography

MARTIN MONPERRUS, University of Lille

This article presents a survey on automatic software repair. Automatic software repair consists of automatically finding a solution to software bugs without human intervention. This article considers all kinds of repairs. First, it discusses behavioral repair where test suites, contracts, models, and crashing inputs are taken as oracle. Second, it discusses state repair, also known as runtime repair or runtime recovery, with techniques such as checkpoint and restart, reconfiguration, and invariant restoration. The uniqueness of this article is that it spans the research communities that contribute to this body of knowledge: software engineering, dependability, operating systems, programming languages, and security. It provides a novel and structured overview of the diversity of bug oracles and repair operators used in the literature.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: Program repair, self-healing software

**ACM Reference format:**
Martin Monperrus. 2017. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (January 2018), 24 pages.
https://doi.org/10.1145/3105906

## 1 INTRODUCTION

This article presents an annotated bibliography on automatic software repair. Automatic software repair consists of automatically finding a solution to software bugs[1] without human intervention. This idea of automatically repairing software bugs is both important and challenging. It is important because software has eaten the world,[2] but, unfortunately, each bite comes with bugs. The software we daily use sometimes crashes, sometimes gives erroneous results, and sometimes even kills people [87]. We do have millions of bugs in the wild, and many of them are being created every day in new software products and releases. To summarize, if automatic software repair could only repair a fraction of those bugs, then it would bring value to society and humanity.

Automatic software repair is challenging because fixing bugs is a difficult task. Of course there are stupid bugs—“blunder” as Knuth puts it [82]—that can be trivially fixed. However, any programmer, whether professional or hobbyist, remembers a bug that took him or her hours, if not days and weeks, to be understood and fixed; these are the “hairiest bugs” [43]. For those bugs, automatic repair is a challenging human-competitive task.

The goal of this article is to draw the big picture of automatic software repair. In particular, it aims at presenting together the two main families of automatic repair techniques: behavioral repair

[1] Automatic repair and tolerance against hardware bugs is out of the scope of this article.

[2] Paraphrasing Silicon Valley’s entrepreneur Marc Andreessen.

Authors’ addresses: M. Monperrus, Centre de Recherche en Informatique Signal et Automatique de Lille, Universitéde Lille 1, 59655 Villeneuve d’Ascq Cedex; email: martin.monperrus@univ-lille1.fr.

https://doi.org/10.1145/3105906

Table 1. The Diverse Terminology of Automatic Software Repair

| Expression | References |
|---|---|
| automatic repair (program repair, self-repair) | [92, 106, 126, 137] |
| automatic fixing (bug fixing, program fixing) | [102, 185] |
| automatic patching | [80, 101, 157, 189] |
| healing (self-healing) | [56, 155, 158] |
| automatic correction (self-correcting) | [79, 90] |
| automatic recovery (self-recovering) | [21, 168] |
| resilience | [28, 155] |
| automatic workaround | [24] |
| survive (survival, survivability) | [123, 142, 148] |
| rejuvenation | [65] |
| biological metaphors: allergies, immunity, vaccination | [73, 105, 142, 159] |

and state repair. The former is about automatically modifying the program code; the latter is about automatically modifying the execution state at runtime. The primary intended audience consists of researchers in computer science, with a focus on the research communities that contribute to this body of knowledge: software engineering, dependability, operating systems, programming languages, and software security. Each section also provides an introductory explanation of the key concepts behind automatic repair, which could be of high interest for practitioners and curious students. This survey aims at covering all important works in the field or automatic software repair, with an emphasis on empiricism: the covered technique must apply to some programs done in industry and bugs that happen in practice. Works are included as follows: For each article, the importance is qualified according to the visibility and reputation of the venue or the novelty of the idea presented in the article. If several articles contain the same idea, then only the most representative one is discussed and cited. It is to be noted that the same concept "repair" has several names in the literature: "patch," "fix," "heal," "recover," and so on. Table 1 lists the main ones, as well as notable references that use the term. In this article, the name "repair" is chosen, because a program has something mechanical in nature, which fits well the daily usage of the word "repair." Also, it is the name used by the most excellent articles in the field.

To my knowledge, there is no comparable bibliography in the literature. The review article by Le Goues et al. [93] is close but only covers a fraction of the articles and only on behavioral repair. On the contrary, Rinard [148] only focused on runtime repair. Yet, there are surveys in related fields, for instance, for fault-tolerance [178], fault localization [153, 190], and algorithmic debugging [162], to only name a few.

In sum, the contribution of this article is a survey on automatic software repair:

- This survey is across different research areas and includes contributions from the following communities: software engineering, dependability, operating systems, programming languages, and software security. Similarly, this survey encompasses different terminologies (automatic repair, self-healing, automatic recovery, etc.).
- This survey provides the reader with an in-depth analysis of the literature according to the type of repair they perform (behavioral versus state repair) and the oracle they consider.

The remainder of this article reads as follows. Section 2 briefly presents the core concepts of automatic repair. Section 3 discusses the main approaches of behavioral repair, and Section 4 is about state repair. Section 5 is dedicated to the empirical works that aim at understanding the foundations of automatic repair. Section 6 is an account on articles that are not directly about automatic repair yet have a close connection.

## 2 CORE CONCEPTS FOR AUTOMATIC REPAIR

Automatic repair is about bugs. The literature is full of synonyms for "bug": "defect," "fault," "error," "failure," "mistake," and so on. There are rather accepted definitions among faults, errors, and failures [9]: A *failure* is an observed unacceptable behavior, an *error* is a propagating incorrect state prior to the failure (without yet having been noticed), and a *fault* is the root cause of the error (in particular, incorrect code). Although there is relative clarity among those three concepts, one can hardly say that the literature, including the most recent articles, sticks to those definitions. Furthermore, if we only consider the repair literature, then there is absolutely no emerging separation among "automatic repair of failures," "automatic repair of errors," and "automatic repair of faults." However, we need a common concept for all words, and in this article, the term "bug" is used as an umbrella word because of its intuitiveness and wide usage, with the following definition: *A bugis a deviation between the expected behavior of a program execution and what it actually happened.*[3]

This definition of bug involves the notions of "behavior," "execution," and "program," but has an implicit third subject: the observer, or reference point, that deems the behavior unexpected. This "observer" can obviously be a human user saying, "this output is not correct." It is also classically a *specification*, in its most general meaning: A specification is a set of expected behaviors. Specifications are polymorphic: They can be natural language documents, formal logic formulas, test suites, and so on. They can even be implicit: For instance, the specification "the program shall not crash on any input" holds for many programs while not often being explicitly written. To some extent, the user saying "this output is not correct" is stating the specification on the fly. Consequently, automatic repair always refers to a specification and yields the following definition of automatic repair: *Automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification.*

A concept that is close to the one of specification is the one of *oracle*. Simply put, an oracle determines whether the result of executing a program is correct [169]. To this extent, specification and oracle refer to the same thing: expectation, acceptability, and correctness. However, there is a major difference between both. An oracle is only a part of specifications; it is the part related to the expected output (when one such exists). In addition, a specification contains information about the input ranges, about non-functional properties, and so on. For instance, a test suite is a specification; it contains test cases, which themselves contain assertions, the latter being the oracles.

With respect to repair, the oracles can be split in two: the *bug oracle* refers to the oracle that detects the unexpected behaviors, and the *regression oracle* refers to the oracles that check that no new bugs have been introduced during repair. The reason is that the program on repair already satisfies all regression oracles, but a repair transformation may accidentally introduce a regression. There are more formal definitions of specification and oracle in the literature [11, 169], but they do not bring much to the context of this article.

Finally, a repair technique often targets a *bug class.*[4] A bug class is an abstract concept referring to a family of bugs that have something in common: the same symptoms, the same root cause, or the same solution [125]. For instance, well-known bug classes include off-by-one errors, memory

[3] Note that some authors use "intended" instead of "expected"; the latter is taken because it is really the viewpoint of the user or client that matters, not the viewpoint of the engineer who designed and developed the software.

[4] or fault class, or error class, and so on.

Table 2. Examples of Repair Operators for Behavioral Repair

| Operator | References |
|---|---|
| add/remove/replace code | [6, 189] |
| add a precondition | [34, 101] |
| replace a condition | [34, 129] |
| replace assignment RHS | [58, 74, 129] |
| addition or removal of method calls | [31] |
| adding a modulo for array read, truncating data for array write | [101] |

leaks, and so on. However, there are many bug classes for which there are no clear definition and scope in the literature, and some of them even miss a name. While some initial taxonomies exist [41, 180], building a comprehensive taxonomy of bug classes will require years of research.

## 3 BEHAVIORAL REPAIR

> Behavioral repair consists of changing the behavior of the program under repair, i.e., changing its code. The modification can be done on source code but also on binary code (e.g., Java bytecode or x86 native code). Behavioral repair can be done offline or online at runtime.

When done offline, behavioral repair may happen in the development environment (IDE) of maintenance developers or in a continuous integration server. Online behavioral repairs means repair done on deployed software. Technically, behavioral repair at runtime involves a kind of dynamic software update (DSU), which is a research topic per se.

Behavioral repair involves a *repair operator*,[5] which is a kind of modification on the program code. For instance, one repair operator is the addition of a precondition, as shown below:

```
+ if (age>=18)
   serve_adult_content()
```

The literature defines many different repair operators that will be presented below and that are summarized in Table 2. Sometimes the repair operator involves a *repair template*,[6] which is a parameterized snippet of code that targets the repair a specific bug class. A *repair model* [115] is a set of repair operators.

For instance, when considering a test suite as specification 3.1.1, a problem statement of behavioral repair is *given a program and its test suite with at least one failing test case, create a patch that makes the whole test suite passing*. This problem statement can be called test-suite based repair [125] and has been famously explored by Genprog, presented in Section 3.1.1

### 3.1 Repair and Oracles

As presented in Section 2, automatic repair is with respect to an oracle. Consequently, this section is organized according to the kind of oracle considered in the literature.

*3.1.1 Test Suites.* A test suite is an input-output–based specification. In modern object-oriented software, the input can be as complex as a set of interrelated objects built with a rich sequence of method calls, and the output can also be a sequence of method calls that observe the execution

[5] or “repair action.”

[6] or “repair strategy” or “fix schema” [185].

state and behavior in various ways. In test-suite-based repair, the failing test case acts as a bug oracle, and the remaining passing test cases act as a regression oracle.

Genprog is a seminal and archetypal test-suite-based repair system developed at the University of Virginia [46, 187, 189]. Genprog uses three repair operators that are mutations over the abstract syntax trees (AST): deletion of AST nodes, addition of AST nodes, and replacement of existing nodes. For addition and replacement, the nodes are taken from elsewhere in the code base. This is called the redundancy assumption [12, 116]. Genprog is able to handle real-world large-scale C code. The largest evaluation of Genprog [92] claims that 55 of 105 bugs can be fixed by Genprog. Those results have been later questioned, as discussed in Section 5. The Genprog thread of ideas yielded other articles in the original team [95, 152] and other laboratories [139, 140]. Now that the core ideas of Genprog are well known and accepted, work needs to be done to improve the core repair operators (such as Reference [133]).

Before Genprog, in the mid-1990s, Stumptner and Wotawa [170] proposed automatic repair in a simple toy language called EXP. The specification is a set of test cases (i.e., a test suite). To my knowledge, it is the first occurrence of test-suite-based repair in the literature.

Arcuri [5–7] defines seven repair operators based on abstract syntax tree modification. For instance, for "promote mutation," a node is replaced by one of its children. The operators are stacked in a random way. The prototype implementation, called Jaff, handles a subset of Java and is evaluated on toy programs.

Debroy and Wong [33, 131] propose to use standard mutations from the mutation testing literature to fix programs. Consequently, their repair models are as follows: replacement of an arithmetic, relational, logical, increment/decrement, or assignment operator by another operator from the same class and decision negation in an if or while statement. Conventionally, they locate fault statements with spectrum-based fault localization technique. Nica et al. [131] also use mutations for repair. Compared to Debroy and Wong, they comprehensively explore the space of all mutations.

The key idea of Kern and Esparza [79] is to generate a meta-program that integrates all possible mutations according to a mutation operator. The mutations that are actually executed are driven by meta-variables. A repair is a set of values for those meta-variables. The meta-variables are valued using symbolic execution.

NGuyen et al. [129] proposed an approach called Semfix for repair based on symbolic execution and code synthesis. The location of the repair is found with angelic debugging [25], and then the repaired expression is synthesized with input-output component synthesis [69]. The repaired locations are the right-hand side (RHS) of assignments and Boolean conditionals, and the synthesized expressions mix arithmetic and first-order logics. One problem with Semfix is scalability. To overcome this problem, the same group has proposed Angelix [118]. Angelix is a repair system like Sefix, where the symbolic execution phase has been seriously optimized to scale to large programs and obtain more than one angelic value, this is an "angelic forest."

The PAR system [80] is an approach for automatically fixing bugs of Java code. PAR is based on repair templates: Each of PAR's 10 repair templates represents a common way to fix a common kind of bug. For instance, a common bug is the access to a null pointer, and a common fix of this bug is to add a nullness check just before the undesired access: This is the template "Null Pointer Checker." Some templates are parameterized by variables; for instance, the "Null Pointer Checker" template takes a variable name as a parameter. The templates are applied and tested in a random search manner.

Nopol [34] targets a specific fault class: conditional bugs. It repairs programs by either modifying an existing if-condition or adding a precondition (a.k.a. a guard) to any statement or block in the code. The modified or inserted condition is synthesized via input-output–based code synthesis with Satisfiability Modulo Theory (SMT) [69] and predicate switching [194]. The Nopol system has been extended for also repairing infinite loops [88].

Tan and Roychoudhury proposed Relifix, a repair system dedicated to fixing regression bugs [175]. The approach consists of eight repair templates, some being transformation operators and the others being parameterized repair templates. The key idea of Relifix is that the template applications are driven by the past changes, for instance, the template "add statement" only add statements that were involved in the previous commits related to the regression.

Mechtaev et al. also perform test-suite-based repair [117], with the noble goal of synthesizing simple patches. To do so, they assume a very specific kind of program: those that can be expressed as trace formulas (related to Boolean programs of Reference [59]). Under this assumption, they can state the repair problem as a Maximum Satisfiability (MaxSAT) problem, where the smallest patch is the one that satisfies the most constraints.

SPR (Staged Program Repair) [109] defines a set of staged repair operators to discard early many candidate repairs that cannot pass the supplied test suite. This allows for exhaustively exploring a small and valuable search space.

The idea of CodePhage [160] is to transfer a check from one application to another application to avoid crashes. The system assumes an error-triggering input that crashes one application but not the other one. The considered errors are out-of-bounds access, integer overflow, and divide-by-zero errors. The missing check is inferred from a symbolic expression over the input fields and validated by a regression test suite.

Ke and colleagues proposes SearchRepair [78], a system inspired from code search. SearchRepair first indexes code fragments as SMT constraints, then, at repair time, a fragment is retrieved by combining the desired input-output pairs and the fragments in a single constraint problem. The system is evaluated on small C programs written by students in an online course.

Prophet [110] is a repair system that uses past commits to drive the repair. What is learned on past commits from version control systems is a probability distribution over a set of features of the patch. This probability distribution is then used to both speed up the repair and increase the likelihood to find correct patches. The evaluation is done on 69 real-world defects from the Genprog benchmark and shows that 15 correct repairs are found. Le et al. [91] also use history to select the most likely patch. Contrary to Prophet, the experiments were made on Java programs.

*3.1.2 Pre- and Post-Conditions.* Some works use classical pre- and post-conditions *á la* design-by-contract [124] as an oracle for repair.

He and Gupta [62] use pre- and post-conditions to compute "hypothesized program states" (from the post condition) and "actual program states" (from the failing input). The repair operators consist of changing the LHS or RHS of assignments, or changing a Boolean condition with simple modifications (change variable, change relational operator) so that the hypothesized program state becomes compatible with the actual program state. A classical test suite is used for detecting regressions.

AutoFix-E is an approach by Wei et al. [185, 193] that generates fixes for Eiffel programs, relying on contacts (pre-conditions, post-conditions, invariants). AutoFix-E uses four repair templates that consist of a snippet and an empty conditional expression to be synthesized. The key intuition behind AutoFix-E is that both the snippet code and the conditional expression are taken from the existing contracts.

Gopinath et al. [58] uses pre- and post-conditions written in the Alloy specification language. The function body is also translated to Alloy formulas. Then, the bounded verification mechanism of Alloy is used both to detect bugs (similarly to Reference [66]) and to identify the repair. The repair operators are changing the RHS of assignments and modifying existing if-conditions.

Könighofer and Bloem [84] considers assertions as specifications in programs that can be translated to SMT. The approach is static and the repair is shown to not violate the assertion for the considered input domain. The approach is based on repair templates, such as changing the RHS of assignments or changing an arithmetic expression by a linear combination. The templates holes are filled by the SMT solver.

*3.1.3 Abstract Behavioral Models.* An abstract behavioral model, such as a state machine encoding the object state and the corresponding allowed method calls can be used to drive the repair.

In 2006, before Genprog, Weimer [186] proposed a first-patch-generation technique. It requires as input a safety policy (i.e., a typestate property or an Application Programming interface (API) usage rule) and the control-flow graph of a method. The whole approach is static: The bug is detected as a static violation of the safety property, and the correctness condition of the patch is only to pass the safety check. Interestingly, the word does not mention the term "repair," as it was not in the zeitgeist at this time.

Dallmeier et al. [31] presented Pachika, an approach for repairing Java programs. The idea of Pachika is to first infer an object usage model from executions and then to generate a fix for failing runs to match the inferred expectedly correct behavior. The evaluation consists of fixing 18 bugs of ASPECTJ (75KLOC) and 8 of RHINO (38KLOC). The two repair operators of Pachika are addition and removal of method calls. The main difference with the previous approach is that the behavioral model is mined and not given.

## 3.2 Static Analysis

Static analysis tools outputs errors and warnings. It is possible to automatically repair them. In this case, the correctness oracle is the static analysis itself.

Logozzo and Ball [106] proposes a repair approach on top of their static analysis toolchain for .Net code. For a set of fault class identified statically (e.g., off-by-one errors), they propose a corresponding repair operations. The repair operators are specific to each fault class, for instance, it is adding a precondition, changing the size of an array allocation, and so on. The static analysis is run again to verify the correctness of the repair.

Logozzo and Martel [107] targets a specific fault class in integer arithmetic (linear combinations). The arithmetic overflow is detected statically, and the suggested fix is a re-ordering of the arithmetic operations. The fix ensures that the overflow cannot happen anymore. On arithmetic overflows, there is also the work by Cocker et al. [27].

Gao et al. [50] present an approach for automatically fixing memory leaks for C programs. The approach consists of statically detecting and fixing memory leaks by inserting a deallocation statement. The evaluation is done on 14 programs in which 242 allocations are considered.

Gupta et al. [60] devise an approach for repairing compiler errors, which is a static oracle. The originality of DeepFix is to use a language model based on deep learning to suggest fixes. They evaluate their approach by repairing student programs from an online course.

Muntean etl al. [127] statically detects buffer overflows. Then they have templates parameterized by a variable. The correct variable to be used in the template is found using SMT.

## 3.3 Crashing Inputs

Behavioral repair can happen as a response to a field failure (e.g., a crashing exception or a SegFault caused by a buffer overflow). The repair process happens once the crashing input has been identified and minimized if possible. The failing test case of a test suite can also be seen as a crashing input. However, the main difference of crashing inputs and test suites from the viewpoint of oracle for repair is the following. A test suite also contains passing test cases (the regression oracle), and that failing test case contains assertions on the expected value, while crashing inputs, as their name suggests, only refer to a violation of the non-functional contract “the program shall not crash.”

Gao et al. [51] repairs crashing exceptions based on Stackoverflow. Their system, called QACrashFix, mines pairs of buggy and fixed code on Stackoverflow, to extracts an edit script. The edit scripts are tried in sequence to suppress the crashing exception. Azim et al. [10] detect field failures on Android smartphone applications. The considered faults are unhandled exceptions, the repair operator consists of adding try/catch blocks with binary rewriting.

Clotho [39] is a system that generates simple catch blocks to handle certain runtime exceptions related to string manipulation in Java. The content of the catch block is based on constraints that are collected both statically and dynamically.

Sidoroglou and Keromytis [157] detect buffer overflow vulnerabilities at runtime in production, then they obtain the source of the vulnerability through the use of ProPolice [45]; finally, they use code transformation rules written in the transformation language TXL to modify source code. Regressions are caught by manually provided test suites.

Lin et al. [101] tries to generate a source code patch from a working exploit that triggers an array overflow in C code. Its repair operators consist of fixing out-of-bound reads by adding a modulo in the read expression and out-of-bound writes by truncating data to be written (similarly to failure-oblivious computing).

Wang et al. [182] target automatic repair of integer overflows. They have three repair operators. The first one is to force taking an error branch before the overflow happens, the second one is to force taking an error branch after the overflow has happened, and the last one is a program stop (exit). The generated conditions are path conditions obtained from dynamic symbolic execution.

### 3.4 Other Oracles

Other specific oracles have been used in an automatic repair setting.

A number of techniques have been proposed to fix concurrency bugs. Jin et al. [71] present AFix: The repair model of AFix consists of putting instructions into critical regions. This work on automatic repair of concurrency bugs has been further extended [104]. Lin et al. [100] also insert locks by encoding the problem as a satisfiability one. In Dfixer [17], no new locks are introduced to repair concurrency bugs; instead, existing locks are pre-acquired in one thread. More recently, Liu et al. [103] have proposed another repair operator for concurrency bugs in a tool called HFix: They propose to automatically add thread-join operations.

Samimi et al. [151] have presented an approach for repairing web application in the PHP programming language that generates HTML tags. The oracle that is used is whether the output HTML string is malformed, i.e., that it does not contain a inconsistent sequence of opening and closing tags (e.g., “`<a></i></a>`”). They encode the repair as a constraint problem on strings. Wang et al. [183] also repairs the HTML code output by PHP code, using runtime tracing instead of constraint solving. Medeiros et al. [119] also repairs web applications, but consider SQL injection, and their repair operator consists of wrapping certain call by a sanitization function.

Liu et al. [102] uses as oracle a manually written bug report. They have parameterized repair templates and extracted the actual value of the template parameter from the bug report. For instance, for a not-null checker template, they extract the name of the variable to be checked from the bug report.

Dennis et al. [38] uses proof-based program verification on ML programs using Isabel as an oracle. When the proof fails, the counter-example of the proof drives a repair approach based on repair templates (replacing one method call by another, adding some code).

It is possible to use a reference implementation as specification for repair. In this case, the reference implementation both acts as the bug oracle (when the behavior of the reference implementation and of the buggy program do not correspond) and as a regression oracle. This has been little explored in the context of repair. The approach by Könighofer and Bloem [85] uses SMT-based templates. The approach by Singh et al. [163] is conceptually similar but is realized differently, and the evaluation is much larger. The reference implementation and the program to be repaired are written in Python. The system translates them to a programming environment called Sketch, which is responsible for exploring the space of candidate fixes. The evaluation is made on thousands of buggy programs submitted for an online course. Qlose [42] is a similar approach based on Sketch; the novelty of Qlose is that it tries to semantic impact of the repair by minimizing the number of inputs for which there is a behavioral change.

Jiang et al. [70] have proposed to use metamorphic relations as a repair oracle. They evaluate their approach on the Introclass benchmark made of student programs. Due to the limited size of their experimental subjects, it is yet to be proven that metamorphic relations can help repair large and real programs. Kneuss et al. [81] use a kind of symbolic test for repairing a purely functional toy language. As metamorphic relations, the symbolic tests enable to generate new test data.

### 3.5 Domain Specific Repair

The concept of automatic repair can be applied on many computational artifacts. Indeed, there are many works doing automatic repair in contexts that are specific to an application domain.

Lazaar et al. [90] repair constraint programs. With a domains-specific fault localization strategy, the repair consists of removing or adding new constraints. Gopinath et al. [57] repair database selection statements in a specific data-oriented language called Abap. Kalyanpur et al. [76] state an automatic repair problem in the context of Web Ontology Language (OWL) ontologies. Griesmayer et al. [59] repair a specific class of programs called Boolean programs: those that only contain Boolean variables. Further work has been done on repairing Boolean programs [150]. Son et al. [167] repairs access-control policies in web applications using a static analysis and transformations tailored to this domain.

Nentwich et al. [128] detect inconsistencies and propose repair actions on XML documents. Their approach is applicable to all structured documents with explicit static inconsistency rules. Along the same line, Xiong et al. [191] detect and fix inconsistencies in Meta-object facility (MOF) and Unified Modeling Language (UML) models; da Silva [30] uses Prolog to propose a repair plan that fixes inconsistencies in UML models; and Xiong et al. [192] focuses on automatically repairing configuration errors in software product lines.

Tran et al. [179] uses repair in the sense of forcing a match between source code dependencies and a dependency model that specifies the acceptable dependencies; this can be called "architectural repair."

The approach of Daniel et al. [32] does not repair programs but the test cases that are broken in the presence of refactoring. Memon [120] and Gao et al. [52] repair GUI test scripts. For instance, the approaches change the identifiers that are used for driving the GUI manipulation. Leotta et al. [96] do test repair in the context of Selenium tests, which are tests for web applications with HTML output.

### 3.6 Fault Classes and Repair

Some fault classes are well-enough understood so that one can write a code transformation that suppresses all instances of the fault class at once. For instance, one can transform 64-bit integers to unlimited precision arithmetic objects (such as BigInteger in Java) to avoid all arithmetic overflows. In the related work, most repair transformations for fault classes are semantic preserving but not necessarily.

For instance, a seminal work on semantic modifying transformations is failure-oblivious computing [145]. Considering erroneous reads out of the bounds of an array, failure-oblivious computing transforms the code so that the read returns either the first non-null element or the element modulo the length of the allocated array. Along the same line, Rinard et al. [146] proposes that out-of-bounds writes are stored in a hashtable and that subsequent reads to the out-of-bound index return the object previously stored in the hashtable. This line of research is based on the philosophical foundation than acceptable results is more important than correct results, this is called "acceptability-oriented computing" [144].

Thomas and Williams [177] propose an approach to automatically transform PHP code to secure SQL statements. The transformations modify the abstract syntax trees to inject secured "prepared statements."

At Google, they develop and use a tool called "error-prone" [2],which does automatic repair of Findbugs-like errors [64]. Lawall et al. [89] also defined an approach for declaratively specifying bug patterns and the corresponding patches in a tool called Coccinelle. The same idea has been developed by Kalval and Warburton [75], where the repair strategy is written using a formal transformation language called Trans.

Shaw et al. [154] describe two transformations to fix C buffer overflows: replacement of unsafe calls by alternative safe libraries and replacement of unsafe types by safer ones. They show that the transformations scale to large programs, do not break the existing tests, and do not slow down the programs. Coker and Hafiz employ a similar approach for another fault class: integer arithmetic bugs [27]. They propose three program transformations dedicated to integers and show that the approach scales to real programs.

Long et al. [111] uses a static analysis specific to integer arithmetic that detects integer overflow. For all detected potential overflows, the system infers a filter that simply discards the input. To this extent, the repair action is denying the input, a technique also done at runtime and discussed in Section 4.5.

Cornu et al. [28] target unhandled exceptions in Java. They analyze test suite executions to identify the good catch blocks that have resilience capabilities. Then they transform the caught exception type into a more generic one (i.e., a superclass exception) to catch exceptions that would not be caught otherwise. The code transformation, called "catch stretching," is a kind of proactive repair against unexpected exceptions.

## 4 STATE REPAIR

> State repair consists of changing the state of the program under repair. The state is meant in its largest acceptation: It can be changing the input, the heap, the stack, or the environment. For instance, automatic breaking a cycle in a linked list is one kind of state repair. As opposed to behavioral repair, state repair is necessarily done at runtime.

State repair consists of changing the state of the program under repair. The state is meant in its largest acceptation: It can be changing the input, the heap, the stack, or the environment. For instance, automatic breaking a cycle in a linked list is one kind of state repair. As opposed to behavioral repair, state repair is necessarily done at runtime.

State repair can be rooted in classical fault tolerance [9]. In this large research field, much research has targeted "recovery," which Avizienis et al. defines as transforming *"a system state that contains one or more errors and (possibly) faults into a state without detected errors"* [9]. In this article, the term "state repair" is used instead of "recovery." This terminological move allows us to have an umbrella term, "repair," above intrinsically related concepts (recovery, resilience, etc.) and

Table 3. Examples of State Change Operators for Runtime Repair

| Operator | References |
|---|---|
| restart | [18, 184] |
| try an alternative implementation | [8, 23, 143] |
| modify the input | [3, 99, 108] |
| simulate a known error (aka error virtualization) | [105, 158] |
| change the execution environment | [54, 130, 142] |

above behavioral and state repair, see Figure 5.1 of Reference [9] for a bird's eye presentation of classical recovery, error handling. and fault handling.

State repair requires an oracle of the bug, an oracle of incorrectness. As opposed to behavioral repair, those oracles have to be available in production at runtime. This rules out certain oracles discussed in Section 3, such as test suites, and oracles based on static analysis. For state repair, there are three main families of bug oracles.

First, state repair often considers violations of non-functional contracts. For instance, crashing with a Segfault or a null pointer exception violates the non-functional contract "the program shall never crash." Second, state repair can also consider functional contracts that are verifiable in production such as pre- and post-conditions. This will be much discussed in Section 4.7.1. Third, there are state repair approaches that reason on "inferred contracts," obtained by observing the regularities of program states at runtime. In this case, a bug is defined as a program state or behavior that violates those inferred contracts, and repair is a follow-up of anomaly detection on program states and executions.

In the following, the approaches are ordered by repair operators. This more fits to the history of the field than the ordering by kind of oracles, as what was done for behavioral repair.

### 4.1 Reinitialization and Restart

Restarting (a.k.a. rebooting) a software application is the simplest repair action.

It has been much explored under the term "software rejuvenation" [65] but rather with a theoretical stance rather than a practical one.

Candea and colleagues [18, 20, 21] explored in depth the concept of microreboot. Microreboot consists of having a hierarchical structure of fine-grain rebootable components and, in the presence of failures, to try to restart the application from the smallest component (an Enterprise Java Bean (EJB)) to the biggest one (the physical machine) (in a way that is similar to progressive retry in distributed computing [184]). Their experiments show that this can significantly improve the availability of systems.

### 4.2 Checkpoint and Rollback

A checkpoint and rollback mechanism takes regular snapshots of the execution state and is capable of restoring them later. The challenges of checkpoint and rollback are, first, the size and boundaries of the captured state and, second, the point in time of checkpointing [77, 86]. When a system is equipped with a checkpoint and rollback mechanism, the rollback is the repair. Despite being an old technique, it is valuable in a number of contexts.

Dira [164] is a system that instruments code to detect and recover from control-hijacking attacks through malicious payloads. The repair consists of finding the least common ancestor of the function in which the attack is detected and the one in which the payload was read in. Then, the execution is resumed to this frame and all state changes are undone. Similarly, Assure [158] is a technique also based on checkpointing to provide self-healing capabilities. Recent articles also do checkpoint and rollback as part of the repair, such as in Reference [23].

### 4.3 Alternatives

Another classical concept of fault-tolerance is n-version programming. Either with voting [8] or retrying with recovery blocks [143], it consists of relying on alternative implementations to recover from errors. This concept is now explored using natural sets of alternatives (as opposed to being engineered) or with automatically created sets of variants [16].

For instance, Carzaniga et al. [24] repair web applications at runtime with a repair strategy that is based on a set of API-specific alternative rules: For instance, calling `bar()` instead of `foo()`. They later applied the same idea for recovering from runtime exceptions in Java [23]. Hosek and Cadar [63] use a different kind of natural diversity: On failures, they switch from past or newer versions of the same application. The key idea is that bugginess is not monotonic: Some bugs disappear while others appear over time.

### 4.4 Reconfiguration

Reconfiguring an application is one kind of recovery [9] and, thus, one kind of state repair. Indeed, it has much been explored when "self-healing" was a hype term. For instance, Cheng et al. [26] use the three core runtime reconfiguration operators (add component, move component, delete component) to optimize quality-of-service values. The same line of repair can be found in References [53, 156], which are relatively cited articles. In the context of web service orchestration, the repair actions of Friedrich et al. [47, 136] consist of substituting it a web service by another (which is a reconfiguration) and retrying a service call.

### 4.5 Input Modification

If the system fails on some input, then one state repair action consists of modifying the input. Denying the input is also a possible option, which can be considered as an extreme case of input modification.

Ammann and Knight's "data diversity" [3] aims at enabling the computation of a program in the presence of failures. The idea of data diversity is that, when a failure occurs, the input data are changed so that the new input resulting from the change does not result in a failure. The assumption is that the output based on this artificial input, through an inverse transformation, remains acceptable in the domain under consideration.

Long et al. [108] present the idea of automated input rectification: Instead of refusing anomalous inputs, they change it so that it fits into the space of typical and acceptable inputs; this is called "input rectification."

Liand and Sekar [99] repair buffer overflows by learning common profiles between the characteristics of crashing inputs. Once a valid profile is identified, crashing inputs are denied. While the article is about security, it can be seen as a runtime technique to repair memory errors of the form of buffer overflows. The fact that the buffer overflow is accidental (due to a bug) or maliciously triggered is irrelevant from a repair perspective. Along the same line of input denying, Vigilante [29] is an integrated approach for mitigating malicious attacks. The counter-measure to worm attacks is filtering: Once invalid or malicious inputs are detected, they are filtered out and the current request or task is aborted.

### 4.6 Environment Perturbation

If the system fails under certain conditions, then one can get the next requests to succeed by changing the runtime environment (e.g., the memory, the scheduling) or the configuration.

Qin et al. [142] shows that memory errors can be avoided by padding allocated memory blocks with extra space. Berger and Zorn [13] do the same thing and add replication. However, the difference with Rx is that their system allows for probabilistic reasoning on the resulting memory safety. Novark et al. [132] explores the same idea. Differently, Nguyen and Rinard [130] enforces a bounded memory size by cyclic memory allocation in a way that is similar to failure-oblivious computing (already presented in Section 3.6). Garvin et al. [54] address configuration bugs and propose "reconfiguration workarounds" that change the configuration causing a failure.

Jula et al. [73] presents a system to defend against deadlocks at runtime. The system first detects synchronization patterns of deadlocks, and when the pattern is detected, the system avoids reoccurrences of the deadlock with additional locks.

Tallam et al. [174] names this family of technique "execution perturbations." For concurrency and memory bugs, they show that removing thread interruptions, padding memory allocations, and performing denial of requests is a way to avoid failures.

## 4.7 Rollforward

Rollforward (or forward recovery) means transforming the current system state into a correct one. There are several techniques of forward recovery: invariant restoration, error virtualization, and so on.

### 4.7.1 *Invariant Restoration.*

In some cases, state correctness can be expressed as an invariant. Consequently, repair means restoring the invariant, if possible, with a minimum of changes from the current erroneous state.

Demsky and Rinard [36] uses a specification language to express correctness properties on data structures. This specification is then used at runtime to automatically repair broken data structure (concrete instances at runtime, not the abstract data type). Elkarablieh et al. [44] also automatically repair data structures at runtime, and the difference with Demsky and Rinard is that they rely on an invariant written in regular Java code (a "repOK" Boolean method).

Perkins et al. [135] presented ClearView, a system for automatically repairing errors in production. The system works on low level x86 binaries and consists of monitoring the system execution to learn invariants. Those invariants are then monitored, and a violation is followed by a forced restoration. The repairs are at the level of CPU registers and memory location changes.

Lewis and Whitehead [98] have a generic repair approach for event-based systems by defining a runtime fault-monitor, but the core idea is the same: When an invariant is violated, the repair system automatically restores it. The example in a video-game domain is fun: If Mario is hanged in the sky due to specific sequence of actions and interactions, it is forcefully put back on the ground. Beyond data structures and video-games, in real systems, many strange and undesired system states can happen from complex chains of events and interactions, but it is often possible to state simple invariants to guide runtime repair.

### 4.7.2 *Error Virtualization.*

Error virtualization consists of handling an unknown and unrecoverable error with error-handling code that is already present in the system yet designed for handling other errors.

This idea has been much explored at Columbia University. For instance, Sidiroglou et al. [159] do error virtualization in system that imitates biological immunity. They combine error virtualization with selective transactional emulation, a technique consisting of emulating the execution of native code with an interpreter in a transactional manner. When a failure occurs in an emulated section, all state changes are undone (a kind of micro rollback at the level of functions). In Assure [158], the idea of error virtualization is associated with fuzzing to discover and test in advance valuable error virtualization points, called rescue points.

*4.7.3 Other Forward Recovery.* Carbin et al. [22] introduced a system that monitors programs to detect infinite loops and escape them. The system works with binary code instrumentation and breaks the loops with no memory state changes detected during their execution. Along the same line is the concept of "loop perforation" [161]. Sidiroglou et al. have shown [161] that it is possible to skip the execution of loop iterations in certain application domains. For instance, in a video decoding algorithm (codec), skipping some loop iterations only has an effect on some pixels or contours but does not completely degrade or crash the software application. On the other hand, skipping loop iterations is key with respect to performance. In other words, there is a tradeoff between the performance and accuracy. This tradeoff can be set offline (e.g., by arbitrarily skipping one every two loops) or dynamically based on the current load of the machine.

Dobilyi and Weimer [40] target repair of null pointer exceptions. Using code transformation, they introduce hooks to a recovery framework. This framework is responsible for forward recovery of the form of creating a default object of an appropriate type to replace the null value or of skipping instructions.

Long et al. [112] introduces the idea of "recovery shepherding." Upon certain errors (null dereferences and divide by zero), recovery shepherding consists of returning a manufactured value, as for failure-oblivious computing. However, the key idea of recovery shepherding is to track the manufactured value to see (1) whether they are passed to system calls or files and (2) whether they disappear. In the former case, system calls and file writes are disabled if they involve a fake manufactured value to limit error propagation. When a manufactured value is no longer used and referenced, it means that the error has somehow evaporated, and the experiments of the article show that this is often the case.

### 4.8 Collaborative Repair

A cross-cutting concern of repair at runtime is to share the repairs that work across all instances of the same application. This has been explored under the name of "application community." Locasto et al. [105] uses application communities to find and distribute repairs of the form of stack manipulation. Rinard et al. [147] also reports on experiments on the centralization of monitoring information and the distribution of repairs across a community of applications.

## 5 EMPIRICAL KNOWLEDGE ON REPAIR

Beyond proposing new repair techniques, there is a thread of research on empirically investigating the foundations, impact, and applicability of automatic repair, whether behavioral or state repair.

There is wealth of information in software repositories that can be used for repair. In particular, one can mine bug reports and commits for knowledge that is valuable for automatic repair. Martinez and Monperrus [115] studied 89,993 commits to mine repair actions from manually written patches. By repair actions, they mean kinds of changes on the abstract syntax trees of programs such as modifying an if condition. They later investigated [116] the *redundancy assumption* in automatic repair (whether you can fix bugs by rearranging existing code) and found that it holds in practice: Many bug fix commits only rearrange existing code, a result confirmed by Barr et al. [12]. Zhong and Su [196] conducted a case study on over 9,000 real-world patches and found important facts for automatic repair: For instance, their analysis outlines that some bugs are repaired with changing the configuration files.

On the goodness of synthesized patches, Fry et al. [48] conducted a study of machine-generated patches based on 150 participants and 32 real-world defects. Their work shows that machine-generated patches are slightly less maintainable than human-written ones. Tao et al. [176] performed a similar study to study whether machine-generated patches assist human debugging. Monperrus [125] further discussed the *patch acceptability* criteria of synthesized patches and emphasized that assessing patch acceptability may require a high level of expertise, a result confirmed by Reference [114]. Qi et al. [141] are the first to thoroughly analyze the patches generated by Genprog and found that most of them are incorrect. It is an open question whether this holds for test-suite-based repair in general [114]. When they are incorrect, it is because they exploit specificities and weaknesses of the test suite, which can be seen as a kind of *overfitting*. A repair technique is said to overfit when the synthesized patch only works on the failing inputs and fails to generalize. Smith et al. [165] also studied the problem of overfitting in automatic repair; on a dataset of student programs, they show that Genprog and related techniques do suffer from overfitting.

A study by Kong et al. [83] compares different repair systems: GenProg [92], RSRepair [140], and AE [188]. They report repair results on 119 seeded bugs and 34 real bugs from the Siemens benchmark and show that not all techniques are equal.

Finally, for the knowledge on repair to consolidate, there is a need for accepted, well-defined, and publicly available benchmarks [125]. Le Goues et al. [94] have set up such a benchmark for bugs in C programs; it totals 1,183 bugs, collected in open-source projects and student code.

## 6 RELATED TECHNIQUES

We now present works that are related to automatic repair, yet are not "automatic repair" per se, according to the definitions we gave in Section 3 and 4. In particular, they either miss the full automation or the actual repair of real programs.

### 6.1 Forward Engineering for Repair

Many authors have tried to list the important principles to have robust and resilient if not self-repairable applications. These principles can be implemented and enforced as first-class concepts in frameworks and libraries. This is what can be called "forward engineering for repair."

Somayaji et al. describe principles to build immune computer systems [166]: distributability, multi-layering, diversity, disposability, autonomy, adaptability, behavioral sense-of-self, and anomaly detection. Candea and Fox [19] define a set of characteristics for programs to recover quickly: with those characteristics an application becomes "*crash-only software*." The two key characteristics are that all interactions between components have a timeout and all resources are leased. Sussmann [173] as well as Gabriel and Goldmann [49] also provide insightful perspectives on how to build resilient and self-repairable software.

There are also frameworks for supporting repair. Flora [168] is a framework to support local restart in applications. It is principally composed of a communication manager for dropping or queuing messages between components. Denaro et al. [37] proposes an architecture to fix interoperability bugs in service-oriented systems. Adaptors between service variants are manually written and are selected at runtime to enable correct communication. Levinson [97] defines an embedded Domain-specific language (DSL) to support runtime searches in a space of program variations. Zhou et al. [197] defines annotations for operating system C code to recover from driver errors in Linux. The annotations are checked by a type system and drives invariant restoration. Demsky and Dash [35] proposes Bristlecone, a language with built-in robustness capabilities. Bristlecone is based on tasks and dependencies between tasks, as well as transactional state changes. Error-handling is thus fully automated.

A known characteristic of bugs is that the same kind of bug can affect many different locations in the same code base. In this case, it is desirable to write a unique patch that is then applied to all those locations. The generic patch can be inferred from a concrete instance at a given location or written in an abstract way. This has been called "systematic editing" by Meng et al. [121]. Similarly, Sun et al. [171, 172] propose tool support for patch applications. The Coccinelle tool [134] also provides this functionality. The abstract patches can be automatically inferred from concrete instances [4, 122].

### 6.2 Repair Suggestions

There are some systems that give "repair suggestions" to the developer. While it is not fully automated, if the suggestion is correct, then such a system can be seen as providing partial automatic repair, where the repair system and the developer work in tandem.

Hartmann et al. [61] designed a system called "HelpMeout" that proposes suggestions to fix error messages. The system targets compiler error messages and runtime exceptions. It first collects error messages and the associated changes that occur on developer's machines that are monitored. Then, when the same error message is encountered by another developer, the system compares the erroneous source file with the closest fixed version that is in the database. It uses a tailored distance metric to increase the relevance of suggestions.

Jeffrey et al. [67] presented a fix suggestion approach based on association rules. The rules suggest a bug fix action for suspicious statements represented by a number of features (in the machine-learning meaning). The features (called "descriptors" in the article) are abstraction over the tokens of the statements. The prediction also uses "interesting value mapping pairs," which are concrete values that enable test cases to pass (a.k.a. value replacement [68] and angelic values [25, 34]). The bug fix recommendations are typical comparison operator change, constant change, add, or increase numerical values.

Kaleeswaran et al. [74] have proposed a repair suggestion approach based on correlations variable values and expected output. The expected output is obtained through concolic executions, and the repair hints consist of changing the RHS of a single assignment statement.

Abraham and Erwig [1] suggest change in Excel formulas. Malik et al. [113] transform runtime data structure repair (see 4.7.1) as fix suggestions. Brodie et al. [15] design a distance metric across call stacks (stack trace) to match issue reports and known fixes.

### 6.3 Theoretical Software Repair

Some authors explore automatic repair with strong assumptions under which there exists no program in practice. To the best of our knowledge, there is no survey article on this area, but the article by Bodik and Jobstmann contains a dedicated section about this [14]. Here, the most notable articles in this area are briefly mentioned for giving the reader a first set of pointers. For instance, Jobstmann et al. [72] repair programs that are expressed in linear temporal logics. George [55] describes a simple and theoretical programming model that supports automatic recovery via a kind of homeostasis that maintains invariants. Fisher et al. [138] also perform repair on a toy formal language. Wang and Cheng [181] state program repair as edit sequences on state machines. Zhang and Ding [195] repair computation tree logic models.

## 7 CONCLUSION

This article has presented an annotated bibliography on automatic software repair. This research field is both old and new. It is old because we can find techniques related to automatic repair in fault-tolerance articles from the 1970s and 1980s, for instance, a 1973 article is entitled "STAREX Self-Repair Routines: Software Recovery in the JPL-STAR Computer" [149]. It is new, because the idea of automatically changing the code, i.e., behavioral repair, has started to be explored only since the end of 2000. Whether old or new, the techniques have to scale to today's size and complexity or software stacks, and we are not there yet. This means that this is only the beginning, and in the upcoming years, we are going to have much fun, surprise, and admiration in the field of automatic software repair.

## REFERENCES

[1] Robin Abraham and Martin Erwig. 2005. Goal-directed debugging of spreadsheets. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. 37–44.

[2] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building useful program analysis tools using an extensible java compiler. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*. 14–23.

[3] Paul E. Ammann and John C. Knight. 1988. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput.* 37, 4 (1988), 418–425.

[4] Jesper Andersen and Julia L. Lawall. 2010. Generic patch inference. *Automat. Softw. Eng.* 17, 2 (2010), 119–148.

[5] Andrea Arcuri. 2009. *Automatic Software Generation and Improvement Through Search Based Techniques*. Ph.D. Dissertation. The University of Birmingham.

[6] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Appl. Soft Comput.* 11, 4 (2011), 3494–3514.

[7] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation*. 162–168.

[8] Algirdas Avizienis. 1985. The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.* 11, 12 (1985), 1491–1501.

[9] Algirdas Avizienis, J.-C. Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Depend. Sec. Comput.* 1, 1 (2004), 11–33.

[10] Tanzirul Azim, Iulian Neamtiu, and Lisa Marvel. 2014. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 623–628.

[11] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Trans. Soft. Eng.* 41, 5 (2015), 507–525.

[12] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–317.

[13] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Not.* 41, 6 (2006), 158–168.

[14] Rastislav Bodik and Barbara Jobstmann. 2013. Algorithmic program synthesis: Introduction. *Int. J. Softw. Tools Technol. Transf.* 15, 5 (2013), 397–411. DOI: http://dx.doi.org/10.1007/s10009-013-0287-9

[15] M. Brodie, S. Ma, G. Lohman, L. Mignet, M. Wilding, J. Champlin, and P. Sohn. 2005. Quickly finding known software problems via automated symptom matching. In *Proceedings of the International Conference on Autonomic Computing*. 101–110.

[16] Yuriy Brun, Earl Barr, Ming Xiao, Claire Le Goues, and Prem Devanbu. 2013. *Evolution Vs. Intelligent Design in Program Patching*. Technical Report. UC Davis.

[17] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1109–1120.

[18] G. Candea and A. Fox. 2001. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*. 125–130.

[19] G. Candea and A. Fox. 2003. Crash-only software. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems*. 12–12.

[20] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2004. Microreboot: A technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*. 3–3.

[21] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. 2003. JAGR: An autonomous self-recovering application server. In *Proceedings of the Workshop on Active Middleware Services*. 168–177.

[22] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and escaping infinite loops with jolt. In *Procceedings of the European Conference on Object-Oriented Programming (ECOOP'11)*. 609–633.

[23] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. 2013. Automatic recovery from runtime failures. In *Proceedings of the International Conference on Software Engineering*.

[24] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2010. Automatic workarounds for web applications. In *Proceedings of the Foundations of Software Engineering Conference*. 237–246.

[25] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *Proceeding of the International Conference on Software Engineering*. 121–130.

[26] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, João Pedro Sousa, Bridget Spitnagel, and Peter Steenkiste. 2002. Using architectural style as a basis for system self-repair. In *Proceedings of the IFIP 17th World Computer Congress/3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*. 45–59.

[27] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *Proceedings of the International Conference on Software Engineering*. 792–801.

[28] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2015. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Inf. Softw. Technol.* 57 (Jan. 2015), 66–76. DOI: http://dx.doi.org/10.1016/j.infsof.2014.08.004

[29] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2005. Vigilante: End-to-end containment of internet worms. In *ACM SIGOPS Operating Systems Review*, Vol. 39. 133–147.

[30] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. 2010. Towards automated inconsistency handling in design models. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering*. 348–362.

[31] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating fixes from object behavior anomalies. In *Proceedings of the International Conference on Automated Software Engineering*. 5.

[32] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting repairs for broken unit tests. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. 433–444.

[33] V. Debroy and W. E. Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 65–74.

[34] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA'14)*. Hyderabad, India. DOI: http://dx.doi.org/10.1145/2593735.2593740

[35] Brian Demsky and Alokika Dash. 2008. Bristlecone: A language for robust software systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'08)*. 490–515.

[36] B. Demsky and M. Rinard. 2003. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Not.* 38, 11 (2003), 78–95.

[37] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. 2009. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 253–262.

[38] Louise A. Dennis, Raul Monroy, and Pablo Nogueira. 2006. Proof-directed debugging and repair. In *Proceedings of the 7th Symposium on Trends in Functional Programming*. 131–140.

[39] Aritra Dhar, Rahul Purandare, Mohan Dhawan, and Suresh Rangaswamy. 2015. CLOTHO: Saving programs from malformed strings and incorrect string-handling. In *Foundations of Software Engineering*. ACM, 555–566.

[40] Kinga Dobolyi and Westley Weimer. 2008. Changing java's semantics for handling null pointer exceptions. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*. 47–56.

[41] J. A. Duraes and H. S. Madeira. 2006. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Softw. Eng.* 32, 11 (2006), 849–867.

[42] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401.

[43] Marc Eisenstadt. 1997. My hairiest bug war stories. *Commun. ACM* 40, 4 (1997), 30–37.

[44] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. 2007. Assertion-based repair of complex data structures. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 64–73.

[45] H. ETO and K. Yoda. 2001. Propolice: Improved stacksmashing attack detection. *IPSJ SIG Not2* 75 (2001), 181–188.

[46] S. Forrest, T. V. Nguyen, W. Weimer, and C. Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. 947–954.

[47] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. 2010. Exception handling for repair in service-based processes. *IEEE Trans. Softw. Eng.* 36, 2 (2010), 198–215.

[48] Zachary P. Fry, Bryan Landau, and Westley Weimer. 2012. A human study of patch maintainability. In *Proceedings of the International Symposium on Software Testing and Analysis*. 177–187.

[49] Richard P. Gabriel and Ron Goldman. 2006. Conscientious software. In *ACM SIGPLAN Notices*, Vol. 41. 433–450.

[50] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for C programs. In *Proceedings of the 37th International Conference on Software Engineering*. 459–470.

[51] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing Q&A sites. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[52] Z. Gao, Z. Chen, Y. Zou, and A. Memon. 2015. SITAR: GUI test script repair. *IEEE Trans. Softw. Eng.* 42, 2 (2015), 170–186.

[53] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. 2003. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*. 61–89.

[54] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems*. 24–33.

[55] Selvin George, David Evans, and Steven Marchette. 2013. A biological programming model for self-healing. In *Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems (SSRS)*. Fairfax, VA, 72–81.

[56] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. 2007. Self-healing systems survey and synthesis. *Decis. Support Syst.* 42, 4 (2007), 2164–2185.

[57] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. 2014. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*. 243–253.

[58] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based program repair using SAT. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[59] Andreas Griesmayer, Roderick Bloem, and Byron Cook. 2006. Repair of boolean programs with an application to C. In *Computer Aided Verification*. 358–371.

[60] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[61] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1019–1028.

[62] Haifeng He and Neelam Gupta. 2004. Automated debugging using path-based weakest preconditions. In *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE'04)*. 267–280.

[63] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the International Conference on Software Engineering*. 612–621.

[64] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM SIGPLAN Not.* 39, 12 (2004).

[65] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. 1995. Software rejuvenation: Analysis, module and applications. In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 381–390.

[66] Daniel Jackson and Mandana Vaziri. 2000. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. 14–25.

[67] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. 2009. BugFix: A learning-based tool to assist developers in fixing bugs. In *Proceedings of the International Conference on Program Comprehension (ICPC'09)*. 70–79.

[68] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the International Symposium on Software Testing and Analysis*. 167–178.

[69] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the International Conference on Software Engineering*, Vol. 1. 215–224.

[70] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. 2016. A metamorphic testing approach for supporting program repair without the need for a test oracle. *J. Syst. Softw.* (2016).

[71] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* 46, 6 (2011), 389–400.

[72] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *Computer Aided Verification*. 226–238.

[73] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. 2008. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 295–308.

[74] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. Minthint: Automated synthesis of repair hints. In *Proceedings of the International Conference on Software Engineering*. 266–276.

[75] Sara Kalvala and Richard Warburton. 2011. A formal approach to fixing bugs. In *Formal Methods, Foundations and Applications*. 172–187.

[76] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and Bernardo Cuenca-Grau. 2006. Repairing unsatisfiable concepts in OWL ontologies. In *The Semantic Web: Research and Applications*. Vol. 4011. 170–184.

[77] M. Kasbekar, C. Narayanan, and C. R. Das. 1999. Selective checkpointing and rollbacks in multi-threaded object-oriented environment. *IEEE Trans. Reliabil.* 48, 4 (1999), 325–337.

[78] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search. In *Proceedings of the International Conference on Automated Software Engineering*.

[79] Christian Kern and Javier Esparza. 2010. Automatic error correction of java programs. In *Formal Methods for Industrial Critical Systems*. 67–81.

[80] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'13)*.

[81] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive program repair. In *International Conference on Computer Aided Verification.* Springer, 217–233.

[82] Donald E. Knuth. 1989. The errors of TEX. *Softw. Pract. Exper.* 19, 7 (1989), 607–685.

[83] Xianglong Kong, Lingming Zhang, W. Eric Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair? In *Proceedings of the International Symposium on Software Reliability Engineering.* IEEE, 194–204.

[84] Robert Könighofer and Roderick Bloem. 2011. Automated error localization and correction for imperative programs. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD'11), 2011.* 91–100.

[85] Robert Könighofer and Roderick Bloem. 2013. Repair with on-the-fly program analysis. In *Hardware and Software: Verification and Testing.* 56–71.

[86] R. Koo and S. Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.* 1 (1987), 23–31.

[87] Matt Lake. 2010. Epic Failures: 11 Infamous Software Bugs. Retrieved from http://www.computerworld.com/article/2515483/enterprise-applications/epic-failures--11-infamous-software-bugs.html.

[88] Sebastian Lamelas-Marcote and Martin Monperrus. 2015. *Automatic Repair of Infinite Loops.* Technical Report hal-01144026. University of Lille. https://arxiv.org/pdf/1504.05078.pdf.

[89] Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A declarative approach to finding API protocols and bugs in linux code. In *International Conference on Dependable Systems & Networks.* 43–52.

[90] N. Lazaar, A. Gotlieb, and Y. Lebbah. 2011. A framework for the automatic correction of constraint programs. In *Proceedings of the International Conference on Software Testing, Verification and Validation.* 319–326.

[91] X. B. D. Le, D. Lo, and C. L. Goues. 2016. History driven program repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16).* 213–224.

[92] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 2012. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *Proceedings of the International Conference on Software Engineering.* 3–13.

[93] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Softw. Qual. J.* 21, 3 (2013), 421–443.

[94] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. (unpublished).

[95] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38 (2012), 54–72.

[96] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. 2013. Repairing selenium test cases: An industrial case study about web page element localization. In *International Conference on Software Testing, Verification and Validation.* IEEE, 487–488.

[97] R. Levinson. 2005. Unified planning and execution for autonomous software repair. In *Workshop on Plan Execution: A Reality Check.*

[98] Chris Lewis and Jim Whitehead. 2010. Runtime repair of software faults using event-driven monitoring. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10),* vol. 2. 275.

[99] Z. Liang and R. Sekar. 2005. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security.* 213–222.

[100] Yiyan Lin and Sandeep Kulkarni. 2014. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* ACM, 237–247.

[101] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. 2007. AutoPaG: Towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security.* 329–340.

[102] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'13).* 282–291.

[103] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 715–726.

[104] Peng Liu and Charles Zhang. 2012. Axis: Automatically fixing atomicity violations through solving control constraints. In *Proceedings of the 2012 International Conference on Software Engineering.* 299–309.

[105] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. 2006. Software self-healing using collaborative application communities. In *Proceedings of the Network and Distributed System Security Symposium.*

[106] Francesco Logozzo and Tom Ball. 2012. Modular and verified automatic program repair. In *Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications.*

[107] Francesco Logozzo and Matthieu Martel. 2013. Automatic repair of overflowing expressions with abstract interpretation. In *Semantics, Abstract Interpretation, and Reasoning About Programs: Essays Dedicated to David A. Schmidt on the Occasion of His Sixtieth Birthday*, Vol. 129. 341–357.

[108] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. 2012. Automatic input rectification. In *Proceedings of Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'12).*

[109] Fan Long and Martin C. Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'15).*

[110] Fan Long and Martin C. Rinard. 2016. Prophet: Automatic patch generation via learning from successful patches. In *Proceedings of the Symposium on Principles of Programming Languages.*

[111] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound input filter generation for integer overflow errors. *ACM SIGPLAN Not.* 49, 1 (2014), 439–452.

[112] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[113] M. Z. Malik, J. H. Siddiqi, and S. Khurshid. 2011. Constraint-based program debugging using data structure repair. In *International Conference on Software Testing, Verification and Validation (ICST'11).* 190–199.

[114] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.* PADERBORN, Germany. DOI:http://dx.doi.org/10.1007/s10664-016-9470-4

[115] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empir. Softw. Eng.* 20, 1 (2015), 176–205. DOI:http://dx.doi.org/10.1007/s10664-013-9282-8

[116] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'14).* Hyderabad, India. DOI:http://dx.doi.org/10.1145/2591062.2591114

[117] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering.*

[118] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering.* 691–701.

[119] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd International Conference on World Wide Web.* ACM, 63–74.

[120] Atif M. Memon. 2008. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.* 18, 2 (2008), 4.

[121] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* 329–342.

[122] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the International Conference on Software Engineering.* 502–511.

[123] Michael G. Merideth. 2003. Enhancing survivability with proactive fault-containment. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks.*

[124] B. Meyer. 1992. Applying "design by contract." *Computer* 25, 10 (1992), 40–51.

[125] Martin Monperrus. 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 *Procedings of the International Conference on Software Engineering.* 234–242. DOI:http://dx.doi.org/10.1145/2568225.2568324

[126] Martin Monperrus. 2017. Principles of antifragile software. In *Proceedings of the Salon des Refusés 2017.* Brussels, Belgium. DOI:http://dx.doi.org/10.1145/3079368.3079412

[127] P. Muntean, V. K. Kommanapalli, A. Ibing, and C. Eckert. 2015. Automated generation of buffer overflows quick fixes using symbolic execution and SMT. In *International Conference on Computer Safety, Reliability & Security (SAFECOMP'15).*

[128] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. 2003. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering.* 455–464.

[129] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering.*

[130] Huu Hai Nguyen and Martin Rinard. 2007. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management.* 15–30.

[131] Mihai Nica, Simona Nica, and Franz Wotawa. 2013. On the use of mutations and testing for debugging. *Softw. Pract. Exper.* 43, 9 (2013), 1121–1142.

[132] G. Novark, E. D. Berger, and B. G. Zorn. 2007. Exterminator: Automatically correcting memory errors with high probability. *ACM SIGPLAN Not.* 42, 6 (2007), 1–11.

[133] Vinicius Oliveira, Eduardo Souza, Claire Le Goues, and Celso G. Camilo. 2016. Improved crossover operators for genetic programming for program repair. In *Proceedings of the 8th International Symposium on Search Based Software Engineering.*

[134] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. 2008. Documenting and automating collateral evolutions in linux device drivers. *ACM SIGOPS Operat. Syst. Rev.* 42, 4 (2008), 247–260.

[135] Jeff H. Perkins, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, Martin Rinard, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, and Stelios Sidiroglou. 2009. Automatically patching errors in deployed software. *Proceedings of the Symposium on Operating Systems Principles* (2009), 87.

[136] Barbara Pernici and Anna Maria Rosati. 2007. Automatic learning of repair strategies for web services. In *Proceedings of the 5th European Conference on Web Services.* 119–128.

[137] Mauro Pezzè, Martin C. Rinard, Westley Weimer, and Andreas Zeller. 2011. *Self-Repairing Programs, Report From Dagstuhl Seminar.* Technical Report 11062. Schloss Dagstuhl—Leibniz-Zentrum für Informatik.

[138] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2009. Program repair as sound optimization of broken programs. In *International Symposium on Theoretical Aspects of Software Engineering.*

[139] Y. Qi, X. Mao, and Y. Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM'13).*

[140] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering.* 254–265.

[141] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'15).*

[142] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. 2005. Rx: Treating bugs as allergies—A safe method to survive software failures. *ACM SIGOPS Operat. Syst. Rev.* 39, 5 (2005), 235–248.

[143] Brian Randell. 1975. System structure for software fault tolerance. *IEEE Trans. Softw. Eng.* 2 (1975), 220–232.

[144] Martin Rinard. 2003. Acceptability-oriented computing. *ACM SIGPLAN Not.* 38, 12 (2003), 57–75.

[145] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee Jr. 2004. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems, Design & Implementation.* 21–21.

[146] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. 2004. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference.* 82–90.

[147] M. Rinard, M. Ernst, and J. Perkins. 2011. *Collaborative Learning for Security and Repair in Application Communities.* Technical Report. Massachusetts Institute of Technology.

[148] Martin C. Rinard. 2006. *Survival Techniques for Computer Programs.* Technical Report. MIT.

[149] J. A. Rohr. 1973. STAREX self-repair routines: Software recovery in the JPL-STAR computer. In *Proceedings of the International Conference on Fault-Tolerant Computing (FTCS'73).*

[150] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. 2014. Cost-aware automatic program repair. In *International Static Analysis Symposium.* Springer, 268–284.

[151] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'12).* 277–287.

[152] E. Schulte, S. Forrest, and W. Weimer. 2010. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering.* 313–316.

[153] A. S. Sethi and others. 2004. A survey of fault localization techniques in computer networks. *Sci. Comput. Program.* 53, 2 (2004), 165–194.

[154] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically fixing C buffer overflows using program transformations. In *International Conference on Dependable Systems and Networks.* 124–135.

[155] Mary Shaw. 2002. Self-healing: Softening precision to avoid brittleness. In *Proceedings of the 1st Workshop on Self-healing Systems.* 111–114.

[156] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. 2008. Using components for architecture-based management: The self-repair case. In *Proceedings of the 30th International Conference on Software Engineering*. 101–110.

[157] S. Sidiroglou and A. D. Keromytis. 2005. Countering network worms through automatic patch generation. *Secur. Priv.* 3, 6 (2005), 41–49.

[158] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. 2009. Assure: Automatic software self-healing using rescue points. *ACM Sigplan Not.* 44, 3 (2009), 37–48.

[159] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. 2005. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 161.

[160] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 43–54.

[161] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 124–134.

[162] Josep Silva. 2011. A survey on algorithmic debugging strategies. *Adv. Eng. Softw.* 42, 11 (2011), 976–991.

[163] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Not.* 48, 15–26.

[164] A. Smirnov and T. Chiueh. 2005. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium.*

[165] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15).*

[166] A. Somayaji, S. Hofmeyr, and S. Forrest. 1998. Principles of a computer immune system. In *Proceedings of the 1997 Workshop on New Security Paradigms*. 75–82.

[167] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Fix me up: Repairing access-control bugs in web applications. In *Proceedings of the Network and Distributed System Security Symposium.*

[168] H. Sözer, B. Tekinerdoğan, and M. Akşit. 2009. FLORA: A framework for decomposing software architecture to introduce local recovery. *Softw. Prac. Exper.* 39, 10 (2009), 869–889.

[169] Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. 2011. Programs, tests, and oracles: The foundations of testing revisited. In *Proceedings of the International Conference on Software Engineering*. 391–400.

[170] M. Stumptner and F. Wotawa. 1996. A model-based approach to software debugging. In *Proceedings on the 7th International Workshop on Principles of Diagnosis.*

[171] B. Sun, R. Y. Chang, X. Chen, and A. Podgurski. 2008. Automated support for propagating bug fixes. In *Proceedings of the International Symposium on Software Reliability Engineering*. 187–196.

[172] B. Sun, G. Shu, A. Podgurski, S. Li, S. Zhang, and J. Yang. 2010. Propagating bug fixes with fast subgraph matching. In *Proceedings of the International Symposium on Software Reliability Engineering*. 21–30.

[173] Gerald Jay Sussman. 2007. Building Robust Systems An Essay. (2007). https://groups.csail.mit.edu/mac/users/gjs/6.945/readings/robust-systems.pdf.

[174] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. 2008. Avoiding program failures through safe execution perturbations. In *International Conference on Computer Software and Applications.*

[175] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated repair of software regressions. In *Proceedings of Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'15).*

[176] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: A human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 64–74.

[177] S. Thomas and L. Williams. 2007. Using automated fix generation to secure SQL statements. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*. 9.

[178] Wilfredo Torres-Pomales and others. 2000. *Software Fault Tolerance: A Tutorial.* Technical Report NASA-2000-tm210616. NASA.

[179] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. 2000. Architectural repair of open source software. In *Proceedings of the International Workshop on Program Comprehension*. 48–59.

[180] K. Tsipenyuk, B. Chess, and Gary McGraw. 2005. Seven pernicious kingdoms: A taxonomy of software security errors. *Secur. Priv.* 3, 6 (2005), 81–84.

[181] Farn Wang and Chih-Hong Cheng. 2008. Program repair suggestions from graphical state-transition specifications. In *Proceedings of the 28th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'08).*

[182] Tielei Wang, Chengyu Song, and Wenke Lee. 2014. Diagnosis and emergency patch generation for integer overflow exploits. In *Detection of Intrusions and Malware, and Vulnerability Assessment.* 255–275.

[183] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 16.

[184] Y. M. Wang, Y. Huang, and C. Kintala. 1997. Progressive retry for software failure recovery in message-passing applications. *IEEE Trans. Comput.* 46, 10 (1997), 1137–1141.

[185] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the International Symposium on Software Testing and Analysis.* 12.

[186] Westley Weimer. 2006. Patches as better bug reports. In *Proceedings of the International Conference on Generative Programming and Component Engineering.*

[187] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Commun. ACM* 53, 5 (2010), 109.

[188] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering.* 356–366.

[189] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering.*

[190] W. E. Wong and V. Debroy. 2009. A survey on software fault localization. University of Texas at Dallas, Tech. Rep. UTDCS-45-09 (2009).

[191] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. 2009. Supporting automatic model inconsistency fixing. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ACM, 315–324.

[192] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. 2015. Range fixes: Interactive error resolution for software configuration. *IEEE Trans. Softw. Eng.* 41, 6 (2015), 603–619.

[193] Andreas Zeller, Yi Wei, Bertrand Meyer, Martin Nordio, Carlo A. Furia, and Yu Pei. 2014. Automated fixing of programs with contracts. *IEEE Trans. Softw. Eng.* 40, 5 (2014), 427–449.

[194] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering.* 272–281.

[195] Y. Zhang and Y. Ding. 2008. CTL model update for system modifications. *J. Artif. Intell. Res.* 31, 1 (2008), 113–155.

[196] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering.*

[197] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. 2006. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation.* 45–60.

Received December 2015; revised March 2017; accepted June 2017