Automatic Repair for Network Programs

Lei Shi[1]✉, Yuepeng Wang[2], Rajeev Alur[1], and Boon Thau Loo[1]

[1] University of Pennsylvania, Philadelphia, USA
[2] Simon Fraser University, Burnaby, Canada
{shilei,alur,boonloo}@seas.upenn.edu yuepeng@sfu.ca

Abstract. Debugging imperative network programs is a difficult task for operators as it requires understanding various network modules and complicated data structures. For this purpose, this paper presents an automated technique for repairing network programs with respect to unit tests. Given as input a faulty network program and a set of unit tests, our approach localizes the fault through symbolic reasoning, and synthesizes a patch ensuring that the repaired program passes all unit tests. It applies domain-specific abstraction to simplify network data structures and exploits function summary reuse for modular symbolic analysis. We have implemented the proposed techniques in a tool called NetRep and evaluated it on 10 benchmarks adapted from real-world software-defined network controllers. The evaluation results demonstrate the effectiveness and efficiency of NetRep for repairing network programs.

1 Introduction

Emerging tools for program synthesis and repair facilitate automation of programming tasks in various domains. For example, in the domain of end-user programming, synthesis techniques allow users without any programming experience to generate scripts from examples for extracting, wrangling, and manipulating data in spreadsheets [13,40]. In computer-aided education, repair techniques are capable of providing feedback on programming assignments to novice programmers and help them improve programming skills [49,14]. In software development, synthesis and repair techniques aim to reduce the manual efforts in various tasks, including code completion [43,10], application refactoring [42], program parallelization [8], bug detection [11,41], and patch generation [11,32].

As an emerging domain, Software-Defined Networking (SDN) offers the infrastructure for monitoring network status and managing network resources based on programmable software, replacing traditional specialized hardware in communication devices. Since SDN provides an opportunity to dynamically modify the traffic handling policies on programmable routers, this technology has witnessed growing industrial adoption. However, using SDNs involves many programming tasks that are inevitably susceptible to programmer errors leading to bugs [3,23]. For example, a device with incorrect routing policies could forward a packet to undesired destinations, and a buggy firewall rule may make the entire network system vulnerable to security threats.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 353–372, 2022.
https://doi.org/10.1007/978-3-030-99527-0_19

In the SDN framework, a logically centralized control plane generates rules that are installed into data planes, which in turn decides the routing of packets throughout the network. While network verification is a well-studied field where operators can be hinted on incorrectly installed rules [3,4,22], little prior work has explored the problem of automatically repairing the corresponding bug in the control plane, especially those written in widely used general-purpose languages such as Java or Python. Existing work mostly restricts the target to control plane programs written in domain-specific languages such as Datalog [51,17].

Since networks cannot tolerate even small mistakes, and most network operators are not trained in programming skills, debugging and repair tools in this domain should prioritize accuracy and automation. This means that many existing techniques for general program repair are not suitable to this domain as they trade off accuracy for heuristics for scaling with the size of analyzed programs and number of discovered potential bugs.

Motivated by the demand for automated repair and the limitations of existing techniques, we develop a precise and scalable program repair technique for network programs. Specifically, our repair technique takes as input a network program and a set of unit tests, reveals the program location that causes the test failure, and automatically generates a patch to fix the program. In the setting of SDN, a unit test corresponds to an incorrectly installed routing rule generated by the control plane from a reported packet. Such unit tests can be discovered by a separate network verification procedure [3,4,22].

Our main idea is to use symbolic reasoning using constraints capturing the semantics of the program for accurate repair, and modular analysis to improve the efficiency. We extended the encoding techniques from prior work [21,12] to support object-oriented features in Java. We also developed a new approach to focus the analysis on one function at a time and gradually narrow down the range of faulty statements along with the specification for the expected behavior.

The proposed technique is implemented in an automatic network program repair tool called NetRep. To evaluate NetRep, we adapt 10 benchmarks from real-world faulty network programs in Floodlight that require changing up to 3 lines of code to fix and apply NetRep to repair the benchmarks automatically. The experimental results show that NetRep is able to find a repair that passes all unit tests for faulty programs up to 738 lines of code for 8 benchmarks using 2 or 3 test cases, outperforming a state-of-the-art repair tool for general Java programs. Furthermore, NetRep is efficient in terms of repair time, requiring only an average running time of 744 seconds across all benchmarks.

Contributions. We make the following main contributions in this paper:

- We present an automated program repair technique that aims to help network operators debug and fix network controller programs automatically.
- We describe a bug localization approach based on symbolic execution and constraint solving for programs with imperative object-oriented features such as virtual function calls.
- We propose novel modular analysis techniques to effectively scale up the symbolic reasoning for automatic repair.

```
@network public class MacAddr {
  private long value;
  private MacAddr(long v) { value = v; }
  public static MacAddr NONE = new MacAddr(0);
  public static MacAddr of(long v) {return new MacAddr(v);}
  ... }
public class FirewallRule {
  public MacAddr dl_dst; public boolean any_dl_dst;
  public FirewallRule() {
    dl_dst = MacAddr.NONE; any_dl_dst = true; ... }
  public boolean isSameAs(FirewallRule r) {
    if (... || any_dl_dst != r.any_dl_dst
            || (any_dl_dst == false &&
                dl_dst != r.dl_dst)) {
        return false; }
    return true; }
  ... }
```

Fig. 1: Code snippet about a bug in Floodlight.

```
public boolean test(long mac1, long mac2) {
  FirewallRule r1 = new FirewallRule();
  r1.dl_dst = MacAddr.of(mac1); r1.any_dl_dst = false;
  FirewallRule r2 = new FirewallRule();
  r2.dl_dst = MacAddr.of(mac2); r2.any_dl_dst = false;
  return r1.isSameAs(r2); }
```

Fig. 2: Unit test that reveals the bug in FirewallRule.

– We develop a tool called NetRep based on the proposed techniques and evaluate it using 10 benchmarks adapted from real-world network programs. The evaluation results demonstrate that NetRep is effective for bug localization and able to generate correct patches for realistic network programs.

2 Overview

In this section, we give a high-level overview of our repair techniques and walk through the NetRep tool using an example adapted from the Floodlight SDN controller [9].

Figure 1 shows a simplified code snippet about firewall rules in Floodlight. Specifically, the program consists of two classes – FirewallRule and MacAddr. The FirewallRule class describes rules enforced by the firewall, including information about source and destination mac addresses. The MacAddr class is an auxiliary data structure that stores the raw value of mac addresses [3].

The network program shown in Figure 1 is problematic because the `isSameAs` function compares two mac addresses using the `!=` operator rather than a negation of the `equals` functions. The `!=` operator only compares two objects based on their memory addresses, whereas the intent of the developer is to check if two mac addresses have the same raw value. The bug is revealed by the unit test in Figure 2, then confirmed and fixed by the Floodlight developers [4]. Next, let

[3] A unique 48-bit number that identifies each network device.

[4] https://github.com/floodlight/floodlight/commit/4d528e4bf5f02c59347bb9c0beb1b875ba2c821e

us illustrate how NetRep localizes this bug based on unit tests `test(1, 2) = false` and `test(1, 1) = true` and automatically synthesizes a patch to fix it.

At a high level, NetRep enters a loop that iteratively attempts to find the fault location and synthesize the patch. Since our repair technique works in a modular fashion, NetRep first selects a function F in the program and tries to repair each possible fault location at a time. If NetRep cannot synthesize a patch consistent with the provided unit tests for any potential fault location in F, it backtracks and selects the next function and repeats the same process until all possible functions are checked. We now describe the experience of running NetRep on our illustrative example.

Iteration 1. NetRep selects the constructor of FirewallRule as the target function. Fault localization determines that the fault is located at the `dl_dst = MacAddr.NONE` part of Line 10, because it is related to the equality checking in the unit test. However, it is not the fault location. NetRep tries to synthesize a patch that passes all unit tests to replace this statement, but fails.

Iteration 2. NetRep selects the same function – constructor of FirewallRule, but the fault localization switches to a different statement `any_dl_dst = true` at Line 10. Similar to Iteration 1, the synthesizer cannot generate a correct patch by replacing this statement.

Iteration 3. Since none of the statements in the constructor is the fault location, NetRep now selects a different function: `isSameAs`. The fault localization determines that `any_dl_dst = false` at Line 13 may be the fault location as it may affect the testing results. However, having tried to replace the statement with many other candidate statements, e.g., `r.any_dl_dst = false`, `any_dl_dst = true`, the synthesizer still fails to generate the correct patch.

Last iteration. Finally, after several attempts to localize the fault, NetRep identifies the fault lies in `dl_dst != r.dl_dst` at Line 14, which is indeed the reported bug location. At this time, the synthesizer manages to generate a correct patch `!dl_dst.equals(r.dl_dst)`. Replacing the original condition at Line 14 with this patch results in a program that can pass all the provided test cases, so NetRep has successfully repaired the original faulty program.

3 Preliminaries

In this section, we present the language of network programs and describe a program formalism that is used in the rest of paper. We also define the program repair problem that we want to solve.

3.1 Language of Network Programs

The language of network programs considered in this paper is summarized in Figure 3. A network program consists of a set of classes, where each class has an optional annotation @network to denote that the class can benefit from network domain-specific abstraction.

$$
\begin{array}{ll}
\textit{Prog } \mathcal{P} ::= \mathcal{C}^+ & \textit{Stmt } s ::= l := e \mid \textbf{jmp } (e)\ L \\
\textit{Class } \mathcal{C} ::= \textbf{@network? class } C\ \{a^+\ F^+\} & \quad \mid \textbf{ ret } v \mid x := \textbf{new } C \\
\textit{Func } F ::= \textbf{function } f(x_1, \ldots, x_n)\ (L : s)^+ & \quad \mid x := C.f(v_1, \ldots, v_n) \\
\textit{Expr } e ::= l \mid c \mid op(e_1, \ldots, e_n) & \quad \mid x := y.f(v_1, \ldots, v_n) \\
\textit{LValue } l ::= x \mid x.a \mid x[v] & \textit{Imm } v ::= x \mid c
\end{array}
$$

$$
x, y \in \textbf{Variable} \quad c \in \textbf{Constant} \quad L \in \textbf{LineID}
$$
$$
C \in \textbf{ClassName} \quad f, f_0 \in \textbf{FuncName} \quad a \in \textbf{FieldName}
$$

Fig. 3: Syntax of network programs.

Each class in the program consists of a list of fields and functions. Each function has a name, a parameter list, and a function body. The function body is a list of statements, where each statement is labeled with its line number. Various kinds of statements are included in our language of network programs. Specifically, assign statement $l := e$ assigns expression e to left value l. Conditional jump statement **jmp** (e) L first evaluates predicate e. If the result is true, then the control flow jumps to line L; otherwise, it performs no operation. Note that our language does not have traditional if statements or loop statements, but those statements can be expressed using conditional jumps. [5]

Return statement **ret** v exits the current function with return value v. New statement $x :=$ **new** C creates an object of class C and assigns the object address to variable x. Static call $x := C.f(v_1, \ldots, v_n)$ invokes the static function f in class C with arguments $v_1, \ldots, v_n$ and assigns the return value to variable x. Similarly, virtual call $x := y.f(v_1, \ldots, v_n)$ invokes the virtual function f on *receiver* object y with arguments $v_1, \ldots, v_n$ and assigns the return value to variable x. Different kinds of expressions are supported including constants, variable accesses, field accesses, array accesses, arithmetic operations, and logical operations. Since the semantics of network programs is similar to that of traditional programs written in object-oriented languages, we omit the formal description of semantics.

In addition, we assume each statement in the program is labeled with a globally unique line number, and line numbers are consecutive within a function.

3.2 Problem Statement

We assume a unit test t is written in the form of a pair (I, O), where I is the input and O is the expected output. Given a network program $\mathcal{P}$ and a unit test $t = (I, O)$, we say $\mathcal{P}$ passes the test t if executing $\mathcal{P}$ on input I yields the expected output O, denoted by $[\![\mathcal{P}]\!]_I = O$. Otherwise, if $[\![\mathcal{P}]\!]_I \neq O$, we say $\mathcal{P}$ fails the test t. In general, given a network program $\mathcal{P}$ and a set of unit tests $\mathcal{E}$, program $\mathcal{P}$ is *faulty* modulo $\mathcal{E}$ if there exists a test $t \in \mathcal{E}$ such that $\mathcal{P}$ fails on t.

Now let us turn the attention to the meaning of fault locations and patches.

Definition 1 (Fault location and patch). *Let $\mathcal{P}$ be a program that is faulty modulo tests $\mathcal{E}$. Line L is called the* fault location *of $\mathcal{P}$, if there exists a statement*

[5] Our repair techniques only handle bounded loops. If there are unbounded loops in the network program, we need to perform loop unrolling.

Algorithm 1 Modular Program Repair

```
1: procedure Repair(P, E)
   Input: Program P, examples E
   Output: Repaired program P' or ⊥ to indicate failure
2:     P ← Abstraction(P);
3:     V ← {L ↦ false | L ∈ Lines(P)}; P' ← ⊥;
4:     while P' = ⊥ do
5:         F ← SelectFunction(P, V);
6:         if F = ⊥ then return ⊥;
7:         V, P' ← RepairFunction(P, F, E, V);
8:     return P';
9: procedure RepairFunction(P, F, E, V)
   Input: Program P, function F, examples E, visited map V
   Output: Updated visited map V, repaired program P'
10:    P' ← ⊥;
11:    while P' = ⊥ do
12:        L ← LocalizeFault(P, F, E, V);
13:        if L ≠ ⊥ then
14:            V ← V[L ↦ true];
15:        else
16:            V ← V[L' ↦ true | TransInFunc(L', P, F)];
17:        if L = ⊥ or IsCallStmt(P, L) then return V, ⊥;
18:        P' ← SynthesizePatch(P, E, F, L);
19:    return V, P';
```

s such that replacing line L of $\mathcal{P}$ *with s yields a new program that can pass all tests in* $\mathcal{E}$*. Here, the statement s is called a* patch *to* $\mathcal{P}$*.*

Problem statement. Given a network program $\mathcal{P}$ that is faulty modulo tests $\mathcal{E}$, our goal is to find a fault location L in $\mathcal{P}$ and generate the corresponding patch s, such that for any unit test $t \in \mathcal{E}$, the patched program $\mathcal{P}'$ can always pass the test t.

4 Modular Program Repair

In this section, we present our algorithm for automatically repairing network programs from a set of unit tests.

4.1 Algorithm Overview

The top-level repair algorithm is described in Algorithm 1. The Repair procedure takes as input a faulty network program $\mathcal{P}$ and unit tests $\mathcal{E}$ and produces as output a repaired program $\mathcal{P}'$ or $\perp$ to indicate repair failure.

At a high level, the Repair procedure maintains a visited map $\mathcal{V}$ from line numbers to boolean values, representing whether each line of $\mathcal{P}$ is checked or not.

The Repair procedure first applies the domain-specific abstraction to program $\mathcal{P}$ (Line 2) and initializes the visited map $\mathcal{V}$ by setting every line in $\mathcal{P}$ as not checked (Line 3). Next, it tries to iteratively repair $\mathcal{P}$ in a modular way until it finds a program $\mathcal{P}'$ that is not faulty modulo tests $\mathcal{E}$ (Lines 4 – 8). In particular, the Repair procedure invokes SelectFunction to choose a function F as the target of repair (Line 5). If none of the functions in $\mathcal{P}$ can be repaired, it returns $\perp$ to indicate that the repair procedure failed (Line 6). Otherwise, it invokes the RepairFunction procedure (Line 7) to enter the localization-synthesis loop inside the target function F.

In addition to the program $\mathcal{P}$ and tests $\mathcal{E}$, the RepairFunction procedure takes as input a target function F and the current visited map $\mathcal{V}$. It produces as output the updated version of the visited map $\mathcal{V}$, as well as a repaired program $\mathcal{P}'$ or $\perp$ to indicate that the function F cannot be repaired. As shown in Lines 11 – 18 of Algorithm 1, RepairFunction alternatively invokes sub-procedures LocalizeFault and SynthesizePatch to repair the target function. In particular, the goal of LocalizeFault is to identify a fault location in function F. If LocalizeFault manages to find a fault location L in F, then line L is marked as visited (Line 14). Otherwise, if LocalizeFault returns $\perp$, it means function F and all functions transitively invoked in F are correct or not repairable. In this case, all lines in F and its transitive callees are marked as checked (Line 16). Furthermore, if the identified fault location L corresponds to a statement that invokes F', it means the fault location is inside F'. Thus, RepairFunction directly returns $\perp$ (Line 17) and SelectFunction will choose F' as the target function in the next iteration. On the other hand, the goal of the sub-procedure SynthesizePatch is generating a patch for function F given the fault location L. If SynthesizePatch successfully synthesizes a patch and produces a non-faulty program $\mathcal{P}'$, then the entire procedure succeeds with repaired program $\mathcal{P}'$. Otherwise, RepairFunction backtracks with a new program location and repeat the same process.

In the rest of this section, we explain fault localization, modular analysis, and patch synthesis in more detail.

4.2 Fault Localization

Next, we give a high-level description of our fault localization technique that aims to find the fault location in a given program. This corresponds to the LocalizeFault procedure in Algorithm 1. We will first show how to encode the problem on an entire program, and then explain how the analysis can be made modular to boost the performance.

At a high level, our fault localization technique uses a symbolic approach by reducing the fault localization problem into a constraint solving problem. In particular, we introduce a boolean variable for each line L, denoted by $\mathcal{B}[L]$, and encode the fault localization problem as an SMT formula, such that the value of the variable $\mathcal{B}[L]$ indicates whether line L is correct or not.

Checking faulty programs. To understand how to encode the fault localization problem, let us first explain how to encode the consistency check given a

program $\mathcal{P}$ and a test case $t = (I, O)$. Specifically, the encoded SMT formula $\Phi(t)$ consists of three components:

1. *Semantic constraints.* For each line $L_i : s_i$, we generate a formula $\Phi_i(S, S')$ to describe the semantics of the statement s_i. Specifically, given a state S that holds before statement s_i, $\Phi_i(S, S')$ is valid if S' is the state after executing s_i. There are two parts of the constraint: the memory contents that are changed, and the memory contents that are preserved. For example, in case of an assignment statement, the constraint will claim that 1) the evaluation result of the right value in state S equals to the left value in state S', and 2) all values except for the left value are the same in S and S'.
2. *Control flow integrity constraints.* In order to ensure all traces satisfying the constraint faithfully follow the control flow structure of a given program $\mathcal{P}$, we generate another set of formulae Φ_f. Specifically, we require that any line of code that is executed must have exactly one predecessor and one successor that are executed, and the branch condition in the code must be respected when picking the successor. This guarantees that there is exactly one valid execution trace corresponding to one test case,
3. *Consistency between program and test.* For the provided test case $t = (I, O)$, we also generate formula $\Phi_{in}(S_0, I)$ and $\Phi_{out}(S_n, O)$ to ensure the program behavior is consistent with the test. In particular, $\Phi_{in}(S_0, I)$ binds input I to the initial state S_0 and $\Phi_{out}(S_n, O)$ describes the connection between output O and final state S_n.

The satisfiability of formula $\Phi(t)$ indicates the result of consistency check. If $\Phi(t)$ is satisfiable, the solver generates a feasible execution trace and an assignment of all intermediate states along this trace. In this case, program $\mathcal{P}$ can pass the test t because there exists a valid trace following the control flow and every pair of adjacent states in the trace is consistent with the semantics of the corresponding statement. Otherwise, if $\Phi(t)$ is unsatisfiable, $\mathcal{P}$ fails the test t.

Now to check whether $\mathcal{P}$ against a set of unit tests $\mathcal{E}$, we can conjoin the formula $\Phi(t_j)$ for each unit test $t_j \in \mathcal{E}$ and obtain the conjunction $\Phi = \bigwedge_{t_j \in \mathcal{E}} \Phi(t_j)$. The satisfiability of formula Φ indicates whether $\mathcal{P}$ is faulty modulo tests $\mathcal{E}$ [6].

Methodology of fault localization. Let $\mathcal{P}$ be a faulty program modulo $\mathcal{E}$, we know the corresponding formula Φ for consistency check is unsatisfiable. Suppose the fault location is line L_i, one key insight is that replacing the semantic constraint $\Phi_i(S, S')$ with *true* yields a satisfiable formula. This is because *true* does not enforce any constraint between the pre-state S and post-state S', so a previously invalid trace caused by the bug at L becomes valid now.

Based on this insight, we develop a methodology to find the fault location using symbolic reasoning. Specifically, given a consistency check formula Φ, we can obtain a fault localization formula Φ' by replacing the semantic constraint $\Phi_i(S, S')$ with $\mathcal{B}[L_i] \rightarrow \Phi_i(S, S')$ for every line $L_i, i \in [1, n]$. Here, variable $\mathcal{B}[L_i]$ decides whether or not it turns the semantic constraint of L_i into *true*. Thus, $\mathcal{B}[L_i] = \mathit{false}$ indicates L_i is a fault location.

[6] The encoding is described in more detail in the extended version [46].

One hiccup here is that formula Φ' is always satisfiable and a model of Φ' can simply assign $\mathcal{B}[L_i] = \mathit{false}$ for all L_i. It means all lines in the program are fault locations, which is not useful for fault localization. To address this issue, we can add a cardinality constraint stating there are exactly K variables in map $\mathcal{B}$ that can be assigned to *false*, which forces the constraint solver to find exactly K fault locations in program $\mathcal{P}$.

Modular analysis. The method above can precisely compute a potential fault location. But an obvious shortcoming is it is hard to scale. Encoding a long program involves 1) a large number of semantic constraints, 2) many fault location choices, as well as 3) many intermediate states to be assigned.

Notice that although a program can be arbitrarily long, developers usually follow the design practice that every function is of limited size. Focusing on analyzing one function at a time and recursively search for the final fault location could be way more efficient than solving one NP-hard problem at the entire program's scale.

To facilitate modular analysis of a function, we need to summarize the behavior of its sub-modules (callee functions) and infer external specification from its higher-level module (caller function).

The encoding method introduced above treats one line of code as a constraint on its pre-state and post-state. To summarize the behavior of a callee function, we aim to turn it into a similar constraint on the pre-state and post-state for the calling statement. The inner states of this callee function should be skipped in the encoding. We can compute such summaries of the target function's callees by symbolic execution. We start with a symbolic representation of the pre-state and execute the callee function until it returns, and claim that the output state equals the post-state. In this way, we can entirely eliminate all bug location choices and inner state assignments in the callee function, as well as greatly simplifying the semantic constraint.

There are two ways to infer the specification of target function. The first way is to encode only the calling stack of the target function up until the top-level function, where we can use the test case as the specification. All function calls made by the target's caller and transitive callers that are not in the stack can be replaced by the automatically computed summary. We can also disable all fault location choices except for lines in the target function. Another way is to infer a possible pre-condition and post-condition of the target function. From the perspective of the caller, the target function is a line of code that puts an incorrect constraint on its pre-state and post-state. After the analysis, the constraint solver will infer a feasible pre-state and post-state assuming this incorrect constraint is removed. This assignment can be used as the pre-condition and post-condition, which eliminates the need to encode any caller function. Since the second approach will possibly introduce incompleteness into the analysis, we use it only to infer a specification to synthesize the final patch, and use the first one for every function's analysis.

Domain-specific abstraction. A domain-specific abstraction is essentially a function summary as discussed above. But for those repeatedly used network

classes (identified by the @network annotation), we can pre-define some more succinct abstractions based on domain knowledge to make the analysis easier. The abstraction $\mathcal{A}[F]$ of a function F is an over-approximation of F that is precise enough to characterize the behavior of F.

The abstraction is useful due to two observations. First, source code for network programs may only be partially available due to the use of high-level interface and native implementation. For example, when comparing the equality between two network addresses, the getClass function is frequently used, but its implementation depends on the runtime and is not available. To make the analysis easier, we can instead use the following abstraction for such comparison:

$$\mathcal{A}[\text{equals}] : \lambda x.\ \lambda y.\ (x.dtype = y.dtype \wedge x.value = y.value),$$

where $x.dtype$ denotes the dynamic type of the object x.

Second, network programs have complex operations that are challenging for symbolic reasoning. For instance, bit manipulations are heavily used in network data structures. While bit manipulations can improve the performance of network programs, they present significant challenges for symbolic analysis due to the encoding in the theory of bitvectors. We can give an abstraction equivalent in correctness but simpler in the behavior, e.g., using the identity function instead of a hash code computation.

4.3 Patch Synthesis

The last step of our repair algorithm is to generate a patch to fix the faulty program. This corresponds to the SynthesizePatch procedure in Algorithm 1. It can be reduced to a sketch finishing problem in program synthesis where we replace the existing faulty line with a hole.

Our general idea is to use plain enumerative search with a depth bound in the candidate patch's space, but with two significant optimizations.

First, we reduce the search space with heuristics. On one hand, we only replace the core expression in the faulty statement with a hole to focus on the most expressive part. To be specific, we consider changing the right-hand-side expressions of assignments, conditional expressions of jump statements, return values of return statements, and functions and arguments for function invocations. On the other hand, we use a limited grammar to guide the search. We parameterize all constants, variables, fields, functions, and operators over the sketch and only instantiate constructs that are in scope. For example, given a particular sketch with a hole, we only populate the variable set with all local and global variables that are in scope of the hole. Also, if the hole corresponds to the conditional expression of a if statement, we only add logical operators to the grammar.

Second, we use the local specification to guide the synthesis. Sketch completion is different from synthesizing a complete program in that the specification is defined for the entire program. We have to repeatedly waste time on executing the correct part of the program to verify a candidate patch. We use the technique described in the modular analysis section to generate a pre-condition and post-condition for only the faulty line. In this way, only the generated patch

needs to be executed to verify against the specification, which greatly saves time when the program grows larger.

5 Implementation

We have implemented the proposed repair technique in a tool called NetRep. NetRep leverages the Soot static analysis framework [26] to convert Java programs into Jimple code, which provides a succinct yet expressive set of instructions for analysis. In addition, NetRep utilizes the Rosette tool [48] to perform symbolic reasoning for fault localization and patch synthesis. While our implementation closely follows the algorithm presented in Section 4, we also conduct several optimizations important to improve the performance of NetRep.

Memories for different types. Since the conversion between bitvectors and integers imposes significant overhead on running time, NetRep divides the memory into one part for integers and another for bitvectors. In this design, NetRep automatically selects the memory chunk based on the variable types. The type checking can guarantee that no such conversion will exist.

Stack and heap. In order to reduce the number of memory operations, NetRep also divides the memory into stack and heap. As is standard, stack only stores static data and its layout is deterministic. Therefore, stacks are implemented using fixed-size vectors, and thus can be efficiently accessed for read and write operations. On the other hand, heap stores dynamic data that are usually not known at compile time, such as allocated objects. Since the heap size cannot be determined beforehand, NetRep uses an uninterpreted function $f(x)$ to represent heaps, where x is the address and $f(x)$ is the value stored at x.

String values. Since reasoning over string values is a challenging task and not always necessary for repairing network programs, we simplified the representation of strings with integer values. Specifically, NetRep maps each string literal to a unique integer and represents all string operations (e.g. concatenation) with uninterpreted functions.

Bounded program analysis. In order to improve the repair time, NetRep only performs bounded program analysis for fault localization and patch synthesis. Namely, we unroll loops and inline functions up to K times, where K is a predefined hyper-parameter. In this way, function summaries can be easily and efficiently computed using symbolic execution.

6 Evaluation

To evaluate the proposed techniques, we perform experiments that are designed to answer the following research questions:

RQ1 Is NetRep effective to repair realistic network programs?
RQ2 How efficient are the fault localization and repair techniques in NetRep?
RQ3 How helpful are modular analysis and domain-specific abstraction for repairing network programs?

ID	Module	LOC	# Funcs	# Tests	Succ	Exp	Loc Time (s)	Synth Time (s)	Total Time (s)
1	DHCP	212	17	2	Yes	Yes	40	117	157
2	Load Balancer	336	28	2	No	No	-	-	-
3	Firewall	262	13	2	Yes	Yes	893	197	1090
4	DHCP	431	32	2	Yes	Yes	95	39	134
5	Utility	809	65	2	No	No	-	-	-
6	Routing	605	44	3	Yes	Yes	271	179	450
7	Utility	454	45	2	Yes	Yes	39	46	85
8	Learning Switch	738	34	2	Yes	No	571	595	1166
9	Database	442	17	2	Yes	No	310	2139	2449
10	Link Discovery	671	46	2	Yes	No	268	158	426

Table 1: Experimental results of NetRep.

RQ4 How is NetRep compared to other repair tools for Java programs?

Benchmark collection. To obtain realistic benchmarks, we crawl the commit history of Floodlight [9], a representative open-source SDN controller in Java that supports the OpenFlow protocol and a rich set of network functions. To distinguish commits caused by bug repairs from those generated for non-repair scenarios, we identify commits based on the following criteria: 1) The commit message contains keywords about repairing bugs, e.g., "bug", "error", "fix"; 2) The commit changes no more than three lines of code.

Following these criteria, we have collected 10 commits from the Floodlight repository and adapted them into our benchmarks. Specifically, given a commit in the repository, we take the code before the commit as the faulty network program and the version after the commit as the ground-truth repaired program. The code is post-processed and the parts irrelevant to the bug of interest are removed. We also identify corresponding unit tests and modify them to directly reveal the bug as appropriate. Each benchmark in our evaluation consists of a faulty network program and its corresponding unit tests.

Experimental setup. All experiments are conducted on a computer with 4-core 2.80GHz CPU and 16GB of physical memory, running the Arch Linux Operating system. We use Racket v7.7 as the compiler and runtime system of NetRep and set a time limit of 1 hour for each benchmark.

6.1 Main Results

Our main experimental results are summarized in Table 1. The column labeled "Module" describes the network module to which the benchmark belongs. The next two columns labeled "LOC" and "# Funcs" show the number of lines of source code (in Jimple) and the number of functions, respectively. The "# Tests" column presents the number of unit tests used for fault localization and patch synthesis. Next, the "Succ" and "Exp" columns show whether NetRep can successfully repair the program and if the generated patch is exactly the same as the ground-truth. Since NetRep returns the first fix that can pass all provided test cases, the repaired programs are not necessarily the same as those

expected in the ground-truth. In this case, the table will show a "Yes" in the "Succ" column and a "No" in the "Exp" column. Finally, the last three columns in Table 1 denote the fault localization time, patch synthesis time and the total running time of NetRep.

As shown in Table 1, there is a range of 13 to 65 functions in each benchmark and the average number of functions is 34 across all benchmarks. Each benchmark has 212 – 809 lines of Jimple code, with the average being 496. NetRep succeeds in repairing 8 out of 10 benchmarks. Furthermore, for 5 benchmarks that can be successfully repaired, NetRep is able to generate exactly the same fix as ground-truth. Given that our benchmarks cover programs from a variety of modules of Floodlight, such as DHCP Server, Firewall, etc, we believe that NetRep is effective to repair realistic network programs (RQ1).

We inspected the reason why NetRep fails to repair benchmarks 2 and 5. NetRep is not able to localize the fault in benchmark 2 due to its incomplete support for unbounded data structures with dynamic allocation such as hash map. For Benchmark 5, NetRep is able to localize the fault but not able to synthesize the correct patch. This is because the expected function to be invoked has side effects with another function, which needs some improvements in the specification checking to verify.

Regarding the efficiency, NetRep can repair 8 benchmarks in an average of 744 seconds with only 2 to 3 test cases. The fault localization time ranges from 39 seconds to 893 seconds, with 50% of the benchmarks within five minutes. The patch synthesis time ranges from 39 seconds to 2139 seconds, with 60% of the benchmarks within five minutes. In summary, the evaluation results show that NetRep only takes minutes to localize bugs in a faulty program and synthesize a correct patch based on two to three unit tests (RQ2).

6.2 Ablation Study

To explore the impact of modular analysis and domain-specific abstraction on the proposed repair technique, we develop three variants of NetRep:

- NetRep-NoMod is a variant of NetRep without modular analysis. Specifically, NetRep-NoMod inlines the functions in a given program but still uses abstractions for network data structures for fault localization and patch synthesis.
- NetRep-NoAbs is a variant of NetRep without domain-specific abstraction. In particular, NetRep-NoAbs uses the original concrete implementation of network functions for symbolic reasoning. If the implementation is written in a different language, we manually translate the implementation to Java.
- NetRep-NoModAbs is a variant of NetRep without modular analysis or domain-specific abstraction. NetRep-NoModAbs simply inlines all functions in the faulty program, including those in the network data structures, and performs symbolic analysis for fault localization and patch synthesis.

To understand the impact of modular analysis and domain-specific abstraction, we run all variants on the 10 collected benchmarks. For each variant, we

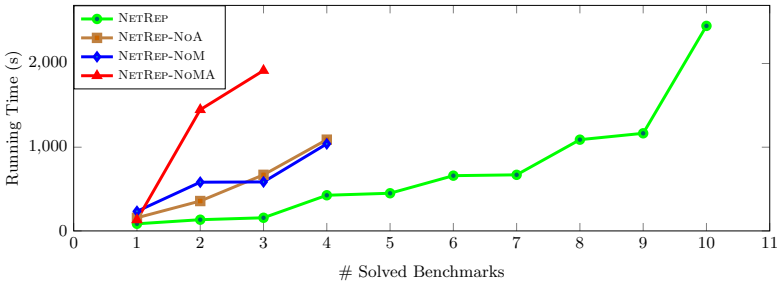

Fig. 4: Comparing NetRep against three variants.

measure the total running time (including time for fault localization and time for patch synthesis) on each benchmark, and order the results by running time in increasing order. The results for all variants are depicted in Figure 4. All lines stop at the last benchmark that the corresponding variant can solve within 1 hour time limit.

As shown in Figure 4, both NetRep-NoAbs and NetRep-NoMod can only solve 4 out of 10 benchmarks in the evaluation, with the average running time being 569 seconds and 610 seconds, respectively. NetRep-NoModAbs solves the least number of benchmarks: 3 out of 10. For the ones that it can solve, the average running time is 1165 seconds. This experiment shows that modular analysis and domain-specific abstraction are both great boost to NetRep's efficiency to repair network programs (RQ3).

6.3 Comparison with the Baseline

To understand how NetRep performs compared to other Java program repair tools, we compare NetRep against a state-of-the-art tool called Jaid [5] on our benchmarks. Specifically, Jaid takes as input a faulty Java program, a set of unit tests, and a function signature for fault localization and patch synthesis, a setting closest to NetRep among a variety of tools. Note that Jaid solves a simpler repair problem than NetRep, because it requires the user to specify a function that is potentially incorrect in the program, whereas NetRep does not need input other than the faulty program and unit tests. In order to run Jaid on our benchmarks, we adjust their formats to fit Jaid's and provide the faulty function (known from the ground truth) as input for Jaid.

Jaid will indefinitely enumerate all possible patches, rather than recommending a most correct one. We think it is successful if the expected patch can be found among the results. In practice, human assistance is needed to pick out this patch from the thousands of candidates.

As a result, Jaid is able to finish on 8 out of 10 benchmarks. The expected patches are found among 2 of them, whereas NetRep can give the expected result for 5 benchmarks on the first recommendation. For one benchmark, Jaid is unable to fix. For another one, it runs out of memory.

We argue that NetRep is better suited for automatically repairing network programs compared to Jaid. First, it only requires network operators to provide

unit test cases. As is discussed above, they can be automatically discovered by another verification or testing procedure. In comparison, Jaid requires users to have skill of programming network controllers to identify the buggy function and pick the correct patch from the results. This is beyond the ability of most network operators and starts to require an expert team. Second NetRep has higher repairing accuracy. As we discussed above, network is sensitive to small mistakes. High accuracy is crucial for a network to function correctly.

In summary, NetRep is more effective in automatically fixing bugs in network programs compared to state-of-the-art repairing tools for Java programs, especially with respect to repairing accuracy and automation (RQ4).

7 Related Work

Automated program repair. Automated program repair is an active research area that aims to automatically fix the mistakes in programs based on specifications of correctness criteria [11,28,39,18], with a variety of applications such as aiding software development [34], finding security vulnerabilities [37], and teaching novice programmers [49,14]. Different techniques have been proposed to solve the automated program repair problem, including heuristics-based techniques [16,31], semantics-based techniques [37,27], and learning-based techniques [45,30,32,47]. NetRep is a semantics-based automated repair tool. Different from prior work, NetRep is specialized to repair network programs based on modular analysis and network data structure abstractions.

Fault localization. Researchers have developed various approaches to fault localization, including spectrum-based, learning-based, and constraint-based techniques. Specifically, the spectrum-based techniques [27,1,2,7,44,6,19] perform fault localization by identifying which part of program is active during a run through execution profiles (called program spectrum). Learning-based techniques [29,53,54] typically train machine learning models to predict and rank possible fault locations. By contrast, constraint-based techniques [21,20,12] encode the semantics of problems as logical constraints and reduce the fault localization problem into constraint satisfaction problem. In spirit, NetRep uses a similar idea for fault localization. However, NetRep performs modular analysis and enables debugging programs involving object-oriented features, whereas prior work only analyzes the entire program in a C-like language. Besides, NetRep reuses the fault localization result to speedup the patch synthesis while prior work mainly focuses on the fault localization step.

Patch synthesis. Many synthesis algorithms have been developed for generating patches, including enumerative search [27], constraint-based techniques [37], statistical model [52], machine learning [15], hint from existing code [25], and so on. In terms of patch synthesis, NetRep generates a context-free grammar from the context of fault locations and performs enumerative search based on the grammar to synthesize patches. It does not require machine learning model or statistical information for ranking all possible patches. However, it is conceivable that NetRep will benefit from the guidance of such ranking techniques.

Verification and synthesis for SDN. In the networking domain, several verification tools [3,33,23,24] have been proposed based on either model checking or theorem proving. For example, VeriCon [3] performs deductive verification to verify the correctness of SDN programs specified by network-wide invariants on all admissible topologies. In addition to verification, synthesis techniques [36,35,38] have also been proposed to aid software-defined networking. NetRep aims to repair network programs automatically, which is a different problem than SDN verification or synthesis.

Repair for network programs. Our work is most related to automated repair of network programs in the SDN domain [50,51,17]. Prior work about autorepair [50,51] relies on using Datalog to capture the operational semantics of the target language to be repaired. The repair techniques work for domain-specific languages (e.g. Datalog or Ruby on Rails) with simple structure. Similarly, Hojjat et al. [17] propose a framework based on horn clause repair problem to help network operators fix faulty configurations. However, NetRep targets Java network programs with object-oriented features and more complex constructs, which cannot be handled by existing techniques.

8 Limitations and Future Work

We discuss several limitations of NetRep that we plan to improve in future work. First, NetRep repairs the faulty network program with the first correct patch that can pass all tests. A user interaction that resumes the synthesis can be introduced in case it is not intended by the user or a more formal specification.

Second, patches that require complicated changes, e.g., those involving control flow structures, are beyond NetRep's ability. They make up 44% of our collection of bug-fixing commits. We envision that the challenge can be addressed by introducing more sophisticated patch synthesis techniques such as searching over a domain-specific language for edits.

Third, in order to force symbolic execution to terminate in finite time, NetRep currently unrolls all loops in the network program, which may result in missing a potential bug. Loop invariant inference techniques can be leveraged to overcome this challenge and still guarantee termination.

9 Conclusion

In this paper, we have proposed an automated repair technique for network controller programs with unit tests as specifications. Our technique internally performs symbolic reasoning for bug localization and patch synthesis, optimized by network domain-specific abstractions and modular analysis to reduce encoding size. we have implemented a tool called NetRep and evaluated it on 10 benchmarks adapted from the Floodlight framework. The experimental results demonstrate that NetRep is effective for repairing realistic network programs with moderate change sizes.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 88–99. IEEE Computer Society (2009)
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. pp. 89–98 (2007)
3. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: towards verifying controller programs in software-defined networks. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 282–293. ACM (2014)
4. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. pp. 155–168 (2017)
5. Chen, L., Pei, Y., Furia, C.A.: Contract-based program repair without the contracts. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 637–647. IEEE Computer Society (2017)
6. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.A.: Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN). pp. 595–604. IEEE Computer Society (2002)
7. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight defect localization for java. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 3586, pp. 528–550. Springer (2005)
8. Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 572–585. ACM (2017)
9. Floodlight: https://github.com/floodlight/floodlight (2021)
10. Galenson, J., Reames, P., Bodík, R., Hartmann, B., Sen, K.: Codehint: dynamic and interactive synthesis of code snippets. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) Proceedings of the International Conference on Software Engineering (ICSE). pp. 653–663. ACM (2014)
11. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)
12. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to c. In: International Conference on Computer Aided Verification. pp. 358–371. Springer (2006)
13. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Ball, T., Sagiv, M. (eds.) Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 317–330. ACM (2011)
14. Gulwani, S., Radicek, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 465–480. ACM (2018)
15. Gupta, R., Pal, S., Kanade, A., Shevade, S.K.: Deepfix: Fixing common C language errors by deep learning. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of

the Thirty-First AAAI Conference on Artificial Intelligence. pp. 1345–1351. AAAI Press (2017)
16. Harman, M.: Automated patching techniques: the fix is in: technical perspective. Commun. ACM **53**(5), 108 (2010)
17. Hojjat, H., Rümmer, P., McClurg, J., Cerný, P., Foster, N.: Optimizing horn solvers for network repair. In: Piskac, R., Talupur, M. (eds.) Proceedings of the Formal Methods in Computer-Aided Design (FMCAD). pp. 73–80. IEEE (2016)
18. Hong, S., Lee, J., Lee, J., Oh, H.: SAVER: scalable, precise, and safe memory-error repair. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 271–283. ACM (2020)
19. Jones, J.A., Harrold, M.J., Stasko, J.T.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA. pp. 467–477. ACM (2002)
20. Jose, M., Majumdar, R.: Bug-assist: Assisting fault localization in ANSI-C programs. In: Proceedings of International Conference on Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 504–509. Springer (2011)
21. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 437–446. ACM (2011)
22. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). pp. 113–126 (2012)
23. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: Veriflow: Verifying network-wide invariants in real time. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 15–27. USENIX Association (2013)
24. Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., Clark, R.J.: Kinetic: Verifiable dynamic network control. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 59–72. USENIX Association (2015)
25. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: International Conference on Computer Aided Verification. pp. 217–233. Springer (2015)
26. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop. vol. 15 (2011)
27. Le, X.D., Chu, D., Lo, D., Goues, C.L., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the Joint Meeting on Foundations of Software Engineering, (ESEC/FSE). pp. 593–604. ACM (2017)
28. Li, G., Liu, H., Chen, X., Gunawi, H.S., Lu, S.: Dfix: automatically fixing timing bugs in distributed systems. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 994–1009. ACM (2019)
29. Li, X., Li, W., Zhang, Y., Zhang, L.: Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In: Zhang, D., Møller, A. (eds.) Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). pp. 169–180. ACM (2019)
30. Li, Y., Wang, S., Nguyen, T.N.: Dlfix: context-based code transformation learning for automated program repair. In: Proceedings of International Conference on Software Engineering (ICSE). pp. 602–614. ACM (2020)

31. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE). pp. 166–178. ACM (2015)
32. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). pp. 298–312. ACM (2016)
33. Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 499–512. USENIX Association (2015)
34. Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., Scott, A.: Sapfix: automated end-to-end repair at scale. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP). pp. 269–278. IEEE / ACM (2019)
35. McClurg, J., Hojjat, H., Cerný, P.: Synchronization synthesis for network programs. In: Majumdar, R., Kuncak, V. (eds.) Proceedings of the International conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10427, pp. 301–321. Springer (2017)
36. McClurg, J., Hojjat, H., Cerný, P., Foster, N.: Efficient synthesis of network updates. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 196–207. ACM (2015)
37. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 691–701. ACM (2016)
38. Padon, O., Immerman, N., Karbyshev, A., Lahav, O., Sagiv, M., Shoham, S.: Decentralizing SDN policies. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 663–676. ACM (2015)
39. Perry, D.M., Kim, D., Samanta, R., Zhang, X.: Semcluster: clustering of imperative programming assignments based on quantitative semantic features. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 860–873. ACM (2019)
40. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. In: Aldrich, J., Eugster, P. (eds.) Proceedings of theACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA). pp. 107–126. ACM (2015)
41. Pradel, M., Sen, K.: Deepbugs: a learning approach to name-based bug detection. Proc. ACM Program. Lang. **2**(OOPSLA), 147:1–147:25 (2018)
42. Raychev, V., Schäfer, M., Sridharan, M., Vechev, M.T.: Refactoring with synthesis. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA). pp. 339–354. ACM (2013)
43. Raychev, V., Vechev, M.T., Yahav, E.: Code completion with statistical language models. In: O'Boyle, M.F.P., Pingali, K. (eds.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 419–428. ACM (2014)
44. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the IEEE International Conference on Automated Software Engineering (ASE). pp. 30–39. IEEE Computer Society (2003)

45. Sakkas, G., Endres, M., Cosman, B., Weimer, W., Jhala, R.: Type error feedback via analytic program repair. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI). pp. 16–30. ACM (2020)
46. Shi, L., Wang, Y., Alur, R., Loo, B.T.: NetRep: Automatic repair for network programs. https://arxiv.org/abs/2110.06303 (2021)
47. Sidiroglou-Douskos, S., Lahtinen, E., Long, F., Rinard, M.: Automatic error elimination by horizontal code transfer across multiple applications. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 43–54. ACM (2015)
48. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 530–541. ACM (2014)
49. Wang, K., Singh, R., Su, Z.: Search, align, and repair: data-driven feedback generation for introductory programming exercises. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 481–495. ACM (2018)
50. Wu, Y., Chen, A., Haeberlen, A., Zhou, W., Loo, B.T.: Automated network repair with meta provenance. In: Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets). pp. 26:1–26:7. ACM (2015)
51. Wu, Y., Chen, A., Haeberlen, A., Zhou, W., Loo, B.T.: Automated bug removal for software-defined networks. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 719–733. USENIX Association (2017)
52. Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 416–426. IEEE / ACM (2017)
53. Xuan, J., Monperrus, M.: Learning to combine multiple ranking metrics for fault localization. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 191–200. IEEE Computer Society (2014)
54. Zhang, Z., Lei, Y., Tan, Q., Mao, X., Zeng, P., Chang, X.: Deep learning-based fault localization with contextual information. IEICE Trans. Inf. Syst. **100-D**(12), 3027–3031 (2017)