The Journal of Systems and Software 209 (2024) 111918

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Extending the range of bugs that automated program repair can handle ☆, ☆☆

Omar I. Al-Bataineh [a], Leon Moonen [a,b,*], Linas Vidziunas [a]

[a] Simula Research Laboratory, Oslo, Norway
[b] BI Norwegian Business School, Oslo, Norway

ARTICLE INFO

Dataset link: https://github.com/secureIT-project/extendingAPR, https://doi.org/10.5281/zenodo.10397656

Keywords:
Automated program repair
Bug classification
Non-observable and liveness bugs
Hybrid techniques

ABSTRACT

Modern *automated program repair* (APR) is well-tuned to finding and repairing bugs that introduce *observable* erroneous behavior to a program. However, a significant class of bugs does not lead to observable behavior (e.g., termination bugs and non-functional bugs). Such bugs can generally not be handled with current APR approaches, so complementary techniques are needed. To stimulate the systematic study of alternative approaches and hybrid combinations, we devise a novel bug classification system that enables methodical analysis of their bug detection power and bug repair capabilities. To demonstrate the benefits, we study the repair of termination bugs in sequential and concurrent programs. Our analysis shows that integrating dynamic APR with formal analysis techniques, such as termination provers and software model checkers, reduces complexity and improves the overall reliability of these repairs. We empirically investigate how well the hybrid approach can repair termination and performance bugs by experimenting with hybrids that integrate different APR approaches with termination provers and execution time monitors. Our findings indicate that hybrid repair holds promise for handling termination and performance bugs. However, the capability of the chosen tools and the completeness of the available correctness specification affects the quality of the patches that can be produced.

1. Introduction

Corrective maintenance, i.e., finding and repairing software bugs, is one of the main categories of software maintenance, and responsible for a large part of the overall costs of software development (Swanson, 1976). *Automated program repair* (APR) promises to increase developer productivity and drastically reduce the costs of corrective maintenance (Le Goues et al., 2019; Monperrus, 2018). Despite the advances of APR for real-world programs (Marginean et al., 2019), these approaches can only handle certain types of bugs because they generally rely on dynamic analysis for functional verification, where a test suite is used to simulate the input and monitor the output to check correct behavior. However, this is only viable if the effects of a bug can be observed when executing the program.

Detecting and repairing *non-observable bugs* and *liveness bugs* (i.e., bugs that do not lead to incorrect results or crashes) pose a far greater challenge. For example, identifying a liveness bug requires finding an infinite execution that will never satisfy the desired liveness property (Alpern and Schneider, 1987). It is not known how long one would need to run the program to reveal an existing liveness bug, making it impractical to find such cases using dynamic analysis. Another critical challenge that makes detection and repair of liveness bugs notoriously hard is that the effects that a liveness bug is triggered are generally unobservable (i.e., they typically produce little debugging information). One option for finding this class of bugs is applying formal program analysis techniques that use correctness specifications to detect liveness bugs. Such a rigorous analysis can both help to detect the presence of liveness bugs as well as assure the absence of these bugs in automatically generated patches.

The question which (combinations of) techniques will be most effective at handling certain bugs is an open research question that forms the foundation of this paper. To stimulate the systematic study of alternative APR approaches and hybrid APR combinations, we devise a novel bug classification system that enables methodical analysis of their bug detection power and bug repair capabilities. In earlier work, various bug classification schemes were developed to understand when and why specific bugs arise, and how they are fixed. These classifications use a number of criteria, such as cause-impact (Li et al., 2006; Tan et al., 2014), severity-priority (Serrano and Ciordia, 2005),

☆ This work has been financially supported by the Research Council of Norway through the secureIT project (RCN contract #288787).
☆☆ Editor: W. Eric Wong.
* Corresponding author at: Simula Research Laboratory, Oslo, Norway.
E-mail addresses: omar@simula.no (O.I. Al-Bataineh), leon.moonen@computer.org (L. Moonen), linasvidz@simula.no (L. Vidziunas).

https://doi.org/10.1016/j.jss.2023.111918
Received 2 March 2023; Received in revised form 12 October 2023; Accepted 4 December 2023
Available online 14 December 2023

and bug complexity (Cotroneo et al., 2016). However, since they were designed for different goals, they do not capture the properties required to determine if a bug is amenable to a particular technique. To that end, we introduce a bug classification system explicitly aimed at comparing different techniques and evaluating the feasibility of their integration.

Contributions: The paper makes the following key contributions:

1. We propose a *novel bug classification system* based on three fundamental properties: *bug observability*, *bug reproducibility*, and *bug tractability*. This classification provides the APR community with a tool to methodologically explore and compare alternative and hybrid APR approaches by (i) analyzing the detection power of different bug detection techniques, (ii) distinguishing APR approaches based on their bug repair capabilities, and (iii) providing a common terminology that helps identify gaps in current APR research.
2. We discuss four *APR approaches* that can handle different classes of bugs: *dynamic APR*, *static APR*, *dynamic–static APR*, and *formal APR*. Moreover, we identify the conditions under which each approach can be effectively applied.
3. To demonstrate the benefits of our method, we study *termination bugs* in sequential and concurrent programs, and sketch novel *hybrid APR algorithms* for repairing such bugs. The study shows that termination bugs in *sequential* programs can be effectively addressed using *dynamic–static APR*, by first generating plausible patches using test cases and then using termination provers (Giesl et al., 2014; Chen et al., 2015; Brockschmidt et al., 2016) to check their correctness. The non-deterministic nature of termination bugs in *concurrent* programs makes them challenging for dynamic analysis, and they are best addressed with *formal APR* that combines termination provers with software model checkers (Jhala and Majumdar, 2009; Godefroid, 1997; Holzmann, 1997; Havelund and Pressburger, 2000; Musuvathi et al., 2002; Thompson et al., 2010; Baranová et al., 2017).
4. This paper extends our earlier work (Al-Bataineh and Moonen, 2022) with an empirical investigation of how well the proposed hybrid approach can handle termination and performance bugs. To this end, we create hybrids of tools representing different APR approaches with termination provers and execution time monitors. We use a dataset for termination bugs in C code that was originally developed to evaluate the efficacy of termination provers (Shi et al., 2022), and extend it with two performance bugs: one simple synthetic example, while the other one is a real-world Apache flaw that has also been analyzed by other researchers (Song and Lu, 2017). As a representative of search-based APR tools, we choose GenProg (Le Goues et al., 2012), and as a representative of semantic-based repair tools, we choose FAngelix (Yi and Ismayilzada, 2022). We were unable to locate a representative of template-based repair tools that can handle C code. Our findings indicate that mutation-based repair tools have a better chance of fixing performance bugs than semantic-based repair tools. Mutation-based repair tools can easily restructure the program using the basic mutation operators like move, swap, delete, and insert. Since programs having performance bugs are semantically correct programs, we observe that these operators can successfully address performance bugs. The satisfaction of a composite property, which combines a termination property and a semantic property describing the loop's logic, is necessary for fixing termination bugs. The integration of APR with termination provers would help to fully fix the termination bug if the semantic property is available. To ensure termination without necessarily maintaining the program's logic, patches can be made by combining termination provers and test cases. Both the GenProg and FAngelix hybrids were unable to fix termination bugs successfully: they produce patches that only guarantee termination and do not maintain the logic of the loop being fixed. The main cause for these issues is the incompleteness of the correctness specification (i.e., the available test suite), which is a frequently observed drawback of dynamic APR.

Remark 1. Different terms have been used to characterize a situation in which a program performs unexpectedly such as bug, defect, error, and fault. These terms are generally similar and refer to instances in which the analyzed program deviates from its intended behavior. However, the terms bug and defect are frequently used in the literature of software engineering, whereas the terms fault and error are frequently used in the literature of formal methods (especially model checking). However, to be consistent with earlier research on automated program repair, in this paper we use the terms bug and defect.

Notations: We start by explaining some notations that we use throughout the paper. When referring to an observer who records the execution of a program and its results, we use the notations O, O_{exp}, and O_{obs}, where O_{exp} stands for the output that would be produced by a correct version of the program, and O_{obs} stands for the output that O actually sees when running a program. We assume that O can distinguish between correct and incorrect outcomes. The property φ_{beh} is a formal property (typically written in temporal logic) that denotes the correct behavior of the program being analyzed, and φ_{reach} is a property that is used to check whether the program being analyzed can reach any of its halting locations. The expression $(P, i) \models \varphi_{beh}$ expresses that program P under input i fulfills property φ_{beh} while expression $(P, i) \not\models \varphi_{beh}$ expresses that P under i violates φ_{beh}. The expression $P \vdash t$ denotes that program P successfully passes test t. Finally, we write $(p_i \parallel p_j)$ to indicate that processes p_i and p_j are run in parallel, and $time(P, i)$ to denote the length of time that program P takes under input i.

2. Bug classification schemes

There are many different ways to expose bugs in programs, including manual inspection, dynamic analysis (testing), static analysis, model checking, or a combination of these techniques. Effective bug classification schemes can help understand why bugs arise and how to fix them. Classification can also help identify the most appropriate analysis technique for handling each class of bugs. Next, we discuss three existing bug classification systems, analyze their limitations, and introduce a new classification system that addresses them:

1. *Cause-impact criteria* (Li et al., 2006; Tan et al., 2014): Bugs are classified based on their cause: algorithmic, concurrency, memory, generic programming, and unknown, as well as based on their impact: security, performance, failure, and unknown.
2. *Severity and priority criteria*: This classification is used in many bug tracking systems (Serrano and Ciordia, 2005). Severity indicates the impact of the bug on the program's functionality and can be categorized as critical, major, moderate, minor, etc. Priority indicates how soon the bug should be fixed and is categorized into levels such as low, medium, and high.
3. *Bug complexity criteria* (Cotroneo et al., 2016): These criteria distinguish four main categories: (i) easy to detect, easy to repair bugs, (ii) easy to detect, difficult to repair bugs, (iii) difficult to detect, easy to repair bugs, and (iv) difficult to detect, difficult to repair bugs.

These three existing bug classification systems were not designed for comparing the capabilities and limitations of different bug detection or program repair techniques. As a result, their criteria do not capture the specific properties needed to determine whether a bug b is amenable to a particular technique T. To address this gap, we propose a new bug classification system that is based on three key properties of bugs, namely *bug observability*, *bug reproducibility*, and *bug tractability*. In Sections 3 and 4, these properties are then used to analyze the power of different bug detection techniques and APR approaches.

Before proceeding further, let us first define a program bug. We base ourselves on a specification of *expected behavior*: the expected responses (output) of the program to a given input.

Definition 1 (*Program Bug*)**.** Let P be a program, I a set of inputs, and φ_{beh} be a specification of expected behavior of P. We say that P suffers from a bug iff there exists at least one input $i \in I$ that leads to an execution trace under which program P violates φ_{beh}, formalized as $(P, i) \nvDash \varphi_{beh}$.

Since a complete specification of a program's expected behavior is often not available, test cases are generally used to model the expected behavior of a program P. We assume there exists (i, o_{exp}), where o_{exp} is the expected output for input $i \in I$. When the observed output $o_{obs} = (P, i)$ does not match the expected output o_{exp}, we say that P contains a bug.

Definition 2 (*Observable Bug*)**.** Let P be a program containing bug b. We say that b is an *observable bug* iff there exists an execution of P where, in a finite number of execution steps, the erroneous behavior of b can be seen by an observer O.

Definition 3 (*Classifying Bugs by Observability*)**.** We classify bugs based on the notion of observability in three types:

1. *observable bugs* whose erroneous behavior is fully observable in finite execution steps (e.g., arithmetic bugs),
2. *partially observable bugs* whose erroneous behavior is only partially observable at runtime because the faulty trace is infinite, so not all output of the program can be observed (e.g., termination bugs),
3. *non-observable bugs* whose erroneous behavior is fully unobservable at runtime (e.g., non-functional bugs).

Note that observability is a relative notion that depends on the observation power of O. In the simplest case, the observer can witness the output produced by P and the corresponding execution time. If we increase the observation power of O (i.e., the amount of information O can gather about the program's execution), some non-observable bugs may become observable. For example, bugs that adversely affect the memory or energy consumption can easily go unnoticed during the execution of the program. Such bugs can be exposed by using monitoring at the virtual machine or operating system level (Hebbal et al., 2015; Dovgalyuk et al., 2017; Gregg, 2020, 2019). Alternatively, the program can be augmented with additional variables and checks that help to keep track of these non-functional aspects at runtime (Al-Bataineh et al., 2021b). However, increasing observation power also increases the chance of affecting the program's execution (Mytkowicz et al., 2008).

We distinguish five common types of observable erroneous behavior: $EB = \{crash, exception, incorrectResult, softHang, hardHang\}$. While most types in EB are easy to understand, we will define the notions of *soft* and *hard* hang bugs. Hang bugs are a particular type of bugs that concern (temporary or permanent) lack of progress in observable behavior (Wang et al., 2008; Dean et al., 2015). Hang bugs can have various causes, such as iteration errors or communication deadlocks. To define hang bugs, we use a temporal specification (Pnueli, 1977; Manna and Pnueli, 1992) that checks if any of the locations where the program might terminate can be reached.

Definition 4 (*Halting Statements*)**.** We refer to a statement s in a program P as a *halting statement* iff the expected behavior of P is that execution terminates after executing statement s.

Examples of halting statements include special termination statements such as `exit`, or simply the final statement in a program. Observe that programs whose expected behavior is to never terminate have no halting statements (e.g., a webserver).

Definition 5 (*Hang Bugs*)**.** Let P be a program with a set of inputs I and H be the set of halting statements of P. Let φ_{temp} be a temporal property that puts an upper bound on the execution time of P, and φ_{reach} be a temporal property that checks whether P reaches a halting statement. Let also $EB' = EB \setminus \{softHang, hardHang\}$. We distinguish:

1. **Soft hang bugs**, also known as **performance bugs**, occur when there exists an input $i \in I$ that makes P unresponsive for a finite amount of time before execution is resumed and a halting statement is reached:

$$S : (P, i) \nvDash \varphi_{temp} \wedge (P, i) \vDash \varphi_{reach} \wedge output(P, i) \notin EB'$$

2. **Hard hang bugs**, also known as **termination bugs**, occur when there exists an input $i \in I$ that makes P unresponsive for an unbounded amount of time, never resuming to normal execution or reaching a halting statement:

$$H : (P, i) \nvDash \varphi_{temp} \wedge (P, i) \nvDash \varphi_{reach} \wedge output(P, i) \notin EB'$$

Hang bugs are also referred to as *liveness violations* in model checking and formal program analysis literature (Lamport, 1977; Alpern and Schneider, 1985; Killian et al., 2007; Li and Regehr, 2010), and we will use these terms interchangeably in the remainder.

We now turn to discuss the property of bug reproducibility.

Definition 6 (*Bug Reproducibility*)**.** Let P be a program containing a bug b and t_b be a test case that exposes b. We say that b is a *reproducible* or *deterministic bug* iff every time program P is executed under test t_b, bug b is exposed and the same erroneous behavior is observed. On the other hand, we say that b is a *hard-to-reproduce* or *non-deterministic bug* iff bug b is exposed in rare circumstances when repeating the execution of P under test t_b (i.e., the result of program P depends not only on the code of P but also on the timing of the execution).

Reproducible bugs are easy to detect, provided that the bug is observable (see Definition 2). Not surprisingly, hard-to-reproduce bugs are also hard to detect. Arithmetic bugs are examples of *easy-to-reproduce* bugs, while concurrency bugs are examples of *hard-to-reproduce* bugs. The last property we study is *bug tractability*, which depends on the *depth* of the bug and the size of the faulty trace it produces.

Definition 7 (*Bug Tractability*)**.** Let P be a program containing bug b and L be the set of reachable locations of P and $\ell_b \in L$ be the buggy location to the bug b. We say that the trace of b is a *tractable trace* iff for each execution of P that is buggy to b, the number of execution steps that are required to reach ℓ_b is bounded and that ℓ_b is not part of a loop that can be executed infinitely often. On the other hand, we say that the trace of b is *intractable* iff ℓ_b is visited infinitely often during the execution of P (i.e., ℓ_b is part of an infinite loop).

The presence of loops plays a crucial role in determining the tractability of a bug. The size of faulty traces for non-loop programs is typically shorter than those in loop programs. Based on the size of the faulty trace, we can further distinguish the class of tractable bugs: (i) *shallow bugs* are tractable bugs with finite short faulty traces, (ii) *deep bugs* are tractable bugs with finite but long faulty traces. For example, a bug in a loop program P that does not occur until a vast number of iterations are executed can be viewed as an example of a deep bug.

Bug Classification System: Table 1 summarizes the three properties in our classification system with their distinguishing attributes and impact on bug detection. Our classification system associates each bug with a three-tuple of concrete attributes for *{observability, reproducibility, tractability}*. For example, an arithmetic bug has the properties: *{observable, easy-to-reproduce, shallow}*.

Table 1
A summary of the key properties for bug classification with their attributes and impact.

Bug property	Property attributes	Impact on bug detection
observability	{observable, non-observable, partially-observable}	Affects detection power
reproducibility	{easy-to-reproduce, hard-to-reproduce}	Affects efficiency and scalability
tractability	{shallow, deep, liveness}	Affects detection power and efficiency

3. Bug detection techniques

Various bug detection techniques can be used to expose bugs in programs. In this work, we are interested in studying three well-known bug detection techniques: dynamic analysis, static analysis, and model checking. While dynamic analysis detects bugs in programs by executing them, static analysis and model checking use different techniques in bug detection that perform bug checking statically, without running the program. We start by discussing the requirements needed to expose bugs in each technique, the advantages and disadvantages of the techniques, and the theoretical foundation and detection power of each (i.e., the classes of bugs that each technique can handle).

3.1. Dynamic analysis

Dynamic analysis is a technique to identify bugs and vulnerabilities in programs by exercising various runs through the program based on valid inputs. Dynamic analysis can be performed using a test suite, which can be developed manually or via test case generation, or through fuzzing, which systematically explores a large amount of automatically generated tests. Fuzzing is one of the most common methods used to find vulnerabilities in programs (Miller et al., 1990, 2006). Early fuzz testing was based on sending random inputs to a program to check if it could be made to crash. The techniques have evolved to systematically explore the input space using knowledge from the source code or input formats to discover bugs that are hidden deep in the code.

Dynamic analysis has several advantages over the other program analysis techniques: (i) the program behavior can be monitored, and bugs can be exposed while the program is running; (ii) it allows for analysis of programs for which we do not have access to the actual code; (iii) it can be conducted against any program; and last but not least, (iv) it can identify bugs that are hard to find using static analysis. We now discuss the conditions under which bugs may be discovered in dynamic analysis.

Definition 8 (*Bug Detection in Dynamic Analysis*)**.** Let P be a program containing bug b, and D be a dynamic program analysis checker (i.e., an automated testing tool such as a fuzzer). We say that b is detectable in D iff

1. P is given in an executable form,
2. bug b is *observable* in some executions of P,
3. there exists a test suite T containing at least one failing test t by which bug b can be exposed in P.

However, many bugs whose detection requires the analysis of infinite traces or the satisfaction of complex composite properties cannot be found using an approach solely relying on test cases. Examples of such classes of bugs include (i) bugs in non-executable programs, (ii) liveness bugs such as termination and starvation bugs, and (iii) non-observable bugs such as non-functional and information flow bugs (Sabelfeld and Myers, 2003; Smith, 2007).

Observation 1. *Dynamic analysis techniques can handle observable classes of bugs with finite execution traces, provided that a testing mechanism is implemented by which the bug can be exposed, and provided that the program is executable.*

To address these limitations, we need to pair dynamic analysis with complementary bug detection techniques to improve the detection power of the approach. The remainder of this section discusses static analysis and model checking, which instead of using the program itself as in dynamic analysis, analyzes abstractions of the program, to improve observability and tractability and enable the detection of deeper bugs (David et al., 2016).

3.2. Static program analysis

Static program analysis is an approach for analyzing a computer program without actually executing it. The most significant advantage of static analysis is the ability to quickly and automatically examine the complete code of the program to find flaws that might be missed by dynamic analysis. The literature on static program analysis for bug detection is rich and mature (D'Silva et al., 2008; Bessey et al., 2010; Sadowski et al., 2018). Many of these techniques build on automatically evaluated analysis rules and bug detection patterns that capture the general conditions under which specific bugs can occur, providing a systematic way for their detection.

To capture the notion of bug detectability in static analysis, one needs to ensure the availability of the source code of the buggy program at which the bug b occurs and the availability of some valid solid theory for the detection of bug b.

Definition 9 (*Bug Detection in Static Analysis*)**.** Let P be a program containing bug b and S be a static program analyzer that can be used to expose bugs of type b. We say that b is a bug detectable by analyzer S if all of the following conditions hold:

1. the source code of P is available, and
2. S contains sound and complete detection method for b, and
3. P is written in a language that is accepted by S.

The increasing complexity of (loop) programs and the large variety of vulnerabilities make it difficult for static code analyzers to detect and identify vulnerabilities in a precise manner. One of the most significant disadvantages of the static code analysis methodology is the presence of false-positive warnings: the tool may signal possible bugs where there are none. However, reducing the number of false positives in static analysis tools is still an open problem.

Observation 2. *Static checkers can handle observable, non-observable, reproducible, and non-reproducible bugs, provided that the checker is built based on some solid mathematical foundation, and provided that the source code of the program is available in a programming language accepted by the checker.*

Dynamic and static analysis techniques have different bug detection powers as they rely on distinct assumptions and use orthogonal detection methods. Overall, these techniques have complementary strengths and weaknesses that are worth combining to improve the reliability of APR systems.

3.3. Model checking

Model checking (Burch et al., 1992; Bérard et al., 2001; Clarke et al., 2018) is an automated formal method for checking whether a finite-state model of a system meets a given specification. The technique

Table 2
A summary of key differences and similarities of dynamic analysis, static analysis, and model checking.

Criteria	Dynamic analysis	Static analysis	Model checking
Bug detection mechanism	Test cases	Pattern-based specifications	Formal correctness specifications
Accuracy of the analysis	Accurate	Inaccurate (suffers from false positives)	Accurate relative to the accuracy of model
The need of code availability	Not needed	Needed to perform the analysis	Needed to construct the model
The need of bug observability	Needed to detect the bug	Not needed (performs code analysis)	Not needed (performs state analysis)
Code coverage of the program	Incomplete code coverage	Complete code coverage	Complete code coverage
The need of code executability	Needed to detect the bug	The program will not be executed	The program will not be executed
Language dependency	Language-independent	Language dependent	Language dependent
Automation of the analysis	Can be automated (fuzz testing)	Fully automated	A model needs to be manually written

has been used successfully to debug complex computer hardware, concurrent systems, and real-world safety-critical systems.

Model checking has several advantages. It can detect errors that would be very difficult to notice with other methods, such as in concurrent programs. The properties that can be verified are more expressive than with traditional testing, depending on the formalism used to express them. For example, properties that require something to happen infinitely often, or properties that require that some alternative is always available. In addition, because every possible behavior of the model is checked, the result is inevitable, provided that the model checking tool itself has no serious errors.

Model checking can be an expensive procedure in a repair process because of its exhaustive nature. Expressing both the model of the system and the properties formally requires great care and expertise. Moreover, one of the most significant problems with model checking in practice is the so-called "state explosion problem": When the number of state variables in the system increases, the size of the system state space grows exponentially. However, abstractions can be applied to bring the verification within feasible bounds of model checking technology.

Software model checking tools (Jhala and Majumdar, 2009; Godefroid, 1997; Holzmann, 1997; Havelund and Pressburger, 2000; Musuvathi et al., 2002; Thompson et al., 2010; Baranová et al., 2017) verify the correctness of software models in a rigorous and automated fashion. Most tools construct a (symbolic) reachability graph for the program-under-analysis (i.e., a graph that contains reachable run-time states of the program) *without* running the program. This graph is then used to check if a property of interest holds. They typically implement sophisticated data structures that enable clever search algorithms and optimizations.

The answer returned by a model checker is either a notion of a successful verification (i.e., the specification holds), or a counterexample — an execution path that violates a given property. However, if the program being verified has an infinite state space, certain types of abstractions are needed, or the analysis may simply not terminate.

Definition 10 (*Bug Detection in Model Checking*)**.** Let P be a program containing bug b and SMC be a software model checker that can be used to expose bugs of type b. We say that b is a detectable bug in the model checker SMC iff:

1. a formal property φ_b is available that is written in the input specification language of SMC, which captures the conditions under which b can occur, and
2. the source code of p is available, and
3. P has finite states or an equivalent finite abstract program P_{abs} can be constructed for the properties of interest, and
4. the program P or its reduced equivalent program P_{abs} is written in a modeling language that is acceptable by SMC.

Observation 3. *Model checking tools can handle observable, non-observable, reproducible, and non-reproducible bugs, provided that the size of the program is finite or a sound abstraction can be developed to bring the program within the feasibility bound of model checking, and provided that a specification is available for the bug of interest.*

Bug Detection Properties: Based on the characteristics of the three bug detection techniques discussed, one can make the following general observations. Model checking is more expensive than static analysis, requiring longer running times and more resources. Static analysis is not as accurate as model checking, and testing is not as complete as model checking. Testing suffers from coverage challenges: it is challenging to cover all possible executions of the program, in particular for programs with an infinite input space where this becomes prohibitively expensive. To ensure the correctness of the program for all inputs, a correctness specification must exist that can be formally analyzed.

Table 2 summarizes the key differences and similarities between the three program analysis techniques using several criteria: code coverage, the need for code executability, accuracy of the analysis, the need for bug observability, the need for code availability, and automation of the technique. By code coverage, we mean the number of feasible execution paths of the program that the technique can cover during the analysis, and accuracy indicates whether the detected bug is a real bug.

4. APR approaches

This section describes four APR approaches that can handle different classes of bugs. The four approaches combine the detection power of dynamic analysis, static analysis, and model checking techniques to improve the reliability of existing APR techniques. We discuss the applicability of these approaches to three classes of bugs: arithmetic bugs (observable bugs), non-functional bugs (non-observable bugs), and liveness bugs (partially observable bugs).

An APR approach generally consists of four steps: fault identification, fault localization, patch generation, and patch validation. The most challenging step in the APR process is the patch validation step, in which the generated patch is extensively evaluated to ensure that the bug is resolved, and that the patch does not introduce any unwanted behavior. In dynamic APR, as the name implies, the patch validation step is primarily performed using test cases. Since these rarely capture the expected behavior in full detail, the technique suffers from the so-called patch overfitting problem, where the patched program may pass the tests in the given test suite, while it is failing for valid inputs not covered by the test suite. It is therefore desirable to combine the power of different analysis techniques while taking into account the distinctive properties of each class of bugs. This leads to our examination of the following four APR approaches:

1. *dynamic APR*: in which fault identification and patch validation are performed using dynamic analysis techniques. GenProg (Le Goues et al., 2012) is an example of a dynamic APR tool;
2. *static APR*: in which fault identification and patch validation are performed using static analysis techniques;
3. *dynamic–static APR*: in which fault identification and patch validation are performed using a combination of static and dynamic analysis, i.e., using test cases and static analysis tools developed for the same class of bugs;

4. *formal APR*: in which fault identification and patch validation are performed using formal methods and verification techniques such as model checking.

A hybrid APR approach aims to improve the overall quality of the generated repairs and alleviate the patch overfitting challenge of dynamic APR systems.

4.1. Arithmetic bugs

Arithmetic calculations affect a wide variety of applications, including safety-critical systems such as control systems for vehicles, medical equipment, and industrial plants. The key properties of arithmetic bugs can be summarized as follows:

1. Arithmetic bugs are observable classes of bugs or can be easily made observable to external observers: the root causes of arithmetic bugs are limited and easy to identify.
2. Arithmetic bugs are tractable bugs with finite traces.
3. Arithmetic bugs introduce various erroneous behavior to the program in which they occur: they may cause the program to crash or may produce incorrect outputs.

These properties make them directly amenable to dynamic (i.e., test-based) APR. Note that many of the available dynamic APR systems rely explicitly or implicitly on observability strategies to expose this class of bugs (e.g., integer overflow, division by zero, etc.). There are also several static analysis tools that can handle arithmetic bugs. Thus, arithmetic bugs can be repaired using dynamic APR, static APR, or dynamic–static APR. These repair approaches differ mainly in the correctness specification used to validate generated patches for the detected arithmetic bug. In dynamic APR, the specification is captured by the test cases, while in static APR, the specification is captured by the formal bug detection rules.

Example 1. Integer overflow (IO) is a type of arithmetic bug that occurs when the computation of an arithmetic operation, such as multiplication or addition, exceeds the maximum size of the integer type used to store it. IO bugs are an observable class of bugs, and thus they are amenable to dynamic APR. IO bugs are also amenable to static APR, and there are several reliable static analysis tools available that can address IO bugs (Muntean et al., 2021; Al-Bataineh et al., 2021a). Thus, devising a hybrid static-dynamic APR system for IO bugs is feasible and will help increase confidence about the soundness of the generated repairs.

Observation 4. *Arithmetic bugs are observable, tractable, and reproducible classes of bugs with finite execution traces. They are amenable to dynamic and static APR since the root causes of arithmetic bugs are easy to identify.*

4.2. Non-functional bugs

Non-functional bugs (Jin et al., 2012; Radu and Nadi, 2019; Al-Bataineh et al., 2021b) are a class of bugs that affect the way a program operates, rather than the functional behavior of the program. Inefficiently written loops in programs and synchronization issues in concurrent programs (i.e., using a large unnecessary number of locks) can be viewed as examples of non-functional bugs. Non-functional bugs are as important as functional bugs. For example, energy consumption saving is getting more urgent, particularly for applications running on embedded systems and IoT in Smart Cities.

Fixing non-functional bugs is generally more complex than fixing functional bugs, since non-functional bugs can hide themselves well in the code. While most functional bugs can be detected through observing the erroneous behavior of bugs, a large percentage of non-functional bugs are detected through manual code review (Fagan, 1976; Gilb et al., 1993; Bacchelli and Bird, 2013). Non-functional bugs usually do not generate incorrect results or crashes. Therefore, they cannot be observed by checking the program output. The key properties of non-functional bugs can be described as follows.

1. Non-functional bugs are generally non-observable classes of bugs: they do not introduce direct observable erroneous behavior to the program in which they occur.
2. Non-functional bugs increase the anticipated running cost of a program (execution time, memory and energy consumption, etc.) due to the inefficient use of resources.

Depending on what quality attributes are considered, programs may suffer from many different types of non-functional bugs. For example, consider the class of non-functional bugs that adversely affect the run-time costs of executing the program, such as execution time, memory consumption, and energy consumption. To expose such types of bugs, the program may need to be augmented with additional variables or online monitors that can be used to observe aspects at runtime (Al-Bataineh et al., 2021b).

Therefore, there is a need to develop effective bug detection tools that can be used to expose non-functional bugs at the early stages of the software development life cycle. Specifically, this requires efficient profiling techniques and oracles that help decide whether the program's non-functional requirements are met under a particular workload. Unfortunately, the lack of effective test oracles for non-functional bugs is a well-known problem that will need to be addressed in the future.

Observation 5. *With a few notable exceptions, such as performance bugs, non-functional bugs are typically non-observable types of bugs because a program with a non-functional bug runs normally and terminates normally. Thus, they are not directly amenable to traditional dynamic bug detection techniques that rely on test cases and observing a program's outputs.*

4.3. Liveness bugs

In this section, we discuss a class of bugs that has received little attention from the APR community, namely liveness bugs. A *liveness property* asserts that "something good will eventually occur when executing a program" (Lamport, 1977; Alpern and Schneider, 1985). Freedom of starvation and program termination are examples of liveness properties. A program that violates a liveness property cannot make progress and thus suffers from a *liveness bug*.

Two fundamental properties make detecting and repairing liveness bugs far more challenging than other classes of bugs. First, the behavioral effects of triggering a liveness bug are generally unobservable. Second, identifying a liveness bug requires finding an infinite execution that will never satisfy the desired liveness property (Alpern and Schneider, 1987), making it impractical to find such bugs using dynamic analysis. Therefore, detecting and repairing liveness bugs generally require more sophisticated repair algorithms since they must be able to generate a finite representation of infinite counterexamples.

To better understand the complexity of repairing liveness bugs, we study a subset of liveness bugs known as termination bugs. There are two advantages to this choice: On the one hand, a termination bug is a specific type of liveness bug whose repair is essential for ensuring software reliability. On the other hand, by examining techniques for handling termination bugs, we gain knowledge that can help address other liveness bugs. Termination bugs have the following specific properties:

1. Termination bugs are *partially observable*: an observer monitoring the behavior of a non-terminating loop program will not witness any erroneous behavior but rather experience unexpectedly long execution times. The only observable behavior of termination bugs is that the program becomes *non-responsive at runtime*.

2. Termination bugs have *infinite faulty traces*: a counterexample to a termination property violation is infinite.

Observation 6. *Termination bugs are an example of partially observable bugs with infinite execution traces. Termination bugs are amenable to dynamic–static APR and static APR.*

5. Hybrid APR for termination bugs

This section describes a hybrid program repair approach for termination bugs that combines the strengths of termination provers with those of software model checkers. Such a combination has two key advantages. First, it considerably reduces the overall computational complexity of the problem by avoiding the exhaustive exploration of the program's input space. Second, it helps avoid the known overfitting problem by generating verified repairs for termination bugs.

The presence of loops in programs can complicate the detection of certain classes of bugs. Recall the two types of hang bugs from Definition 5; It is not clear how one can distinguish between the following two types of loops using a dynamic analysis technique: (i) inefficiently written loops that introduce a soft hang bug, and (ii) incorrect infinite loops that introduce a hard hang or termination bug.

Earlier work (Le Goues et al., 2015) that evaluated the effectiveness of different APR tools on the ManyBugs and IntroClass datasets, used a simple timeout mechanism to handle termination bugs in these two datasets: when the execution time of a program exceeds some pre-specified period, they consider the program to be *likely* non-terminating due to an infinite loop. Marcote and Monperrus instrument loops with iteration counters that are monitored to detect infinite loops (Marcote and Monperrus, 2015). Both options *can* lead to false conclusions about the program under analysis (i.e., even when the watchdog triggers, the program may not have a termination bug but suffer from a soft hang bug). Moreover, hang bugs can have complicated causes: programs that become unresponsive may contain deadlocks, infinite loops, or other bugs that lead to non-termination but are *not* infinite loops.

An effective solution for addressing termination bugs is to apply *termination provers*: tools that can check combinations of many complex termination criteria. They take a program as input and return one of three answers: *terminating* (TR), *non-terminating* (NT), or *unknown* (UN). In general, when the prover returns definite answer for a given program (i.e., $answer \in \{TR, NT\}$), the answer is valid with high confidence. Termination provers have been successfully used to analyze termination of a wide variety of loop programs (Berdine et al., 2007; Chawdhary et al., 2008; Tsitovich et al., 2011; Gulwani et al., 2009a,b; Bradley et al., 2005; Cousot, 2005; Gupta et al., 2008; Harris et al., 2010; Kroening et al., 2010; Podelski and Rybalchenko, 2004).

Definition 11 (*Valid Termination Bug Repair*)**.** Let P be a buggy non-terminating program with a set of inputs I, φ_{beh} a specification of expected behavior of P, and φ_{reach} a specification that checks the reachability of some halting statement of P. We say that P' is a valid repair of P iff for every input $i \in I$ we have $(P', i) \vDash \varphi_{beh}$ and $(P', i) \vDash \varphi_{reach}$.

In other words, the patched version P' should preserve the expected behavior of P, and it should terminate.

5.1. Termination bugs in sequential programs

To repair termination bugs in sequential programs, we first generate plausible patches, and then use termination provers to check the correctness of these patches. AProVE (Giesl et al., 2014) and 2LS (Chen et al., 2015) are among the most reliable candidates to analyze the termination of sequential programs.

Definition 12 (*Validity of Patches for Termination Bugs in Sequential Programs*)**.** Let P_s be a non-terminating sequential program and $T = (T_p \cup T_f)$ be a test suite consisting of passing test cases T_p and failing test cases T_f. Let P'_s be a candidate patch of P_s and TP be a termination prover that returns one of the verification answers $\{TR, NT, UN\}$. We say that P'_s is a valid patch of P_s iff all of the following conditions hold:

1. all failing test cases from T_f pass on program P'_s,
2. none of the passing test cases from T_p fail on program P'_s,
3. termination prover TP returns "TR" when analyzing termination of P'_s.

The soundness of generated patches for termination bugs in sequential programs can be captured formally as:

$$\varphi_{seq} = (\forall_{t \in T}(P'_s \vdash t) \wedge TP(P'_s) = TR) \tag{1}$$

where T is the set of available test cases, and $P'_s \vdash t$ indicates that patch P'_s runs successfully against test t.

Fig. 1 sketches a hybrid repair procedure for termination bugs in sequential programs. The algorithm takes five inputs: the buggy sequential program P_s, a specification of expected behavior φ_{beh}, a test suite T_s, a termination prover TP, and the allocated time budget $TimeBudget$. It uses two functions: (i) $faultLocalizer(P_s, CE, T_s)$ computes the set of suspicious statements $SuspStats$ whose mutation may lead to generate a valid patch. It finds these suspicious statements by combining the counterexamples CE generated by termination prover TP with the results of spectrum-based fault localization (Jones et al., 2002; Naish et al., 2011; Heiden et al., 2019; Xie and Xu, 2021) on the test suite (executed with a timeout mechanism to avoid getting stuck in infinite loops); (ii) the function $mutate(P_s, SuspStats)$ is used to construct the patch space (i.e., patch generation) by mutating the computed set $SuspStats$ that may affect the truth value of the termination condition of the detected buggy non-terminating loop in P_s.

5.2. Termination bugs in concurrent programs

In the automated repair of concurrent programs, the goal is to generate a patch that ensures that a concurrent program is correct under all interleavings. It is difficult, if not impossible, to examine all possible executions of a concurrent program using dynamic analysis techniques. Therefore, a concurrent program cannot be debugged and repaired in the same manner as sequential programs. In the case of concurrency, it usually refers to action interleaving. That is, if the processes p_i and p_j are in parallel composition ($p_i \parallel p_j$) then the actions of these will be interleaved. Each process executes a sequence of actions (sub-program), then the set of possible interleavings of several processes consists of all possible sequences of actions. Before proceeding further, let us introduce the notion of successful termination in concurrent programs.

Definition 13 (*Termination of Concurrent Programs*)**.** Let $P_c = (p_1 \parallel p_2 \parallel \ldots \parallel p_n)$ be a concurrent program that consists of a collection of sub-programs, where each process p_i executes a sub-program. Let H_i be the set of halting statements at the sub-program executed by process p_i. We say that P_c is terminating iff every sub-program is eventually terminating. That is, the sub-program executed by process p_i eventually executes some halting statement $s \in H_i$ and terminates.

Definition 14 (*Termination Failures in Concurrent Programs*)**.** Let $P_c = (p_1 \parallel p_2 \parallel \ldots \parallel p_n)$ be a concurrent program that consists of a collection of sub-programs executed by processes $p_1, \ldots, p_n$. The program P_c fails to terminate iff:

1. there exists a logical bug that leads to an infinite loop, or
2. there exists a concurrency bug, such as deadlock or livelock, that prevents the program from making any further progress in reaching a halting statement and terminating.

```
1: Inputs: P_s, φ_beh, T_s, TP, TimeBudget
2: Output: P_repaired
3: PatchSpace := ∅, P_repaired := NoPatchFound, found := false
4: (∗, CE) := TP(P_s)                          # get CE, verdict known to be NT
5: SuspStats := faultLocalizer(P_s, CE, T_s)
6: PatchSpace := mutate(P_s, SuspStats)                    # patch generation
7: while PatchSpace ≠ ∅ ∧ TimeBudget > 0 ∧ ¬found do
8:     select patch from PatchSpace
9:     if patch ⊨ φ_beh then                          # 2-step patch validation
10:        if TP(patch) = (TR, ∗) then
11:            P_repaired := patch
12:            found := true
13:        end if
14:    end if
15: end while
16: return P_repaired
```

Fig. 1. Repair algorithm for sequential programs.

Thus, if deadlocks and livelocks are formally proven to never occur in the program-under-analysis, and all loops in the sub-programs are proven to be terminating, then one can conclude that the concurrent program is terminating. The distinction between the two causes of termination failures in concurrent programs (logical bug or concurrency bug) helps to select a strategy for fixing the detected termination bug.

Combining Model Checking and Termination Provers: Repairing termination bugs in concurrent programs can be a computationally complex task. This is mainly because termination bugs in concurrent programs can be caused by either a logical or concurrency bug. Furthermore, the vast number of possible interleavings of parallel processes of a given concurrent program can increase the complexity of the repair problem. Therefore, it is necessary to employ both termination provers and model checking to reduce the computational complexity of the problem. Fortunately, we know how to write specifications to check the absence of concurrency bugs in concurrent programs (Gupta et al., 2018; Lin and Kulkarni, 2014; Zhou et al., 2017). In Definition 15, we describe the conditions that are necessary to ensure the correctness of the generated patches for termination bugs in the buggy non-terminating concurrent program.

Definition 15 (*Validity of Generated Patches for Termination Bugs in Concurrent Programs*)**.** Let $P_c = (p_1 \parallel p_2 \parallel \ldots \parallel p_n)$ be a buggy non-terminating concurrent program, TP be a termination prover, and SMC be a software model checker. Let $\varphi_{deadlock}$ and $\varphi_{livelock}$ be specifications that check respectively the absence of deadlocks and livelocks in P_c and φ_{beh} be a specification that captures the expected behavior of P_c. We say that a candidate patch P_c' for the buggy program P_c is a valid patch iff it meets the following requirements:

1. the prover TP returns "terminating" when analyzing termination of the sub-program executed by process p_i, and
2. the checker SMC returns "holds" when checking the specifications $\varphi_{deadlock}$ and $\varphi_{livelock}$ against P_c', and
3. the checker SMC returns "holds" when checking the specification of expected behavior φ_{beh} against the patch P_c'.

Formally, we can capture the requirements described above in a 4-part correctness specification of the following form:

$$\varphi_{con} = \forall_{p_i \in P_c'}(TP(p_i) = TR) \wedge SMC(P_c', \varphi_{deadlock}) \wedge SMC(P_c', \varphi_{livelock}) \wedge SMC(P_c', \varphi_{beh}) \quad (2)$$

A key challenge when dealing with termination bugs is to ensure that the generated repair guarantees not only the termination of the program for each possible input, but also the semantic preservation of the program. This requires the analysis of a composite correctness property that checks both termination and semantic preservation. By using formula (2), we entirely avoid the patch overfitting problem, which is one of the major problems of dynamic APR. Formula (2) uses sequential termination provers to check the absence of infinite loops in each individual process. This can be performed while abstracting away concurrency details that are irrelevant to the local computations of processes. Note that one cannot prove the termination of program P_c by simply applying a sequential termination prover: a sound proof of termination must consider all possible interactions among the sub-programs of P_c.

An alternative way to use model checking in detecting and repairing termination bugs of the concurrent program is to reduce the termination problem to the reachability analysis problem. That is, to check whether each process will eventually reach some halting location and terminate. However, the feasibility of the approach relies mainly on the size and number of processes of the buggy program under analysis (i.e., computing state-reachability is known to be PSPACE-complete when processes are finite state Kozen, 1977). The reduction of the termination problem to the reachability problem in model checking leads to the following temporal formula

$$\varphi'_{con} = \forall_{p_i \in P_c'}(\mathbf{AF}(p_i.\ell_h^{(j)} \mid \ell_h^{(j)} \in H_i)) \wedge SMC(P_c', \varphi_{beh}) \quad (3)$$

where **A** is a temporal path quantifier which should be read as "for all paths", and **F** is the "future" temporal operator (Pnueli, 1977). Intuitively, formula (3) checks whether for each reachable execution path of process p_i, some halting location $\ell_h^{(j)} \in H_i$ will eventually be reached. However, the proper termination analysis of the concurrent program P_c using formula (3) requires the precise computation of the

1: **Inputs**: $P_c,\ \varphi_{beh},\ \varphi_{deadlock},\ TP,\ SMC,\ TimeBudget$
2: **Output**: $P_{repaired}$
3: $PatchSpace := \emptyset,\ P_{repaired} := NoPatchFound,\ found := false$
4: $(Outcome_1, CE_1) := SMC(P_c, \varphi_{deadlock})$
5: **if** $(Outcome_1 = fails)$ **then** # P_c contains a deadlock
6: $SuspStatsD := faultLocalizer(P_s, CE_1)$
7: $PatchSpace := mutateConcur(P_c, SuspStatsD)$
8: **end if**
9: $(Outcome_2, CE_2) := TP(P_c)$
10: **if** $Outcome_2 = NT$ **then** # P_c contains an infinite loop
11: $SuspStatsL := faultLocalizer(P_s, CE_2)$
12: $PatchSpace := PatchSpace \cup mutateLogic(P_c, SuspStatsL)$
13: **end if**
14: **while** $PatchSpace \neq \emptyset \wedge TimeBudget > 0 \wedge \neg found$ **do**
15: **select** $patch$ **from** $PatchSpace$
16: **if** $patch \models \varphi_{beh}$ **then** # 2-step patch validation
17: **if** $TP(patch) = (TR, *) \wedge SMC(patch, \varphi_{deadlock}) = (holds, *)$ **then**
18: $P_{repaired} := patch,\ found := true$
19: **end if**
20: **end if**
21: **end while**
22: **return** $P_{repaired}$

Fig. 2. Repair algorithm for concurrent programs.

set of halting locations H_i for each process p_i of the patched concurrent program P_c'.

There are several software model checkers[1] that can be used to verify reachability properties and detect concurrency bugs, including VeriSoft (Godefroid, 1997), Java Pathfinder (Havelund and Pressburger, 2000), CMC (Musuvathi et al., 2002), DIVINE (Baranová et al., 2017), and GMC (Thompson and Brat, 2008). DIVINE is a modern, explicit-state model checker that can verify programs written in multiple real-world programming languages, including C and C++. On the other hand, GMC is a model checker based on the generic Monte-Carlo model-checking algorithm. It takes as input a C program, the target program to be verified, and the linear temporal logic specification that needs to be checked.

There are also a few termination provers that can be used to analyze the termination of concurrent programs. For instance, Cook et al. (2007) have extended the termination prover T2 (Brockschmidt et al., 2016) to support the analysis of concurrent programs, which can be used to validate generated patches for termination bugs in concurrent programs. Termination checker T2 supports nested loops, recursive functions, pointers, side-effects, and function-pointers, as well as concurrent programs. Of course, the prover cannot handle termination of all concurrent programs since the general problem is undecidable. The use of concurrent termination provers leads to the following specification

$$\varphi''_{con} = \forall_{p_i \in P'_c}(TP(p_i) = TR) \wedge SMC(P'_c, \varphi_{beh}) \quad (4)$$

[1] Unlike traditional model checking, a software model checker does not require a user to manually construct an abstract model of the program to be checked, but instead, the tool works directly on the program's source code.

The termination provers AProVE, 2LS, and T2 can be viewed as complementary tools: it is possible that some tool fails to detect certain forms of termination bugs while other succeeds, depending on the implemented theory and the complexity of the program under analysis. Therefore, termination checkers can be run in parallel to expose termination bugs.

While checking the satisfaction of formula (2) may require higher computational complexity than formulas (3) and (4) (i.e., it employs both termination provers and software model checkers to check the absence of deadlocks, livelocks, and infinite loops), it has several advantages. First, it helps identify the root causes of non-termination in the program-under-repair. Second, the patch validation approach that uses formula (2) can benefit from the counterexamples generated by both termination provers and software model checkers. This helps to develop effective program synthesis for termination bugs.

Repairing Termination Bugs in Concurrent Programs: To fix termination bugs in concurrent programs, it is first essential to identify the root cause of the termination bug since both logical and concurrency bugs can cause non-termination. On the one hand, there are several mechanisms for handling deadlocks in concurrent programs (Lin and Kulkarni, 2014; Zhou et al., 2017; Cai and Cao, 2016). On the other, repair algorithms based on genetic programming can help fix termination bugs that occur due to infinite loops (Le Goues et al., 2012; Yu et al., 2020).

Fig. 2 sketches a hybrid repair procedure for termination bugs in concurrent programs. It uses three helper functions: (i) $faultLocalizer(P_c, CE)$ finds the statements that are suspected to be faulty in P_c using the counterexamples CE generated by termination prover TP and software model checker SMC, (ii) $mutateConcur(P_c, SuspStatsD)$ constructs the patch space for a concurrency bug by mutating the synchronization

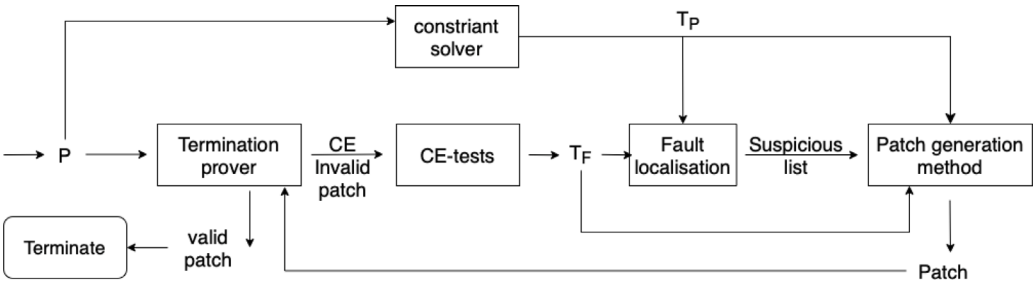

Fig. 3. High-level overview of our iterative hybrid repair methodology for termination bugs that is guided by counter-examples (CEs) generated by the termination prover.

Table 3
A comparison of the complexity of the repair problem of termination bugs in sequential and concurrent programs.

Program class	Root causes of non-termination	Feasible APR
Sequential programs	Infinite loops	Dynamic–static
Concurrent programs	Infinite loops, deadlocks, or livelocks	Static or formal
Program class	**Patch validation procedure**	**Patch validation tools**
Sequential programs	Test cases and termination provers	AProVE, 2LS, T2, and GMC
Concurrent programs	Termination provers and SMC	T2 and GMC

primitives of the program, and (iii) $mutateLogic(P_c, SuspStatsL)$ constructs the patch space for a logical bug by mutating expressions that affect the control of faulty loops.

Generating verified repairs of termination bugs in both sequential and concurrent programs is a challenging open problem that requires formal analysis techniques. The application of the termination provers 2LS and AProVE on the programs in the two datasets, SNU real-time benchmark and the Power-Stone benchmark suite (Ku et al., 2007), show that the tools are able to successfully prove termination of around 85% of the examined programs using very little computational time (a few seconds). This demonstrates the feasibility of using termination provers to validate the generated patches of termination bugs. We build on this result in the next section, which discusses our preliminary empirical investigation of integrating termination provers for sequential programs with dynamic APR approaches to generate verified repairs for termination bugs and performance bugs.

Table 3 compares the complexity of termination bugs in both sequential and concurrent programs using several criteria: (i) root causes of termination bugs, (ii) feasible APR approach to be applied for termination bugs in both classes of programs, (iii) patch validation procedure to validate generated patches, (iv) patch validation tool that can be used to check termination.

6. Empirical evaluation

The proposed repair method integrates formal methods with APR and is mainly designed to address a class of bugs that are challenging to handle using dynamic analysis techniques. Specifically, bugs that do not introduce observable erroneous behavior to the program at which they occur. This class of bugs can be effectively addressed using formal analysis techniques such as software model checkers, static analysis, and theorem and termination provers. Examples of such type of bugs include liveness and non-functional bugs. In this section, we demonstrate the advantages of using hybrid repair approaches to two categories of bugs, namely termination bugs and performance bugs, which are difficult to handle using dynamic repair approaches.

6.1. Research questions

We consider two key research questions in the empirical evaluation of the proposed hybrid repair approach on termination and performance bugs. The first aims to assess the effectiveness of addressing termination bugs, and the second assesses the effectiveness of addressing performance bugs that occur due to inefficient loops.

$RQ1$: How effective is hybrid APR in fixing termination bugs?
$RQ2$: How effective is hybrid APR in fixing performance bugs?

6.2. Prototype implementation

A high-level overview of the methodology that can be used to produce a verifiable fix for a bug found by a termination prover is provided in Fig. 3. Five major components make up the methodology: (i) a termination prover component to detect termination bugs and assess patch correctness with respect to the termination requirement, (ii) a CE-test conversion component to turn formal CEs into failing tests (T_F), (iii) a constraint solver (CS) to generate passing test cases corresponding to bug-free program instances (T_P), (iv) an FL method to produce a list of suspicious statements related to the bug being fixed, and (v) a patch generation method to generate candidate patches for the termination bug. The process is iterative in nature and comes to an end when the termination prover successfully validates the program.

6.3. Dataset for non-terminating loop programs

The dataset for non-terminating loop programs that we consider in this paper was initially created for the assessment of the performance of the available termination provers (Shi et al., 2022). However, since there are no test suites connected to the non-terminating programs provided in the dataset, APR tools cannot be used directly to handle these programs. To produce patches for a specific buggy program, the majority of the current APR tools require a test suite. It is, therefore, necessary to extend the dataset to contain failing and passing test cases for each non-terminating loop program. For this task, we use the UAutomizer termination prover, which has the capability to generate counterexamples when the loop program fails to terminate. The generated counterexamples can be used to assist in creating failing test cases for the buggy non-terminating loop program. The issue then becomes how we categorize test cases for termination bugs as failing or passing. We say that a test t is a *passing test* for a loop program L if L succeeds in terminating and produces the correct outputs. On the other side, we say that t' is a *failing test* for L if either L fails to terminate or terminates but gives wrong results.

An intriguing aspect of the dataset is that it includes a fixed version (a human-written patch) for each buggy non-terminating loop program, which can be used to assess the efficacy of patches generated by APR tools. The given patched versions of the loop programs are used to produce passing test cases that can be used to direct the APR engine

to produce fixes for the original buggy loop program. In our setup, we employ the provided buggy program and a termination prover to automatically generate a set of failed test cases, while the provided human-written patches are used to generate a set of passing test cases. These passing and failing test cases are given to an APR tool along with the original buggy program to create a patch.

A pre-configured timer whose value represents an anticipated upper limit for termination for the loop program under repair is added to failing test cases. There are many benefits to using a timer for fixing termination bugs. First, it is important to utilize a timer to force the end of the analysis performed by the APR tool because the loop program is known to be non-terminating under the specified failing test. Second, by utilizing a timer with a predetermined upper bound, it notifies the APR tool that the bug under repair belongs to a specific class of bugs that have an impact on the program's termination. This is essential because it enables the APR tool to modify the original buggy program so that it ends within the predetermined upper bound. This would result in a patch that fixes the termination bug. However, using merely a timer in the test suite is insufficient to produce reliable patches for termination bugs, as we will show when we examine the patches produced by APR tools. Information on the semantics or functionality of the loop program that is being repaired must be included in the test suite. The APR tools can quickly add a halting statement at any random location in the code when a program termination fault occurs. Such repairs are unacceptable since they have a negative impact on the program's semantics.

6.4. Selected termination provers and APR tools

To evaluate the effectiveness of APR tools in repairing termination bugs, we select two APR tools: the search-based repair tool GenProg (Le Goues et al., 2012) and the semantic-based repair tool FAngelix (Yi and Ismayilzada, 2022). Both FAngelix and GenProg are chosen as representatives of tools for mutation-based and semantic-based repairs, respectively. These are general-purpose repair tools that can be used to fix a range of program bugs, including loop program bugs. The termination prover UAutomizer is chosen, and it is integrated with the APR tools indicated above. Among the termination provers available, UAutomizer was chosen because it performs the best on loop programs and can generate counterexamples when the loop program under investigation fails to terminate. The termination prover UAutomizer is employed several times during the repair process: initially, to identify the non-termination bug and build a counter-example; and, secondly, to confirm the validity of the set of patches produced by the APR tool, which should pass all of the test cases.

6.5. Factors affecting repair framework for termination bugs

As indicated earlier, the objective of termination repair is to correctly recover termination without adversely affecting the semantics of the loop program. In comparison to other classes of bugs, this one is, in fact, difficult to fix. The patch's validity must be verified in relation to both the termination requirement and the semantics requirement. Note that a fix for a termination bug introduced by an APR tool typically makes one of the following forms: (i) updating the expressions for some control variables of the buggy loop or, (ii) updating the termination condition of the loop, (iii) inserting a halting statement at some location of the loop, or a combination of these. The number of times the loop is iterated would change as a result. This would have an impact on the loop's calculations and results. Due to this, it is necessary to use the composite correctness property to validate the generated patches for termination bugs. Before proceeding further, let us discuss the key assumptions we make about the presented repair framework for termination bugs.

1. The availability of termination provers that produce counterexamples. This is crucial as it allows us to check the overall accuracy of the patches created in regard to the termination requirement. Specifically to determine whether each and every input results in the corrected program terminating. Additionally, by utilizing the previously discussed CE-guided repair approach, the created CEs can be used to gradually enhance the quality of the patches that are generated. The employment of termination provers is the only feasible and practical method for proving the general termination of loop programs, assuming that the termination problem of the particular loop program is a solvable problem.
2. The availability of failing and passing test cases for termination bugs. The quantity of information encoded in the test cases, as well as the number of accessible passing and failing test cases, affect the quality of the patches that are created for termination bugs, as we will see in more detail later. There are two parts to validating that a loop program with a termination bug has been successfully repaired. The first step is to assess the desired behavior of the program that should be maintained when the program is modified by the APR tool. Secondly, the anticipated upper bound for termination of the program being repaired should be asserted. We assume here that a test can be constructed that includes information representing the desired behavior as well as the expected upper bound for termination. A test $t \in T$ for a termination bug is defined as follows

$$t = (\texttt{timer} \leq val \wedge \{(in_1, \ldots, in_n)\} \wedge \{(out_1, \ldots, out_n)\}) \quad (5)$$

   where the collection of inputs to the non-terminating loop program L is represented by $in_1, \ldots, in_n$, and the set of *observable expected outputs* of the program corresponding to those inputs is represented by $out_1, \ldots, out_n$, respectively. We firmly believe that using a timer is advantageous for both passing and failing tests. The timer can be used to instruct the APR tool to preserve the behavior for passing tests without increasing processing or execution times (if possible). The timer is used to notify the APR tool that failing tests have a specific type of bug that affects the program's execution time. This would help the APR tool to generate a patch that fixes the termination bug.
3. The availability of general-purpose APR tools. When we refer to general-purpose APR tools, we mean those which were not created to tackle certain bug types, built on, or configured for particular datasets. This is essential so that programs with arbitrary bugs can be handled by the tools. In our quest for such kinds of tools, we found that search-based APR tools like GenProg and semantic-based APR tools like FAngelix are suitable options for handling termination bugs. In terms of generality in the context of dynamic APR, they are general APR tools for C programs.

6.6. Hybrid APR tools for handling hard hang bugs (termination bugs)

Although adding more details to test cases may increase the time it takes to fix termination bugs, it promotes the creation of high-quality patches. While adjusting the level of detail in the provided test suite, we assess how well hybrid APR tools handle termination bugs. More specifically, we aim to evaluate the quality of generated patches under each of the following hypotheses:

1. The quality of patches produced by APR tools when only the timing information on the upper bound of the execution time is provided. When only the desired or anticipated termination time is given, this configuration aims to evaluate the effectiveness of APR tools in repairing termination bugs. The objective is to determine whether this is sufficient to create trustworthy patches for termination bugs.

2. The quality of patches produced by APR tools when both the timing information and expected behavior are provided. This configuration intends to assess the APR tool's performance in creating trustworthy patches when both the expected termination time and expected outputs are specified. This setup, however, makes the assumption that the intended time for termination, as well as the anticipated outputs corresponding to the inputs that result in the termination bug, are known.

6.7. Hybrid APR tools for handling soft hang bugs (performance bugs)

In this section, we explore the possibility of building hybrids based on available APR tools that can be used to optimize inefficiently written loop programs in addition to producing accurate patches. We believe this objective can be accomplished if test suites were more robust and thorough and included information about both the expected upper bound for termination and the expected behavior of the program. The goal is to employ the APR technology outside of its conventional application by not only repairing the bug but also, if possible, optimizing the buggy loop program by reducing the observed execution time of the program.

Hypothesis: By adding additional variables to the program (such as timing variables, loop counting variables, etc.) whose values have no direct or indirect impact on the behavior of the program, we can transform the optimization problem of inefficient loops into a repair problem and redefine the concepts of passing and failing test cases.

A number of questions need to be addressed in order to use the APR technology in optimizing loop programs. For example, how should the APR tools be configured to deal with performance bugs that occur due to inefficient loops? How should the notions of passing and failing test cases be defined for performance bugs? How can the validity of patches created to fix performance bugs be verified? And last but not least, what correctness property can be used to validate the desired optimization?

Performance bugs are a form of non-functional bugs that unnecessarily lengthens the program's execution time, but *they are not behavioral bugs*. Given that the correct behavior is known and can be deduced from the original program, this actually makes the validation process for this type of bugs somewhat easier than it is for functional bugs whose correctness specifications are typically unavailable. In this situation, the original program might be utilized as a reference program to direct the fixing of the identified performance bug. The semantics preservation of the program after being repaired must still be verified, because fixing non-functional bugs often have a direct impact on how the program behaves.

We now turn to discuss the notions of passing and failing tests and validity criteria for performance bugs. A failing test in the context of APR is an input that results in an execution that does not satisfy the desired specification (e.g., an assertion, timeout, formal specification of the intended behavior, access control policy, etc.), whereas a passing test is an input that results in an execution that does. When it comes to performance bugs, passing and failing tests are specified in relation to a timed specification: a specification states that the program must terminate and produce correct results within a specified time bound.

Definition 16 (*Passing and Failing Tests for Performance Bugs*)**.** Let L be a loop program containing a performance bug b, and $T = (T_P \cup T_F)$ be a test suite developed w.r.t. bug b, where each test $t \in T$ is associated with a time-bound U_t representing the expected upper bound for termination of program L under test t. We say that a test $t \in T_P$ is a passing test if L terminates and produces a correct output within the time bound U_t. On the other hand, we say that a test $t' \in T_F$ is a failing test if P terminates and produces a correct output but exceeds the time bound $U_{t'}$.

We refer to runs corresponding to passing tests of performance bugs as *fast runs* and runs corresponding to failing tests as *slow runs*. Both fast and slow runs terminate normally and produce correct results. As one can see, the ability to compute the predicted upper bound for termination for each specific input is a requirement for the design of passing and failing tests for performance bugs. The predicted number of iterations and the time required for each iteration needs to be estimated in order to achieve this. The average time required for each iteration can be estimated using passing tests.

Definition 17 (*Validity of Patches for Performance Bugs*)**.** Let L be a loop program containing a performance bug b and $T = (T_P \cup T_F)$ be a test suite developed w.r.t. b, where T_P lead to fast runs and T_F lead to slow runs. We say that a patch pt is a valid patch for b if the following conditions hold:

1. for each test $t \in T$ the original program L and the generated patched program $pt(L)$ produces the same output, and
2. for each test $t \in T_F$ the observed execution time in program $pt(L)$ is smaller than to the one observed in program L (i.e., the generated patch transforms slow runs into fast runs).

The above two requirements can be captured formally as follows

$$\varphi_{validity} = \forall_{t \in T}\ (output(L,t) = output(pt(L),t)) \wedge \forall_{t \in T_F}(time(pt(L),t) < time(L,t))$$

Observation 7. *One of the main characteristics of performance bugs is that since the bug is not a functional bug, a portion of the correctness specification related to the semantics of the program is implicitly known (i.e., the original program itself can be used as a reference program when repairing the bug). This facilitates the validation procedure for the generated optimized versions.*

7. Experimental results

Open Science: To support reproducible research, we publish a replication package containing the source code for the 15 termination bugs and 2 performance bugs considered, the scripts to conduct the empirical investigation, as well as the outcomes of the hybrid repair processes conducted during the evaluation.[2]

7.1. RQ1: How effective is hybrid APR in fixing termination bugs?

Setup: This research question intends to assess the effectiveness of the hybrid repair approach that combines termination provers, execution time monitors, and existing APR tools in producing valid patches for termination bugs and investigate the kinds of patches that the tools provide when dealing with termination bugs. As mentioned earlier, termination bugs can be repaired in a variety of ways. For example, the bug may be repaired by inserting a halting statement, modifying the termination condition, modifying the update expression of some control variable, adding a new control variable, or a combination of these. However, these changes should be made while preserving the semantics or functionality of the loop program that is being repaired. During the analysis, we considered 15 infinite loops taken from the dataset described by Shi et al. (2022). As mentioned earlier, we extended the dataset of infinite loops by adding a set of failing and passing test cases to each loop program. The termination provers are used to create failing test cases (i.e., inputs corresponding to generated CEs are transformed

[2] Available at https://github.com/secureIT-project/extendingAPR, and archived on Zenodo https://doi.org/10.5281/zenodo.10397656.

into failing tests), whereas reference loop programs are used to create passing test cases.

Results: The first repair tool we examined in our repair approach is GenProg, a search-based repair tool. In a hybrid setting, the tool was able to generate patches for 5 out of the 15 considered infinite loops. The termination prover indicates the successful termination of the generated patches. The use of termination provers together with test suites was effective in generating patches that ensure the general termination of the non-terminating loop program being repaired but not necessarily maintaining the program's logic. However, when comparing the generated patches w.r.t. reference ones (human-written patches), we found that none of the patches generated by the tool was correct. The GenProg hybrid generates patches for termination bugs by merely inserting a halting statement at some random location of the loop program or swapping two statements that have some influence on the behavior of the program being repaired. We did not observe any case in which the GenProg hybrid generated a patch by modifying a conditional expression or an update expression of some of the control variables of the loop. The main cause for these issues is the incompleteness of the correctness specification (i.e., the test suite not covering enough of the desired behavior), which is a frequently observed drawback of search-based repair approaches. When integrating the hybrid repair approach with the tool FAngelix, we observed that the tool was unable to handle infinite loops effectively. This is primarily caused by the tool's inability to synthesize expressions that meet timed specifications, which are required when handling termination bugs.

Answer to RQ1: While not always maintaining the program's logic, integrating termination provers, execution time monitors, and test suites can assist in creating patches that guarantee termination. The analysis shows that search-based repair tools like GenProg can produce patches for infinite loops by inserting a halting statement at some location of the program or swapping statements that have some direct influence on the termination of the loop. On the other hand, due to the inability to synthesize expressions that meet timed specifications, the semantic-based repair tool FAngelix was unable to handle programs that contained infinite loops.

7.2. RQ2: How effective is the hybrid APR in fixing performance bugs?

Setup: This research question intends to assess the ability of the presented hybrid repair approach in fixing performance bugs, particularly soft hang bugs that unnecessarily increase the execution time of a loop program but does not adversely affect its functionality. To analyze this type of bugs, we use a time monitor in addition to the anticipated correct behavior. By imposing a rigid upper bound on the execution time, the APR tool might be able to improve the efficiency of the loop program by identifying a patch that preserves the functionality of the program while speeding up execution.

We consider two programs with performance bugs in order to assess how well the current APR tools can handle performance bugs. The first program (Listing 1) has a fairly straightforward, observable performance bug where we unnecessarily insert a sleep statement in the body of the loop. The second one (Listing 2) is based on a real-world flaw that occurred in Apache and has also been analyzed by other researchers (Song and Lu, 2017). Listing 2 shows this more challenging performance bug that may need to be repaired by restructuring a portion of the program.

To run APR tools on these programs, we need to develop passing and failing test cases. Test cases that lead to fast runs are considered as passing tests while test cases that lead to slow runs are considered as failing tests. A repair that transforms slow runs into fast runs while

```
1  int main() {
2      int n = __VERIFIER_nondet_int();
3      int s = 0;
4      for ( int i=0; i < n; i++ ){
5          s += i;
6          usleep(n * 0.01);
7      }
8      printf("%
9      return 0;
10 }
```

Listing 1: A simple performance bug containing unnecessary sleep statement

```
1  int found = -1;
2  while (found < 0 ) {
3    // Check if string source[] contains target[]
4    char first = target[0];
5    int max = sourceLen - targetLen;
6    for (int i = 0; i <= max; i++) {
7      // Look for first character.
8      if (source[i] != first) {
9        while (++i <= max && source[i] != first);
10     }
11     // Found first character
12     if (i <= max) {
13       int j = i + 1;
14       int end = j + targetLen - 1;
15       for (int k=1; j<end && source[j]==target[k]; j++, k++);
16       if (j == end) {
17         /* Found whole string target. */
18         found = i;
19         break;
20       }
21     }
22   }
23   // append another character; try again
24   source[sourceLen++] = getchar();
25 }
```

Listing 2: A challenging performance bug found in Apache

preserving the expected behavior of the original program is considered to be a valid repair.

When dealing with the second performance bug, we first run the termination prover to formally confirm that the program is really terminating and that the bug is a performance bug and not a termination bug. The main reason for this is that we discovered throughout the analysis that some inputs could result in extremely lengthy runs, and we want to rule out the potential of having inputs that can result in infinite runs. This gives us the ability to set up the APR tools to address a performance bug rather than a termination bug. This also impacts the validation procedure that will be used to examine the validity of generated patches. Note that due to the ineffectiveness of the tool UAutomizer in handling loop programs with character arrays, we instead analyze a semantically equivalent version of the program using integer arrays. According to UAutomizer, the program is terminating, which indicates that it has a performance bug rather than a termination bug.

Analysis of the program in Listing 2: The program aims to determine whether a given (target) string is contained within another (source) string. If the target string is found in the source string, the program sets the variable `found` to the index of the target string's first character. But there is a significant performance flaw in the program: when the target string is at the start of the source string, the run is fast, and the program stops almost instantaneously. On the other hand, the run is slower and takes longer to finish when the target string is closer to the end of the source string. This is mostly because there will be a significant increase in the number of redundant computations. The fault is that the initialization statement of the control variable `i` of the for loop at line 6 should be placed outside the scope of the main while loop just after the initialization of the variable `found`. The longest run that we reported occurs when the source string has a length of 10^7

characters, and the target is a single character that is present at the end of the source string. In this instance, the program runs for 30 h before terminating and producing correct results. This is one of the test scenarios that failed (i.e., it leads to an extremely slow run) and was provided to the APR tools for use in repairing the bug. By changing the size of the source string and the positioning of the target string w.r.t. the source string, it is easy to create numerous passing and failing test cases for this program.

Results from the GenProg hybrid: The first simple performance bug is easily fixed by the GenProg hybrid by producing a patch that removes the sleep statement and creates a loop program that is more efficient. Intuitively, the original program and the patched version are semantically equivalent, and thus no formal validation procedure is required. The second performance bug was fixed by the GenProg hybrid by swapping the initialization statements of the variables `found` and `i` at lines 1 and 6. To avoid doing repetitive calculations in the original inefficient loop, the initialization statement of the variable `i` is moved outside of the for loop. In this case, the generated patch passes the test-cases since the variable `i` is no longer being set to 0 every time the loop receives a new character. Although this patch does not appear very elegant from a programming perspective, it is a valid patch that turns slow runs into fast runs without adversely affecting the program's functionality. A fix that places the assignment statement for the variable `i` right after the assignment statement for the variable `found` in the original program is considered to be more elegant. It is interesting to mention that before identifying the patch that passes all of the provided failing test cases, the GenProg hybrid investigated about 3000 candidate patches and spent a total of almost three hours searching the patch space. We set the timer in the generated failed tests to 7 s to allow for the fact that some inputs may take a little longer to complete, especially if the size of the source string is quite huge. The generated patches were assessed in relation to a collection of test cases that included both the anticipated behavior and termination time. The results obtained by the GenProg hybrid show that addressing performance bugs is possible using mutation-based hybrid repair tools, especially with their remove, move, and swap operators. This is primarily due to the observation that we deal with *semantically correct programs* that generate results consistent with the intended behavior. It is possible to avoid repetitive computations by deleting specific statements that unnecessarily increase the computations of the program or restructuring it by moving or swapping specific statements.

Results from the FAngelix hybrid: Using the same set of passing and failing tests that were given to the GenProg hybrid, we also ran the FAngelix hybrid on the two performance bugs previously mentioned. Unfortunately, the FAngelix hybrid was unable to produce patches for both bugs. The fact that none of the termination bugs nor any of the performance bugs could be satisfactorily addressed indicates that the current implementation of FAngelix is unable to handle these classes of bugs. Investigating the causes of its ineffectiveness in handling performance and termination bugs lead us to the following observations:

1. In relation to the two considered performance bugs, FAngelix was unable to locate a suspicious statement. Recall that an APR tool does not generate a patch if no suspicious statements are discovered. This raises questions about the capability of the implemented FL methods in handling performance bugs. Furthermore, the tool is unable to effectively handle bugs whose correction requires the synthesis of expressions that satisfy timed properties.
2. The types of buggy statements that FAngelix can handle are constrained. It has the ability to correct bugs in the following types of statements: if statements, assignment statements, conditional statements, and guard statements. The two inefficient loop programs can be repaired without modifying such types of statements. By deleting, moving, or swapping existing lines in the buggy programs, the considered performance bugs can be resolved. Such alterations to the program under repair cannot be made by FAngelix.

Answer to RQ2: Depending on the APR tool selected, our hybrid repair strategy's efficacy for performance bugs varies. Integrating our approach with GenProg can successfully generate valid patches for both of the performance bugs under consideration. This suggests that mutation-based repair methods may be effective in handling simple and complex performance bugs. However, combining the approach with the semantic-based repair tool FAngelix was unable to handle performance bugs, including the simple one involving the sleep statement. The main observed issues were the inability of the tool to identify the expression that leads to the inefficient behavior of the program and that the types of buggy statements that FAngelix can handle are constrained (performance bugs like the ones considered here may fall outside the scope of semantic repair).

7.3. Key findings from the analysis of termination and performance bugs

Compared to other functional bugs, the repair of termination and performance bugs is more challenging due to their distinctive characteristics. These bugs primarily affect the program's execution time and generate little debugging information. For these kinds of bugs, the main issue is defining passing and failing test cases that allow the fault localization (FL) methods to compute a candidate list of suspicious statements and APR tools to create patches for these bugs. Our empirical investigation of the efficacy of hybrid APR against bugs that adversely affect the execution time of the program leads to the following observations:

1. During the analysis, we considered 15 termination bugs and 2 performance bugs. Both the GenProg and FAngelix hybrids were unable to correctly handle termination bugs: all patches produced by the tools were invalid due to the incompleteness of the correctness specification (i.e., available test suite). While the FAngelix hybrid replaces the conditional expression of the loop with one that iterates a zero time, the GenProg hybrid typically fixes termination bugs by inserting a halting statement into the body of the loop. This clearly demonstrates the complexity of fixing termination bugs using current APR tools for two reasons. First, the faulty traces generated by a termination bug is infinite. Second, fixing a termination bug using dynamic APR requires the satisfaction of complex timed property, where most tools are unable to handle timed properties effectively. On the other hand, the GenProg hybrid successfully handled performance bugs while the FAngelix hybrid was unable to produce patches for performance bugs.
2. The current FL methods that are employed by these tools are generally unable to generate a suspicious list for performance bugs because no erroneous behavior will be observed while an inefficient loop is being executed. The absence of incorrect behavior makes it more challenging to locate the faulty statement that correlates to the performance bug that needs to be fixed. The GenProg hybrid mutates the statements of the program one by one in order to create patches for the two performance bugs, but the FAngelix hybrid was unable to detect any suspicious statements, so no patches were generated. Thus, it is necessary to develop new FL techniques for bugs that generate little observab information. One option would be to use dynamic–static FL methods to handle performance bugs.

3. Correcting termination bugs (infinite loops) is considerably harder than correcting performance bugs (inefficient loops), as the analysis of the aforementioned two research questions demonstrates clearly, for the following reasons. First, whereas we solely deal with finite runs in performance bugs, the analysis of termination bugs can involve both finite and infinite runs. Second, whereas for termination bugs, the relevant semantic property might not be available, the validity property of performance bugs is fully known (the original program can be used as a reference program). Third, generating patches to fix termination bugs might be computationally expensive since it may necessitate making repeated calls to numerous termination provers to ensure the termination of the repaired program.
4. Termination provers can be used to distinguish termination bugs (infinite loops) from performance bugs (inefficiently written loops). This is important because we observe several loops that were not constructed efficiently and take a very long time to terminate (i.e., we observe examples of inefficient loops that terminate after around 30 h). In this situation, a developer may incorrectly assume that the program has a termination bug and not a performance bug. By distinguishing between different types of bugs, it is possible to use the appropriate repair technique for the loop program under repair. Termination provers can also generate CEs which can be used to guide the repair process when searching for repairs.

8. Related work

We discuss the related literature on APR, bug classification, use of formal specifications in APR, previous attempts to integrate different analysis techniques, and termination analysis.

Automated Program Repair: Source-based, automated program repair approaches (Monperrus, 2018) can be separated into search-based and semantic-based approaches. *Search-based* approaches such as GenProg (Le Goues et al., 2012), Astor (Martinez and Monperrus, 2016), and SCRepair (Yu et al., 2020) predominantly use failing test cases to identify bugs, and then mutate the source code until the program passes all failing test cases. They do not provide patch correctness guarantees beyond the fact that the provided test cases now pass. Recent work introduced property-based testing to strengthen the validation of candidate repairs and address overfitting (Gissurarson et al., 2022). Nevertheless, these approaches require executing the program-under-repair, first to find the bug, and then to generate and validate candidate repairs. *Semantic-based* approaches like SemFix (Nguyen et al., 2013), Nopol (Xuan et al., 2017), DirectFix (Mechtaev et al., 2015), SPR (Long and Rinard, 2015), Angelix (Mechtaev et al., 2016), and JFIX (Le et al., 2017) infer repair constraints for the buggy program via symbolic execution of the given tests. The completeness of inferred constraints relies on the available test suite. Similarly, Infinitel (Marcote and Monperrus, 2015) uses an SMT solver to synthesize a loop termination condition and then uses test cases for patch validation.

Bug Classification Systems: Several works target bug classification using a wide variety of classification criteria and for different goals. The work of Li et al. (2006) and Tan et al. (2014) introduced a bug classification system based on the cause-impact criteria. They studied bug characteristics of around 2,060 real-world bugs in three representative open-source projects. They concluded that semantic bugs are the dominant root cause of bugs, and memory-related bugs have decreased due to the development of effective detection tools. Many bug tracking systems classify bugs using severity and priority criteria, where severity indicates the seriousness of the bug on the program functionality and priority indicates how soon the bug should be fixed (Serrano and Ciordia, 2005). Cotroneo et al. present a maintenance-oriented bug classification system in which the characteristics of the bug manifestation are studied (Cotroneo et al., 2016). The study identifies the set of failure-exposing conditions under which a bug may occur. Neither of these classification systems considers properties that can be used to analyze the detection power of different bug detection techniques and the conditions under which they can be integrated, which we add with the classification system proposed in this paper. Asadollah et al. (2015) do use bug observability to study the erroneous behavior of a wide variety of concurrency bugs. However, the authors restrict themselves to concurrency bugs and do not study the detection power of different bug detection techniques.

Integrating Bug Detection Techniques: Few attempts have been made to integrate different program analysis techniques to alleviate the impact of the patch overfitting problem. Al-Bataineh et al. used the static detection patterns/rules as the source for formulating formal specifications and discussed the possibility of integrating static and dynamic analysis techniques to improve the overall quality of generated patches (Al-Bataineh et al., 2021a). There also exists a few examples of using information from debugging to aid APR: Facebook's APR tool SapFix takes information generated during the bug detection process and applies various techniques, including templates specific to given bug types, to fix the program (Marginean et al., 2019).

Termination Analysis of Programs: A huge body of work has been published on proving termination of programs based on a variety of techniques, such as abstract interpretation (Berdine et al., 2007; Chawdhary et al., 2008; Tsitovich et al., 2011), bounds analysis (Gulwani et al., 2009a,b), ranking functions (Bradley et al., 2005; Cousot, 2005), recurrence sets (Gupta et al., 2008; Harris et al., 2010), and transition invariants (Kroening et al., 2010; Podelski and Rybalchenko, 2004). Based on these techniques, a number of program termination checkers have been developed in the prior literature including AProVE (Giesl et al., 2014), 2LS (Chen et al., 2015), T2 (Brockschmidt et al., 2016), and ARMC (Podelski and Rybalchenko, 2007). Unfortunately, termination provers have not yet been used to *validate* the generated candidate patches of termination bugs in the previous APR approaches. We strongly believe that integrating APR approaches with contemporary termination checkers would help advance the current state-of-the-art of APR, not only for repairing termination bugs, but also for other classes of concurrency bugs. This is mainly because fixing termination bugs in concurrent programs would ensure the absence of certain classes of concurrency bugs, such as deadlock and livelock bugs. It would also help avoid the patch overfitting problem by generating verified repairs for termination bugs in both sequential and concurrent programs.

9. Concluding remarks

9.1. Conclusions

A significant class of bugs cannot be handled with current APR approaches, and there is a need to study complementary techniques. To stimulate this work, we propose a novel bug classification system based on three key properties: bug observability, bug tractability, and bug reproducibility. This provides a tool to methodologically explore and compare alternative and hybrid APR approaches by (i) analyzing the detection power of different bug detection techniques, (ii) distinguishing APR approaches based on their bug repair capabilities, and (iii) providing a common terminology that helps identify gaps in current APR research. Moreover, it allows analysis of how techniques can be combined to handle challenging classes of bugs. As a demonstrating example, we study termination bugs in sequential and concurrent programs, and present novel *hybrid* algorithms for their repair by integrating termination provers and software model checkers in the APR pipeline. Our analysis shows that such an integration reduces the complexity of the repair algorithms and improves the overall reliability.

As a followup on this more theoretical analysis, we empirically investigate how well such a hybrid approach can repair termination and

performance bugs. To this end, we create hybrids of tools representing different APR approaches with termination provers and execution time monitors. We use a dataset for termination bugs in C code that was originally developed to evaluate the efficacy of termination provers (Shi et al., 2022), and extend it with two performance bugs: one simple synthetic example, while the other one is a real-world Apache flaw that has also been analyzed by other researchers (Song and Lu, 2017). Our findings indicate that the proposed hybrid repair approaches hold promise for handling termination and performance bugs. More specifically, mutation-based hybrid repair tools seem to perform better at fixing performance bugs than semantic-based repair tools. We hypothesize that this comes from the fact that mutation-based hybrids can easily restructure the program using the basic mutation operators like move, swap, delete, and insert. Since programs with performance bugs are semantically correct programs, we observe that these operators can successfully address performance bugs while they are much harder to address using a semantics-based approach. Fixing termination bugs requires the satisfaction of a composite property, which combines a termination property and a semantic property describing the loop's logic. Hybrid APR can only fully fix such termination bugs if the semantic property is available. Unfortunately, very much aligned with other APR studies, we observe that the incompleteness of (this part of) the correctness specification negatively affects the quality of the patches that can be produced.

9.2. Future work

We identify the following as promising avenues for future research:

1. we are in the process of further empirical validation of the ideas described in this work by combining selected APR tools with termination provers and software model checkers mentioned earlier. One concrete direction is to include template-based repair approaches, which with the current state-of-the-art in template-based APR tools would require a curated dataset of termination and performance problems in Java source code (with tests that indicate correct behavior);
2. there is a need for efficient fault localization mechanisms for termination and liveness bugs, since these create infinite traces that cannot be handled with the current spectrum-based fault localization approaches. We currently circumvent this with a timeout mechanism, but more efficient techniques could find a more precise set of suspicious statements;
3. the combination of CounterExample-Guided Inductive Synthesis (CEGIS) (Solar-Lezama et al., 2006) with termination provers and software model checkers could enable efficient patch space exploration without the user guidance that is normally needed for CEGIS;
4. aside from the termination bugs studied in Section 5, various other bugs cannot be handled by APR approaches solely based on dynamic analysis. One particularly interesting class of non-observable bugs for future research are security-related vulnerabilities where sensitive information may be disclosed to unauthorized parties as a result of violations of information flow security (Sabelfeld and Myers, 2003; Smith, 2007).

CRediT authorship contribution statement

Omar I. Al-Bataineh: Conceptualization, Methodology, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing. **Leon Moonen:** Conceptualization, Methodology, Validation, Data curation, Writing – original draft, Writing – review & editing, Resources, Supervision, Project administration, Funding acquisition. **Linas Vidziunas:** Software, Investigation, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data is available on GitHub at https://github.com/secureIT-project/extendingAPR, and archived on Zenodo at https://doi.org/10.5281/zenodo.10397656.

References

Al-Bataineh, O.I., Grishina, A., Moonen, L., 2021a. Towards more reliable automated program repair by integrating static analysis techniques. In: IEEE Int'L Conference on Software Quality, Reliability and Security (QRS). pp. 654–663, doi:10/gp6kq6.

Al-Bataineh, O.I., Moonen, L., 2022. Towards extending the range of bugs that automated program repair can handle. In: IEEE Int'L Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 1–12.

Al-Bataineh, O., Ng, D.J.X., Easwaran, A., 2021b. Monitoring cumulative cost properties. In: Int'L Conference on Formal Methods in Software Engineering (FormaliSE). pp. 19–30, doi:10/gp6kqw.

Alpern, B., Schneider, F.B., 1985. Defining liveness. Inform. Process. Lett. (ISSN: 0020-0190) 21 (4), 181–185, doi:10/d97bw4.

Alpern, B., Schneider, F.B., 1987. Recognizing safety and liveness. Distrib. Comput. (ISSN: 1432-0452) 2 (3), 117–126, doi:10/fmbptq.

Asadollah, S.A., Hansson, H., Sundmark, D., Eldh, S., 2015. Towards classification of concurrency bugs based on observable properties. In: Int'L Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS). pp. 41–47, doi:10/f3nh6c.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Int'L Conference on Software Engineering (ICSE). pp. 712–721, doi:10/gf2h2r.

Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V., 2017. Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayana Kumar, K. (Eds.), Automated Technology for Verification and Analysis. Springer, Cham, ISBN: 978-3-319-68167-2, pp. 201–207, doi:10/gp6kq7.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P., 2001. Systems and Software Verification: Model-Checking Techniques and Tools. Springer, Berlin, Heidelberg, ISBN: 978-3-662-04558-9.

Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P., 2007. Variance analyses from invariance analyses. In: SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). POPL '07, ACM, New York, NY, USA, ISBN: 978-1-59593-575-5, pp. 211–224, doi:10/frk5km.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D., 2010. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM (ISSN: 0001-0782) 53 (2), 66–75, doi:10/bj8r36.

Bradley, A.R., Manna, Z., Sipma, H.B., 2005. Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (Eds.), Int'L Conference on Computer Aided Verification (CAV). Springer, Berlin, Heidelberg, ISBN: 978-3-540-31686-2, pp. 491–504, doi:10/fpjv58.

Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N., 2016. T2: Temporal property verification. In: Chechik, M., Raskin, J.-F. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer, Berlin, Heidelberg, ISBN: 978-3-662-49674-9, pp. 387–393, doi:10/gp6kq9.

Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L., 1992. Symbolic model checking: 10^{20} states and beyond. Inform. and Comput. (ISSN: 0890-5401) 98 (2), 142–170, doi:10/bvrsx5.

Cai, Y., Cao, L., 2016. Fixing deadlocks via lock pre-acquisitions. In: Int'L Conference on Software Engineering (ICSE). ICSE '16, ACM, New York, NY, USA, ISBN: 978-1-4503-3900-1, pp. 1109–1120, doi:10/gp6kq3.

Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H., 2008. Ranking abstractions. In: Drossopoulou, S. (Ed.), European Symposium on Programming Languages and Systems (ESOP). Springer, Berlin, Heidelberg, ISBN: 978-3-540-78739-6, pp. 148–162, doi:10/d4wm5s.

Chen, H.-Y., David, C., Kroening, D., Schrammel, P., Wachter, B., 2015. Synthesising interprocedural bit-precise termination proofs (T). In: Int'L Conference on Automated Software Engineering (ASE). pp. 53–64, doi:10/gp6krb.

Clarke, E., Grumberg, O., Kroening, D., Peled, D., Veith, H., 2018. Model Checking, second ed. In: Cyber Physical Systems Series, MIT Press, ISBN: 978-0-262-03883-6.

Cook, B., Podelski, A., Rybalchenko, A., 2007. Proving thread termination. In: SIGPLAN Conference on Programming Language Design and Implementation (PLDI). PLDI '07, ACM, New York, NY, USA, ISBN: 978-1-59593-633-2, pp. 320–330, doi:10/dp236p.

Cotroneo, D., Trivedib, K., Russoa, S., Pietrantuonoa, R., 2016. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. J. Syst. Softw. 113, 27–43, doi:10/gpd4t5.

Cousot, P., 2005. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In: Cousot, R. (Ed.), Verification, Model Checking, and Abstract Interpretation. Springer, Berlin, Heidelberg, ISBN: 978-3-540-30579-8, pp. 1–24, doi:10/dzfpc6.

David, C., Kesseli, P., Kroening, D., Lewis, M., 2016. Danger invariants. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (Eds.), Formal Methods. In: Lecture Notes in Computer Science, Springer, Cham, ISBN: 978-3-319-48989-6, pp. 182–198, doi:10/gp6kqr.

Dean, D.J., Wang, P., Gu, X., Enck, W., Jin, G., 2015. Automatic server hang bug diagnosis: Feasible reality or pipe dream?. In: IEEE Int'L Conference on Autonomic Computing Hang Bug Diagnosis. pp. 127–132, doi:10/gp6kq5.

Dovgalyuk, P., Fursova, N., Vasiliev, I., Makarov, V., 2017. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). In: ESEC/FSE 2017, ACM, ISBN: 978-1-4503-5105-8, pp. 944–948, doi:10/gp6kqz.

D'Silva, V., Kroening, D., Weissenbacher, G., 2008. A survey of automated techniques for formal software verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (ISSN: 1937-4151) 27 (7), 1165–1178, doi:10/frf7ww.

Fagan, M., 1976. Design and code inspections to reduce errors in program development. IBM Syst. J. (ISSN: 0018-8670) 15 (3), 182–211, doi:10/btfv3v.

Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R., 2014. Proving termination of programs automatically with aprove. In: Demri, S., Kapur, D., Weidenbach, C. (Eds.), Int'L Joint Conference on Automated Reasoning (IJCAR). Springer, Cham, ISBN: 978-3-319-08587-6, pp. 184–191, doi:10/f3ssz2.

Gilb, T., Graham, D., Finzi, S., 1993. Software inspection. Addison-Wesley, Wokingham, England ; Reading, Mass, ISBN: 978-0-201-63181-4.

Gissurarson, M.P., Applis, L., Panichella, A., van Deursen, A., Sands, D., 2022. PROPR: Property-based automatic program repair. In: Int'L Conference on Software Engineering (ICSE). pp. 1768–1780, doi:10/gqhgs7.

Godefroid, P., 1997. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In: Grumberg, O. (Ed.), Int'L Conference on Computer Aided Verification (CAV). Springer, Berlin, Heidelberg, ISBN: 978-3-540-69195-2, pp. 476–479, doi:10/b7ntq3.

Gregg, B., 2019. BPF Performance Tools: Linux System and Application Observability. Addison-Wesley, ISBN: 978-0-13-655482-0.

Gregg, B., 2020. Systems Performance: Enterprise and the Cloud, second ed. In: Addison-Wesley Professional Computing Series, Addison-Wesley, Boston, ISBN: 978-0-13-682015-4.

Gulwani, S., Jain, S., Koskinen, E., 2009a. Control-flow refinement and progress invariants for bound analysis. ACM SIGPLAN Notices (ISSN: 0362-1340) 44 (6), 375–385, doi:10/fxdmzw.

Gulwani, S., Mehra, K.K., Chilimbi, T., 2009b. SPEED: precise and efficient static estimation of program computational complexity. ACM SIGPLAN Notices (ISSN: 0362-1340) 44 (1), 127–139, doi:10/fxgdsx.

Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G., 2008. Proving non-termination. In: SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). POPL '08, ACM, New York, NY, USA, ISBN: 978-1-59593-689-9, pp. 147–158, doi:10/dd76q5.

Gupta, A., Kahlon, V., Qadeer, S., Touili, T., 2018. Model checking concurrent programs. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (Eds.), Handbook of Model Checking. Springer, Cham, ISBN: 978-3-319-10575-8, pp. 573–611, doi:10/jkpx.

Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K., 2010. Alternation for termination. In: Cousot, R., Martel, M. (Eds.), Static Analysis Symposium (SAS). Springer, Berlin, Heidelberg, ISBN: 978-3-642-15769-1, pp. 304–319, doi:10/djthds.

Havelund, K., Pressburger, T., 2000. Model checking JAVA programs using JAVA PathFinder. Int. J. Softw. Tools Technol. Transfer (ISSN: 1433-2779) 2 (4), 366–381, doi:10/d2pmx6.

Hebbal, Y., Laniepce, S., Menaud, J.-M., 2015. Virtual machine introspection: Techniques and applications. In: Int'L Conference on Availability, Reliability and Security Introspection. pp. 676–685, doi:10/gp6kq4.

Heiden, S., Grunske, L., Kehrer, T., Keller, F., Hoorn, A.v., Filieri, A., Lo, D., 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. Softw. - Pract. Exp. (ISSN: 1097-024X) 49 (8), 1197–1224, doi:10/gp6kq8.

Holzmann, G., 1997. The model checker SPIN. IEEE Trans. Softw. Eng. (ISSN: 1939-3520) 23 (5), 279–295, doi:10/d7wqxt.

Jhala, R., Majumdar, R., 2009. Software model checking. ACM Comput. Surv. 41 (4), 1–54, (ISSN: 0360-0300, 1557-7341). doi:10/fd3pxq.

Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S., 2012. Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices (ISSN: 0362-1340) 47 (6), 77–88, doi:10/f372jr.

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Int'L Conference on Software Engineering (ICSE). p. 467, doi:10/bxz64c.

Killian, C., Anderson, J.W., Jhala, R., Vahdat, A., 2007. Life, death, and the critical transition: Finding liveness bugs in systems code.

Kozen, D., 1977. Lower bounds for natural proof systems. In: Annual Symposium on Foundations of Computer Science (SFCS). pp. 254–266, doi:10/dbkc79.

Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M., 2010. Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (Eds.), Computer Aided Verification. Springer, Berlin, Heidelberg, ISBN: 978-3-642-14295-6, pp. 89–103, doi:10/b54tf5.

Ku, K., Hart, T.E., Chechik, M., Lie, D., 2007. A buffer overflow benchmark for software model checkers. In: Int'L Conference on Automated Software Engineering (ASE). ASE '07, ACM, New York, NY, USA, ISBN: 978-1-59593-882-4, pp. 389–392, doi:10/fds27b.

Lamport, L., 1977. Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. (ISSN: 1939-3520) SE-3 (2), 125–143, doi:10/d25dpw.

Le, X.-B.D., Chu, D.-H., Lo, D., Le Goues, C., Visser, W., 2017. JFIX: semantics-based repair of java programs via symbolic PathFinder. In: Int'L Symposium on Software Testing and Analysis (ISSTA). In: ISSTA 2017, ACM, New York, NY, USA, ISBN: 978-1-4503-5076-1, pp. 376–379, doi:10/gp6krg.

Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W., 2015. The ManyBugs and IntroClass benchmarks for automated repair of c programs. IEEE Trans. Softw. Eng. 41 (12), 1236–1256, (ISSN: 0098-5589, 1939-3520). doi:10/gpd4jv.

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2012. GenProg: A generic method for automatic software repair. IEEE Trans. Softw. Eng. (ISSN: 1939-3520) 38 (1), 54–72, doi:10/cfztf3.

Le Goues, C., Pradel, M., Roychoudhury, A., 2019. Automated program repair. Commun. ACM 62 (12), 56–65, (ISSN: 0001-0782, 1557-7317). doi:10/gkgf29.

Li, P., Regehr, J., 2010. T-check: bug finding for sensor networks. In: Int'L Conference on Information Processing in Sensor Networks. IPSN '10, ACM, ISBN: 978-1-60558-988-6, pp. 174–185, doi:10/djrwkg.

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C., 2006. Have things changed now? an empirical study of bug characteristics in modern open source software. In: Workshop on Architectural and System Support for Improving Software Dependability. ACM, New York, NY, USA, ISBN: 978-1-59593-576-2, pp. 25–33, doi:10/cqsw86.

Lin, Y., Kulkarni, S.S., 2014. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In: Int'L Symposium on Software Testing and Analysis (ISSTA). In: ISSTA 2014, ACM, New York, NY, USA, ISBN: 978-1-4503-2645-2, pp. 237–247, doi:10/gp6krh.

Long, F., Rinard, M., 2015. Staged program repair with condition synthesis. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). In: ESEC/FSE 2015, ACM, New York, NY, USA, ISBN: 978-1-4503-3675-8, pp. 166–178, doi:10/gfvmzm.

Manna, Z., Pnueli, A., 1992. The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York, NY, ISBN: 978-1-4612-0931-7.

Marcote, S.R., Monperrus, M., 2015. Automatic repair of infinite loops. doi:10/jb2f.

Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., Scott, A., 2019. SapFix: Automated end-to-end repair at scale. In: Int'L Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 269–278, doi:10/gkgf2c.

Martinez, M., Monperrus, M., 2016. ASTOR: a program repair library for java. In: Int'L Symposium on Software Testing and Analysis (ISSTA). ACM, Saarbrücken Germany, ISBN: 978-1-4503-4390-9, pp. 441–444, doi:10/gndn55.

Mechtaev, S., Yi, J., Roychoudhury, A., 2015. DirectFix: Looking for simple program repairs. In: Int'L Conference on Software Engineering (ICSE), Vol. 1. pp. 448–458, doi:10/gndpcr.

Mechtaev, S., Yi, J., Roychoudhury, A., 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Int'L Conference on Software Engineering (ICSE). pp. 691–701, doi:10/ggsskp.

Miller, B.P., Cooksey, G., Moore, F., 2006. An empirical study of the robustness of macos applications using random testing. In: Int'L Workshop on Random Testing. RT '06, ACM, New York, NY, USA, ISBN: 978-1-59593-457-4, pp. 46–54, doi:10/bw7w9h.

Miller, B.P., Fredriksen, L., So, B., 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM (ISSN: 0001-0782) 33 (12), 32–44, doi:10/fqnt9s.

Monperrus, M., 2018. Automatic software repair: A bibliography. ACM Comput. Surv. (ISSN: 03600300) 51 (1), 1–24, doi:10/ggssbj.

Muntean, P., Monperrus, M., Sun, H., Grossklags, J., Eckert, C., 2021. IntRepair: Informed repairing of integer overflows. IEEE Trans. Softw. Eng. (ISSN: 1939-3520) 47 (10), 2225–2241, doi:10/gh97rm.

Musuvathi, M., Park, D.Y., Chou, A., Engler, D.R., Dill, D.L., 2002. CMC: A pragmatic approach to model checking real code. In: Symposium on Operating Systems Design and Implementation (OSDI). Usenix.

Mytkowicz, T., Sweeney, P.F., Hauswirth, M., Diwan, A., 2008. Observer Effect and Measurement Bias in Performance Analysis. Technical Report CU-CS-1042-08, University of Colorado at Boulder.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 1–32, doi:10/c5xnmd.

Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S., 2013. SemFix: Program repair via semantic analysis. In: Int'L Conference on Software Engineering (ICSE). pp. 772–781, doi:10/gg82z6.

Pnueli, A., 1977. The temporal logic of programs. In: Annual Symposium on Foundations of Computer Science (SFCS). pp. 46–57, doi:10/dn8cpn.

Podelski, A., Rybalchenko, A., 2004. Transition invariants. In: Symposium on Logic in Computer Science. pp. 32–41, doi:10/fbsbdm.

Podelski, A., Rybalchenko, A., 2007. ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (Ed.), Practical Aspects of Declarative Languages. Springer, Berlin, Heidelberg, ISBN: 978-3-540-69611-7, pp. 245–259, doi:10/dmrfmf.

Radu, A., Nadi, S., 2019. A dataset of non-functional bugs. In: Int'L Conference on Mining Software Repositories (MSR). IEEE, Montreal, Quebec, Canada, pp. 399–403, doi:10/gp6kq2.

Sabelfeld, A., Myers, A., 2003. Language-based information-flow security. IEEE J. Sel. Areas Commun. (ISSN: 1558-0008) 21 (1), 5–19, doi:10/dnvjwg.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C., 2018. Lessons from building static analysis tools at google. Commun. ACM 61 (4), 58–66, (ISSN: 0001-0782, 1557-7317). doi:10/ggsshq.

Serrano, N., Ciordia, I., 2005. Bugzilla, ITracker, and other bug trackers. IEEE Softw. (ISSN: 1937-4194) 22 (2), 11–13, doi:10/fbgs99.

Shi, X., Xie, X., Li, Y., Zhang, Y., Chen, S., Li, X., 2022. Large-scale analysis of non-termination bugs in real-world OSS projects. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2022, ACM, New York, NY, USA, ISBN: 978-1-4503-9413-0, pp. 256–268, doi:10/grmn6m.

Smith, G., 2007. Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (Eds.), Malware Detection. Springer, Boston, MA, ISBN: 978-0-387-44599-1, pp. 291–307, doi:10/fnfjff.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V., 2006. Combinatorial sketching for finite programs. In: Int'L Conference on Architectural Support for Programming Languages and Operating Systems. In: ASPLOS XII, ACM, New York, NY, USA, ISBN: 978-1-59593-451-2, pp. 404–415, doi:10/fdsnnt.

Song, L., Lu, S., 2017. Performance diagnosis for inefficient loops. In: Int'L Conference on Software Engineering. ICSE '17, regexpIEEE, Buenos Aires, Argentina, ISBN: 978-1-5386-3868-2, pp. 370–380, doi:10/gq4wj3.

Swanson, E.B., 1976. The dimensions of maintenance. In: Int'L Conference on Software Engineering (ICSE). ICSE '76, IEEE, pp. 492–497.

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2014. Bug characteristics in open source software. Empir. Softw. Eng. (ISSN: 1573-7616) 19 (6), 1665–1705, doi:10/f6m38x.

Thompson, S., Brat, G., 2008. Verification of C++ flight software with the MCP model checker. In: IEEE Aerospace Conference. IEEE, Big Sky, MT, USA, ISBN: 978-1-4244-1487-1, pp. 1–9, doi:10/ds9chc.

Thompson, S.J., Brat, G., Venet, A., 2010. Software model checking of ARINC-653 flight code with MCP. In: NASA Formal Methods Symposium. NASA.

Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D., 2011. Loop summarization and termination analysis. In: Abdulla, P.A., Leino, K.M. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer, Berlin, Heidelberg, ISBN: 978-3-642-19835-9, pp. 81–95, doi:10/cd8sdb.

Wang, X., Guo, Z., Liu, X., Xu, Z., Lin, H., Wang, X., Zhang, Z., 2008. Hang analysis: fighting responsiveness bugs. Oper. Syst. Rev. (ISSN: 0163-5980) 42 (4), 177–190, doi:10/b9chm8.

Xie, X., Xu, B., 2021. Essential Spectrum-based Fault Localization. Springer, Singapore, ISBN: 978-981-336-178-2, doi:10/jb2d.

Xuan, J., Martinez, M., DeMarco, F., Clément, M., Marcote, S.L., Durieux, T., Lea Berre, D., Monperrus, M., 2017. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. (ISSN: 1939-3520) 43 (1), 34–55, doi:10/gm5s3h.

Yi, J., Ismayilzada, E., 2022. Speeding up constraint-based program repair using a search-based technique. Inf. Softw. Technol. (ISSN: 0950-5849) 146, 106865, doi:10/grvb97.

Yu, X.L., Al-Bataineh, O., Lo, D., Roychoudhury, A., 2020. Smart contract repair. ACM Trans. Softw. Eng. Methodol. 29 (4), 1–32, (ISSN: 1049-331X, 1557-7392). doi:10/gpd4hr.

Zhou, J., Silvestro, S., Liu, H., Cai, Y., Liu, T., 2017. UNDEAD: Detecting and preventing deadlocks in production software. In: Int'L Conference on Automated Software Engineering (ASE). pp. 729–740, doi:10/gp6kqx.