Debug-localize-repair: a symbiotic construction for heap manipulations

Sahil Verma1 · Subhajit Roy2

Received: 17 April 2020 / Accepted: 8 December 2021 / Published online: 8 February 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract
We present WOLVERINE2, an integrated Debug-Localize-Repair environment for heap manipulating programs. WOLVERINE2 provides an interactive debugging environment: while concretely executing a program via an interactive shell supporting common debugging facilities, WOLVERINE2 displays the abstract program states (as box-and-arrow diagrams) as a visual aid to the programmer, packages a novel, proof-directed repair algorithm to quickly synthesize the repair patches and a new bug localization algorithm to reduce the search space of repairs. WOLVERINE2 supports “hot-patching” of the generated patches to provide a seamless debugging environment, and also facilitates new debug-localize-repair possibilities: specification refinement and checkpoint-based hopping. We evaluate WOLVERINE2 on 6400 buggy programs (generated using automated fault injection) on a variety of data-structures like singly, doubly, and circular linked lists, AVL trees, Red-Black trees, Splay Trees and Binary Search Trees; WOLVERINE2 could repair all the buggy instances within realistic programmer wait-time (less than 5 s in most cases). WOLVERINE2 could also repair more than 80% of the 247 (buggy) student submissions where a reasonable attempt was made.

Keywords Program repair · Bug localization · Program debugging · Heap manipulations

1 Introduction

Hunting for bugs in a heap manipulating program is a hard proposition. We present WOLVERINE2, an integrated debugging-localize-repair tool for heap-manipulating programs. WOLVERINE2 uses gdb [17] to control the concrete execution of the buggy program to provide a live visualization of the program (abstract) states as box-and-arrow diagrams. Programmers
routinely use such box-and-arrow diagrams to plan heap manipulations and in online education [25].

Similar to popular debugging tools, WOLVERINE2 packages common debugging facilities like stepping through an execution, setting breakpoints, fast-forwarding to a breakpoint (see Table 1). At the same time, WOLVERINE2 provides additional commands for driving in situ repair: whenever the programmer detects an unexpected program state or control-flow (indicating a buggy execution), she can repair the box-and-arrow diagram to the expected state or force the expected control-flow (like forcing another execution of a while loop though the loop-exit condition is satisfied) during the debugging session. These expectations from the programmer are captured by WOLVERINE2 as constraints to build a (partial) specification.

When the programmer feels that she has communicated enough constraints to the tool, she can issue a repair command, requesting WOLVERINE2 to attempt an automated repair. WOLVERINE2 is capable of simulating hot-patching of the repair patch (generated by its repair module), allowing the debugging session to continue from the same point without requiring the user to abort the debug session, recompile the program with the new repair patch and start debugging. As the repair patch is guaranteed to have met all the user expectations till this point, the programmer can seamlessly continue the debugging session from the same program point, with the repair-patch applied, without requiring an abort-compile-debug cycle. This debug-repair scheme requires the user to point out the faults in the program states, while WOLVERINE2 takes care of correcting (repairing) the fault in the underlying program.

WOLVERINE2 enables a seamless integration of debugging, fault-localization and repair (debug-localize-repair), thereby facilitates novel debug strategies wherein a skilled developer can drive faster repairs by communicating her domain knowledge to WOLVERINE2: if the programmer has confidence that a set of statements cannot have a bug, she can use specification refinement to eliminate these statements from the repair search space. Hence, rather than eliminating human expertise, WOLVERINE2 allows a synergistic human-machine interaction. Additionally, WOLVERINE2 allows for a new repair-space exploration strategy, that we refer to as checkpoint-based hopping, wherein the developer can explore multiple strategies of fixing the program simultaneously, examine the repairs along each direction, and switch between the different candidate fixes seamlessly—to converge to the final fix.

WOLVERINE2 bundles a novel proof-directed repair strategy: it generates a repair constraint that underapproximates the potential repair search space (via additional underapproximation constraints). If the repair constraint is satisfiable, a repair patch is generated. If proof of unsatisfiability is found (indicating a failed repair attempt) that does not depend on

---

1 We thank the anonymous reviewers of the preliminary conference version of this paper for suggesting this feature.
an underapproximation constraint, it indicates a buggy specification or a structural limitation in the tool’s settings; else, the respective underapproximation constraint that appears in the proof indicates the widening direction.

To further improve the scalability of repair, we also design an inexpensive bug localization technique that identifies suspicious statements by tracking the difference in the states in the forward execution (proceeding from the precondition to the postcondition) and an (abstract) backward execution (commencing from the postcondition to the precondition). The algorithm leverages on an insightful result that buggy statements always appear at program locations that exhibit a non-zero gradient on the state differences between the forward and backward execution. We prove that our algorithm is sound, i.e., it overapproximates the set of faulty statements, thereby shrinking the repair space appreciably without missing out on the ground truth bug. Our experiments show that this algorithm can shrink the suspicious statements to less than 12% of the program size in 90% of our benchmarks and works better than popular statistical bug localization techniques.

We evaluate Wolverine2 on a set of 6400 buggy files: 40 randomly generated faulty versions over four faulty configurations of 40 benchmark programs collected from online sources [15] spanning multiple data-structures like singly, doubly and circular linked lists, Binary Search Trees, AVL trees, Red-Black trees, and Splay trees. We classify the 40 programs into two categories: smaller (20 programs) and larger (20 programs) based on the program size. Wolverine2 successfully repairs all faults in the benchmarks within a reasonable time (less than 5 s for most programs). To evaluate the effectiveness of our bug localization algorithm, we switch off bug localization before repair: Wolverine2 slows down by more than 225 × without bug localization on our larger benchmarks and fails to repair 1262 programs (out of 6400) within a timeout of 300s.

We also evaluate Wolverine2 on 247 student submissions from an introductory programming course [10], consisting of problems for five heap manipulating problems; Wolverine2 could repair more than 80% of the programs where the student had made a reasonable attempt.

We make the following contributions in this paper:

- We propose that an integrated debug-localize-repair environment can yield significant benefits; we demonstrate it by building a tool, Wolverine2, to facilitate debug-localize-repair on heap manipulations;
- We propose a new proof-directed repair strategy that uses the proof of unsatisfiability to guide the repair along the most promising direction;
- We propose advanced debugging techniques, specification refinement and checkpoint-based hopping, that are facilitated by this integration of debugging and repair;
- We design a new fault localization algorithm for heap manipulating programs based on the gradient between the states in a forward and backward execution.

Wolverine2 extends our previous work on Wolverine [65]: Wolverine2 augments the abilities of Wolverine with a new module for bug localization (Sect. 5), which has significantly improved (33-779 ×) its runtime performance, allowing it to solve many instances that were beyond Wolverine. We evaluate Wolverine2 on a larger benchmark set to demonstrate the advanced capabilities of the tool (Sect. 7). We have also added new debugging capabilities (Sect. 6.2) (some of which were suggested by the reviewers of the conference version).
2 Overview

2.1 A WOLVERINE2 debug-localize-repair session

We demonstrate a typical debug-localize-repair session on WOLVERINE2: the program in Fig. 1 creates a doubly linked-list (stack) of four nodes using the push() functions, and then, calls the reverse() function to reverse this list. The reverse() function contains three faults:

1. The loop condition is buggy which causes the loop to be iterated for one less time than expected;
2. The programmer (possibly due to a cut-and-paste error from the previous line) sets temp2 to the prev instead of next field;
3. The head pointer has not been set to the new head of the reversed list.

The programmer uses the _start command to launch WOLVERINE2, followed by four _next commands to concretely execute the statements in push() functions, creating the doubly-linked list. Figure 2a shows the current (symbolic) state of the program heap, that is displayed to the programmer.

(Wolverine2) start
Starting program...
push(2)
(Wolverine2) next; next; next; next;
push(4);
...

The programmer, then, uses the _step command to step into the reverse() function.

reverse();
(Wolverine2) step
current = head;
(Wolverine2) next

The programmer deems the currently displayed state as desirable as this program point and decides to assert it via the _spec command. The asserted states are registered as part of the specification, and the repair module ensures that any synthesized program repair does exhibit this program state at this program location.

while(temp1 != NULL)
(Wolverine2) spec
Program states added

_Bug1_ prevents the execution from entering the while-loop body, the programmer therefore employs _enter_ command to force the execution inside the loop.

while(temp1 != NULL)
(Wolverine2) enter

The programmer issues multiple _next_ commands to reach the end of this loop iteration.

temp1 = current->prev;
(Wolverine2) next; next; next; next;
while(temp1 != NULL)

The program state at this point (Fig. 2b) seems undesirable as current and prev field of node n4 point to null (instead of pointing to n3). The programmer _corrects the program state_ by bringing about these changes via the _change_ command.

(Wolverine2) change current n3
struct node *head;
void reverse()
{
    struct node *current, *temp1 = NULL, *temp2 = NULL;
    current = head;
    while (temp1 != NULL) // FIX1: current != NULL
    {
        temp1 = current->prev;
        temp2 = current->prev; // FIX2: current->next
        current->prev = temp2;
        current->next = temp1;
        current = current->prev;
    } // head = temp1->prev; // FIX3: insert stmt.
}
...
int main()
{
push(2); push(4); push(8); push(10);
reverse();
}

Fig. 1 Our motivating example

(a) Nodes added (before entering reverse)

(b) At the end of first loop iteration (without user changes)

(c) Changes brought about by the user after first loop iteration

(d) At the end of second loop iteration (after user changes)

(e) At the end of third loop iteration (no changes required)

(f) At the end of fourth loop iteration

Fig. 2 Visualization of the program execution provided by WOLVERINE2

(Wolverine2) change n4 -> prev n3

Figure 2c shows the updated program state, and the programmer commits them to specification.

(Wolverine2) spec
Program states added
The execution is now forced in the loop for the second time, again using the `enter` command.

```
(Wolverine2) enter
while(temp1 != NULL) ...
```

The state at the end of the second iteration is not correct; the programmer performs the necessary changes and commits it to the specification.

```
while(temp1 != NULL)
(Wolverine2) change current n2
(Wolverine2) change n3 -> prev n2
(Wolverine2) spec
Program states added
```

She then uses the `repair` command to request a repair patch.

```
(Wolverine2) repair
Repair synthesized...
```

To repair the program, WOLVERINE2 first launches its bug localization module that searches for potentially faulty statements; in this case, it identifies the second, third, and fifth statements (lines 7, 6, and 10) in the “while” loop (which is the statement with Bug2) as suspicious candidates.

The repair module, then, searches for possible mutations of the potentially faulty statements (identified by the bug localizer) to synthesize a repair patch that is guaranteed to satisfy the given specifications committed thus far.

In the present case, the repair synthesized by WOLVERINE2 correctly fixes Bug2; however, the other bugs remain as the trace has not encountered these faults yet. WOLVERINE2, further, simulates hot-patching of this repair, allowing the user to continue this debugging session rather than having to abort this debug session, recompile, and restart debugging.

To check the generality of the repair, the programmer steps through the third loop iteration to confirm that it does not require a state change, alluding to the fact that the repair patch is possibly correct.

```
while(temp1 != NULL)
(Wolverine2) enter ...
```

The fourth iteration also updates the program heap as per the programmer’s expectations, reinforcing her confidence in the repair patch.

Due to Bug1, the loop termination condition does not hold even after the complete list has reversed; the programmer, thus, forces a change in the control flow via the `leave` command to force the loop exit.

```
while(temp1 != NULL)
(Wolverine2) leave
Exiting function...
```

At this point, the programmer notices that the state is faulty as the head pointer continues to point to the node n4 rather than n1, the new head of the reversed list (Fig. 2f).

The programmer adds this change to the specification and requests another repair patch.

```
(Wolverine2) change head n1
(Wolverine2) spec
Program states added
```
This repair requires the insertion of a new statement; WOLVERINE2 is capable of synthesizing a bounded number of additional statements to the subject program. On our machine, the first repair call takes 0.5 s (fixing Bug2) while the second repair call returns in 0.3 s (fixing Bug1 and Bug3).

To summarize, the debug session builds a correctness specification via corrections to the program state, that WOLVERINE2 uses to drive automated repair, aided by fault localization to prune the repair space.

2.2 The claws of WOLVERINE2

The high-level architecture of WOLVERINE2 is shown in Fig. 3. The Driver module is the heart of the tool, providing the user shell and coordinating between other modules.

After receiving a C program, WOLVERINE2 employs the C-to-IR generator to compile it into its intermediate representation (IR) as a sequence of guarded statements ($\Gamma$) and a location map ($\Lambda$) to map each line of the C-source code to an IR instruction (see Sect. 3). Each C-source code instruction can potentially be mapped to multiple IR instructions. For the sake of simplicity, we assume that each C-source code line appears in a new line. Note that each C-code instruction can get compiled down to multiple IR instructions.

The Driver module initiates the debug session by loading the binary on gdb: many of the commands issued by the programmer are handled by dispatching a sequence of commands to gdb to accomplish the task. However, any progress of the program’s execution (for example, the next command from the programmer) is routed via the instruction screening module that manages specification refinement and simulates hot-patching (see Sect. 3 and Algorithm 2).

On the repair command, the driver invokes the repair module to request an automated repair based on the specification collected thus far. The repair module, in turn, invokes the fault localization engine, to identify a set of suspicious locations. The fault localization algorithm is sound but not complete—though it may return multiple suspicious statements (including ones that are not faulty), the set of these suspicious locations is guaranteed to contain the buggy location. The repair module restricts its mutations within the set of suspicious statements to synthesize a repair patch. This patch is propagated to the instruction screening module to enable hot-patching, enabling the user to continue as if she was executing this transformed program all along. If satisfied, she invokes the rewrite module to translate the intermediate representation of the repaired program to a C language program.
1 if (head != NULL) {
2 temp = head->next;
3 }
4 if (head != NULL || temp != NULL) {
5 return;
6 }
7 last = head;
8 temp = last->next;

(a) An example program in C

(b) Intermediate representation of the program

Fig. 4 Our intermediate representation

3 Heap debugging

The state of a program (S) contains a set of variables \( \sigma_V \) and a set of heap nodes \( \sigma_H \) with fields \( \sigma_F \) as \( V \times H \); the state of the program variables, \( V \), is a map \( \sigma_V \to D \) and the program heap (\( H \)) is a map \( \sigma_H \times \sigma_F \to D \). The domain of possible values, \( D \), is \( I \cup \sigma_H \) where \( I \) is the set of integers. For simplicity, we constrain the discussions in this paper to only two data-types: integers and pointers. We use the function \( \Upsilon(e) \) to fetch the type of a program entity; a program entity \( e \in E \) is either a variable \( v \in \sigma_V \) or a field of a heap node \( h \in \sigma_H \times \sigma_F \).

Also, pointers can only point to heap nodes as we do not allow taking reference to variables.

Memory state witnessed by concrete execution via \texttt{gdb} is referred to as the concrete state, from which we extract the symbolic state as a memory graph \cite{72}, where machine addresses are assigned symbolic names. For our symbolic state, pointers are maintained in symbolic form, whereas scalar values (like integers) are maintained in concrete form. In the concrete state, all entities are maintained in their concrete states.

3.1 Symbolic encoding of an execution

We use Hoare triples \cite{27} to show the semantics of our intermediate representation (Fig. 5). In the intermediate representation, the program is maintained as a sequence of guarded statements, i.e. a statement is executed only if its guard evaluates to \texttt{true} at that program point (rules \texttt{grd1} and \texttt{grd2} in Fig. 5). The scope of our repairs includes modification/insertion of both statements and guards.

Assignment \((x := y)\), getfield \((x := y.f)\) and putfield \((x.f = y)\) are the primary statements in our intermediate representation. For a map \( M \), we use the notation \( M_1 = M_2[a_1 \mapsto a_2] \) to denote that \( M_2 \) inherits all mappings from \( M_1 \) except that the mappings \( a_1 \mapsto a_2 \) is added/updated. For brevity, we skip discussions of other statements like \texttt{print}.

Each statement comes with its set of preconditions that must be satisfied for the update to ensue, for example, the assignment statement requires that the type of the variables match, and a getfield statement requires type matching along with the need for the dereferenced variable to be a pointer and non-null.

We use concrete statement to allow an interesting debugging strategy (referred to as specification refinement, see Sect. 6.1). For such statements, \( \xi \), we extract a concrete precondition, execute \( \xi \) concretely and fetch a concrete postcondition for concrete statement.

The guards are predicates that can involve comparisons from \{\( \leq, <, \geq, >, =, \neq \}\) for integers and only \{\( =, \neq \)\} for pointers (we omit their formal semantics).
The Boolean predicates for the guards can be constructed using the Boolean assignment operator (\( \triangleright \)) and the Boolean-and operator (\( \wedge \)). The program also allows for non-deterministic assignments (\( \triangleright \)). Figure 4 shows a program with compound conditions and its intermediate representation.

When the repair command is invoked at a particular program point \( p \), by the conjunction of the semantic encoding of the instructions in the trace, the repair module constructs a symbolic model, \( \Phi_{sem} \), of the execution trace up to \( p \) (we assume that the trace has a length of \( n \)).

\[
\Phi_{sem} \equiv \prod_{i=1}^{n} \mathcal{T}_i(S_i, S_{i+1})
\]

\( \mathcal{T}_i \) encodes the semantics of the \( i \)th instruction (Fig. 5) and \( S_i \) (and \( S_{i+1} \)) denote the input (and output) state of this instruction (respectively).

### 3.2 The heap debugger

Algorithm 1 provides a high-level sketch of the operation of Wolverine2. Our algorithm accepts a program as a sequence of guarded statements \( \langle \pi, \omega \rangle \) where \( \pi \) is a guard predicate of the form \( \langle op, arg_1, arg_2 \rangle \), and \( \omega \) can be one of assignment, getfield, putfield, Boolean-or, Boolean-and, Boolean-assign, non-deterministic or concrete statement.

Before the algorithm enters the command loop, \( \Theta \) is initialized to the set of entities in the program; these entities are recorded whenever \( \text{spec} \) is invoked.
Algorithm 1: The Heap Debugger

1. $\Theta = \{ e | e \in \mathcal{H} \}$
2. while true do
3.   cmd := Prompt()
4. switch cmd do
5.   case start
6.     loc = gdb_start()
7.   case next
8.     loc = ExecuteStatement(loc)
9.     $S_c$, $\gamma = $ create_symbolic_state($S_c$, $\gamma$
10.    display_map($S_s$)
11.   case break <loc>
12.     gdb_send(break <loc>)
13.   case change <es> <vs>
14.     $S_c$ = fetch_concrete_state()
15.     $S_s$, $\gamma = $ create_symbolic_state($S_c$, $\gamma$
16.     $S_s[es] := vs$
17.     gdb_set_address($\gamma[es], vs$)
18.   case spec
19.     $S_c$ = fetch_concrete_state()
20.     $S_s$, $\gamma = $ create_symbolic_state($S_c$, $\gamma$
21.     assert_spec($\Lambda[loc].IR_id, S_s \cap \Theta$)
22.   case repair
23.     patch := $\text{repair\_run}$()
24.     hot_patch(patch)
25. endsw
26. end

Algorithm 2: ExecuteStatement

Input: $\Gamma := [\langle \text{stmt.action, stmt.grd, stmt.loc} \rangle,$
$\Lambda := N \rightarrow \langle L, \{\text{changed, inserted, preserved} \} \rangle$

1. if stmt == "concrete" then
2.   loc = gdb_send("next")
3.   $S_c$ = fetch_concrete_state()
4.   $S_s$, $\gamma = $ create_symbolic_state($S_c$, $\gamma$
5.   repair_add_spec($\Gamma[pp], IR\_id, S_s \cap \Theta$)
6. else if $\Lambda[loc].status == \text{changed}$ then
7.   gdb_send("skip")
8.   irstm = IR2gdbStm($\Lambda[loc].IR\_id$)
9.   gdb_exec_stm(irstm)
10. else if $\Lambda[loc].status == \text{inserted}$ then
11.   irstm = IR2gdbStm($\Lambda[loc].IR\_id$)
12.   gdb_exec_stm(irstm)
13. else loc = gdb_send("next") return loc

When _next_ command is invoked, WOLVERINE2 dispatches the next program statement to be executed (at source line loc) to the statement screening module (Algorithm 2), which returns the next C-code source line to be executed. After execution, WOLVERINE2 uses the function fetch_concrete_state() to query gdb for the updated program state, and then generates the memory map [72] (using the create_symbolic_state() function). This function returns the symbolic map $\hat{S}$ and a map $\gamma$; the map $\gamma$ records the mapping of
the concrete entities to their symbolic forms. The symbolic memory map is displayed to the programmer as a visual aid for debugging (see Sect. 1).

The functionality of \texttt{break} command is similar to the default instruction in \texttt{gdb} for inserting a breakpoint.

The \texttt{change} command allows the programmer to convey expectations of the desired program state at a point. The program state can be changed by providing a new value $v_s$ to a symbolic state $e_s$. \textsc{Wolverine2} fetches the relevant concrete state and issues multiple commands (summarized by the function \texttt{gdb-set-address()}) to modify the concrete program state.

The \texttt{spec} command asserts the symbolic state at the current program point.

Finally, the \texttt{repair} command invokes the \textit{repair module}, which attempts to synthesize a repair patch that satisfies the concatenated assertions added thus far. The \textit{repair module} internally calls the fault localizer and, hence, fault localization is not available as an explicit command. If the repair succeeds, the repair patch is passed to the \textit{instruction screening} module, which simulates hot-patching for a seamless debugging session.

In addition to the above, \textsc{Wolverine2} also supports altering of the control flow (like entry/exit of loops via the \texttt{enter} and \texttt{leave} commands respectively), flip branch directions etc. We demonstrated these features in Sect. 1 but we omit the details.

The \texttt{ExecuteStatement()} function, provided by the \textit{instruction screening} module, is described in Algorithm 2; this function accepts a list of guarded statements $\Gamma$ and a map $\Lambda$ from the source line numbers (in $N$) to a tuple containing the corresponding IR instruction(s) (in $L$). Status bits ($F \in \{\text{changed, inserted, preserved}\}$) are used to indicate if: (a) the IR instruction has been modified (changed) by a repair patch, (b) appears as a new instruction (inserted) due to a repair patch, or (c) is unmodified (preserved); this information is required to simulate hot-patching. Deletion of a statement is also marked (changed); the corresponding patch simply sets the guard of the instruction to false. The \textit{instruction screening} module handles two primary tasks:

- \textbf{Handling concrete statements.} Upon encountering a concrete statement, \textsc{Wolverine2} executes it via \texttt{gdb} by issuing the \texttt{next} command. The effect of this concrete execution is then asserted by taking a snapshot of the updated concrete state (again via \texttt{gdb}) and adding the corresponding symbolic state to the specification. We refer to this debugging strategy as specification refinement (see Sect. 6.1).

- \textbf{Simulate hot-patching.} If a repair patch has modified the statement, \textsc{Wolverine2} requests \texttt{gdb} to skip the execution of this statement, and translates the “effect” of the modification into a string of \texttt{gdb} commands (\textit{irstm}) via the \texttt{IR2gdbStm()} function and consigns the command-list to \texttt{gdb} using the \texttt{gdb-exec-stm()} function.

Otherwise, the next statement is concretely executed via \texttt{gdb} by issuing the \texttt{next} command.

### 4 Proof-guided repair

Algorithm 3 shows our repair algorithm: it takes a (buggy) program $P$ as a sequence of guarded statements, a set of \textit{locked locations} $\mathcal{L}$, and a bound on the number of new statements that a repair is allowed to insert (num\_insert\_slots). The repair algorithm attempts to search for a repair candidate $\hat{P}$ (of size $n = |P| + \text{num\_insert\_slots}$) that is “close” to the existing program and satisfies the programmer’s expectations (specification). Our algorithm is allowed to mutate and delete existing statements and insert at most $n$ new statements;
Formal Methods in System Design (2021) 58:399–439

however, mutations are not allowed for the locations contained in $\mathcal{L}$. The insertion slots contain a guard $\text{false}$ to begin with (Line 8); the repair algorithm is allowed to change it to “activate” the statement. Deletion of a statement changes the guard of the statement to $\text{false}$.

Wolverine2 allows for new nodes and temporary variables by providing a bounded number of additional (hidden) nodes/temporaries, made available on demand. The programmer configures the number of insertion slots, but these slots are activated by the repair algorithm only if needed. For loops, we add additional constraints so that all loop iterations encounter the same instructions.

4.1 Primary constraints

We use a set of selector variables $\{r_1, \ldots, r_n, s_1, \ldots, s_n\}$ to enable a repair. Setting a selector variable to $\text{true}$ relaxes the respective statement, allowing Wolverine2 to synthesize a new guard/statement at that program point to satisfy the specification. We define a metric,
closeness($P_1, P_2$), to quantify the distance between two programs by summing up the set of
guards and statements that match at the respective lines. As the insertion slots should
be allowed to be inserted at any point in the program, the closeness metric would have to
be ‘adjusted’ to incorporate this aberration due to insertions. For this purpose, our repair
algorithm also infers a relation $\xi$ that maps the instruction labels in the repair candidate $\hat{P}$
to the instruction labels in the original program $P$; the instruction slots are assigned labels
from the set $\{|P| + 1, \ldots, n\}$. We define our closeness metric as:

$$
closeness(P, \hat{P}) = \sum_{i=1}^{\mid P \mid} (P.grd[i] = \hat{P}.grd[\xi(i)]) + \sum_{i=1}^{\mid P \mid} (P.stm[i] = \hat{P}.stm[\xi(i)]) + \sum_{i=\mid P \mid + 1}^{n} (\hat{P}.grd[\xi(i)] \neq false)
$$

The above metric weights a repair candidate by the changes in the statements/guards and
new statements added (insertion slots activated).

Algorithm 3 starts by asserting the input program $P$, via the selector variables if it is not
a locked location, as part of the constraints $\Phi_{grd}$ and $\Phi_{stm}$ (lines 3–11), and initializes the
insertion slots to their deactivated state (lines 12–14) with selector variables $t_i$. The constraint
$\Phi_{bk}$ ensures that the function $\xi$ is well-formed: for each $i$, $\xi(i)$ is a distinct value in the range
$\{1 \ldots n\}$ and is a monotonically increasing function (this ensures that the statements preserve
the same order in $\hat{P}$ as the order in $P$).

Finally, it uses issues a $\text{Solve()}$ query to an SMT solver to solve the repair constraint; the
sub-constraint $\Phi_{sem}$ contains the semantic encoding of our intermediate statements (Fig. 5)
and $\Phi_{spec}$ contains the specification collected during the debugging session as a result of the
_\text{spec} commands.

4.2 Proof-guided search space widening

To ensure that the repaired candidate program $\hat{P}$ is close to the original program, we progressively relax the closeness bounds. The variables $\tau_{grd}$, $\tau_{stm}$, and $\tau_{ins}$ constrain the distance
(in terms of changed guards, statements and activated insertion slots) of a repair candidate
from the original program.

We use a $\text{Proof-Guided Repair Strategy}$; the unsat core (uc) produced from the proof of
unsatisfiability directs us to the bound that needs to be relaxed. The unsat core represents the
central reason as to why the program cannot be made to satisfy the specification; if a constraint
$\langle \text{si} \Rightarrow \ldots \rangle$ is found in the unsat core, it implies that the reason for unsatisfiability may be
attributed to the fact that $s_i$ is false! Hence, one possible way to remove this unsatisfiability
is to increase the bound on $\tau_{stm}$ that allows $s_i$ to turn false.

At the same time, we would also like to enforce a priority on the relaxations; for instance,
deletion of a statement or mutation of a guard can be considered “smaller” changes than
changing a statement, or worse, inserting a new statement. The chain of conditions (lines 25–
27) ensures that, if the unsat core directs us to a possibility of smaller change, we relax
the respective bound before others. Finally, on a successful repair, we return the repaired
program $\hat{P}$.
Guiding repair via the unsat proof has multiple advantages:

- The unsatisfiability core (uc) guides us to a feasible repair; for example, if uc does not contain the constraints pertaining to activation of the insertion slots, then it is unlikely that inserting a new statement will fix the bug;
- It allows us to prioritize the repair actions; one would prefer mutation of a statement than the insertion of a new statement;
- The strategy is fast as the solver operates on a constrained search space that is incrementally widened (in a direction dictated by the proofs) as the search progresses. In case the program to be repaired is close to the original program, the solver will be provided only “easy” instances that are allowed to mutate/insert a small number of statements;
- It allows a fail-fast (line 28) if the specification is buggy or the repair is not possible due to structural constraints (like the number of insertion slots provided); if uc does not contain any constraint from \{\Phi_{grd}, \Phi_{stm}, \Phi_{ins}\}, then the program cannot be repaired via any repair action without violating the hard constraints (like the program semantics).

The unsat core not only identifies the possible culprits (a sort of bug localization) but also allows us to define a priority among our repair preferences. To the best of our knowledge, ours is the first repair algorithm that uses unsat proofs to direct repair; however, this idea has threads of similarity with a model-checking algorithm, referred to as underapproximation widening [23] (see Sect. 8).

We evaluated a variant (AlgVar) of our proof directed repair scheme: instead of increasing the respective repair bound, we randomly relax one of the constraints from the unsat core. However, we found that the unsat cores are poor—quite far from the minimum unsat core. Hence, this variant of our algorithm performs poorly, both in terms of success rate and the time taken for repair (see Sect. 7).

5 Bug localization

The objective of our bug localization module is to identify a (small) set of statements that are likely to contain the fault(s). Our algorithm is targeted at localizing faults for use by the repair phase of WOLVERINE2: our localization algorithm localizes faults on concrete program traces using the assertions as precondition/postcondition pairs.

The bug localization phase exposes two primitives to the repair phase:

- Statement locks: Adding a “locked” attribute to a statement asserts the statement in its position;
- Non-deterministic assignment: A non-deterministic assignment allows us to assign an angelic value.

**Definition 1** (Upward exposed statement) A statement whose left-hand side expression (variable or field definition) or its alias has not been assigned by any preceding program statement.

**Definition 2** (Downward exposed statement) A statement whose left-hand side expression (variable or field definition) or its alias has not been assigned by any following program statement.

**Definition 3** (Sandwiched statement) A statement that is neither upward exposed nor downward exposed.
We show states as a tuple with the values of variables \( \langle a, b, c, d \rangle \)

The text highlighted in yellow are program statements. The text in blue is some of the program analysis concepts. Text in black are the program states

### 5.1 Intuition

In this section, we provide the intuition behind our localization algorithm with a few examples.

#### 5.1.1 Program with a single bug and semantically independent statements

Consider the program shown in the second column of Table 2: in this case, all the statements are semantically independent, i.e., there does not exist any true dependencies among the statements. To understand the situation, let us also consider the states in the correct execution. The reader should be able to understand the state updates for the forward execution intuitively (first column); the details are discussed in Sect. 5.4.1.

The backward execution (third column) is an attempt at matching the correct execution—we commence from the postcondition, and “copy” states from the precondition whenever the assigned variable is upward exposed. Let us illustrate how the backward execution is performed: the backward execution commences from the postcondition and moves towards the precondition (it is essentially the weakest precondition computation, but under the assumption that some of the states could be buggy). For arriving at the backward execution state at the third statement, it checks if \( c \) is upward exposed, i.e., if it has a preceding assignment statement that assigns to the variable \( c \). As there is no such statement, it simply reads the state for \( c \) from the precondition. The same happens for the other statements. In this case, the backward execution exactly matches the states from the execution of the correct program (fourth column).

Now, one can see that the forward execution states agree with backward execution states at statements before the buggy statement, upon whose execution the difference between forward and backward execution builds up. The statement where this difference starts to build up is precisely the buggy location.

The procedure differs when a statement is not upward exposed: for the program shown in second column of Table 3, the statements are not upward exposed (first and second statement assign to the same variable \( b \)). Due to the previous update, it cannot be determined what would be the state when such a statement is run through a backtrackward traversal (as discussed above). In such a situation, we split the states, assigning all possible nodes in the data-
Table 3  Single bug program with semantically independent but not upward exposed statements

| Forward  | Backward | Diff. | Grad. | Correct Execution |
|----------|----------|-------|-------|-------------------|
| \(\langle n_1, n_2, n_3, n_5 \rangle\) | \(\langle n_1, n_2, n_3, n_5 \rangle\) | 0     |       |                   |
| \(b = a.\text{next}\) | \(b = a.\text{next}\) | 0     |       | \(\langle n_1, n_2, n_3, n_5 \rangle\) |
| \(\langle n_1, n_2, n_3, n_5 \rangle\) | \(\langle n_1, n_1, n_3, n_5 \rangle\) | 0     |       | \(\langle n_1, n_2, n_3, n_5 \rangle\) |
| \(\langle n_1, n_2, n_3, n_5 \rangle\) | \(\langle n_1, n_2, n_3, n_5 \rangle\) | 0     |       | \(\langle n_1, n_2, n_3, n_5 \rangle\) |
| \(b = b.\text{next}\) (fix: \(b = d\)) | \(\langle n_1, n_3, n_5, n_5 \rangle\) | 1     |       | \(\langle n_1, n_5, n_3, n_5 \rangle\) |
| \(c = c.\text{next}\) | \(\langle n_1, n_3, n_4, n_5 \rangle\) | 0     |       | \(\langle n_1, n_5, n_4, n_5 \rangle\) |

We show states as a tuple with the values of variables \(\langle a, b, c, d \rangle\)
The text highlighted in yellow are program statements. The text in blue is some of the program analysis concepts. Text in black are the program states.

Table 4  Single bug program with semantically dependent and not upward exposed statements

| Forward  | Backward | Diff. | Grad. | Correct Execution |
|----------|----------|-------|-------|-------------------|
| \(\langle n_1, n_2, n_3, n_5 \rangle\) | \(\langle n_1, n_2, n_3, n_5 \rangle\) | 0     |       |                   |
| \(a = a.\text{next}\) | \(a = a.\text{next}\) | 0     |       | \(\langle n_2, n_2, n_3, n_5 \rangle\) |
| \(\langle n_2, n_2, n_3, n_5 \rangle\) | \(\langle n_2, n_2, n_3, n_5 \rangle\) | 0     |       | \(\langle n_2, n_2, n_3, n_5 \rangle\) |
| \(b = b.\text{next}\) (fix: \(b = d\)) | \(\langle n_2, n_5, n_3, n_5 \rangle\) | 1     |       | \(\langle n_2, n_5, n_3, n_5 \rangle\) |
| \(c = b.\text{next}\) | \(\langle n_2, n_3, n_4, n_5 \rangle\) | 1     |       | \(\langle n_2, n_5, n_4, n_5 \rangle\) |

We show states as a tuple with the values of variables \(\langle a, b, c, d \rangle\)
The text highlighted in yellow are program statements. The text in blue is some of the program analysis concepts. Text in black are the program states.

structure to the concerned variable (here \(b\)), in the hope that at least one of them would be a state agreeing with the execution of the correct program.

We can compute the difference between two states by the number of variables whose values disagree amongst these states. We can define a gradient of the difference along between two traces \(t_1\) and \(t_2\) (of same length) by:

- Computing a differential trace \(\delta t\) by computing the difference between the corresponding states in \(t_1\) and \(t_2\);
- Finding a gradient of the differential trace, \(\partial(\delta t)\), by computing \((\partial(\delta t))_i = ||(\delta t)_i - (\delta t)_{i+1}||\) i.e. computing the distance between two corresponding trace elements. For a trace (sequence) \(t\), we use \(t_i\) to refer to its \(i\)th element.

For our example, there are points where there is a non-zero gradient of difference between states in forward and backward execution at one location—exactly the buggy statement (in the given example).
5.1.2 Program with a single bug and semantically dependent statements

In the program shown in Table 4, the third statement is not buggy, but is semantically dependent on the second statement, which is buggy, leads to a non-zero gradient of state difference even at the third statement, as the forward execution’s states further diverge from the backward (and correct) execution owing to dependence. Hence, both the buggy statement and its dependent statement is added to the set of suspicious statements.

5.1.3 Program with multiple bugs and containing semantically dependent statements

If the bugs occur in the dependent statements, then as discussed above, all of the statements are added to the set of suspicious statements. If the bugs are in independent statements, they will create a non-zero gradient independently, and, hence, the faulty statements will be added to the suspicious set.

5.2 Program transformation applied while invoking WOLVERINE2

Figure 6b shows the transformation when localization is performed by WOLVERINE2 on the intermediate representation (IR), expressed as a sequence of guarded statements. The input program is shown in Fig. 6a.

The statement identified as suspicious (shown in yellow color) are free to undergo mutations while the remaining statements are locked (denoted by enclosing them in square brackets). Since these statements do not depend on any buggy statement, they can also be made non-deterministic (shown by ★ on the right-hand side of these locked statements).

Fig. 6 Illustration of program transformation without insert slots

Fig. 7 Illustration of program transformation with insert slots
struct node *head,*slow_ptr,*fast_ptr

void deleteMid()
{
    struct Node *temp,*prev = NULL;
    slow_ptr = head;
    fast_ptr = head;
    temp = fast_ptr->next;
    while (fast_ptr != NULL && temp != NULL){
        fast_ptr = temp->next;
        if (fast_ptr != NULL)
        {
            temp = fast_ptr; //FIX: fast_ptr->next
        }
        prev = slow_ptr;
        slow_ptr = slow_ptr->next;
    }
    temp = slow_ptr->next;
    prev->next = temp;
    delete slow_ptr;
}

int main()
{
    push(2); push(4); push(8);
    deleteMid();
}

Fig. 8 Example code for illustrating bug localization

However, when the program allows the insertion of statements (the programmer had allowed non-zero insert slots), locked statements cannot be made non-deterministic. Figure 7b shows the transformed IR for a program with insert slots (Fig. 7a) and therefore the locked statements are frozen.

5.3 Running example

This section ties all the steps to show how WOLVERINE2 operates: we illustrate our localization algorithm in a typical repair session. The program in Fig. 8 deletes the middle of a singly-linked list; the list is created using a sequence of push() functions in the main() function.

The deleteMid() function has a bug in the while loop. The user starts the execution of the program and steps into the deleteMid() function with the created linked list. It has two pointers, slow_ptr and fast_ptr, which are both initiated to head of the list. In the loop, fast_ptr moves at a pace double that of slow_ptr until it points to the last or last but one node. The node that slow_ptr points at this time is deleted from the list.

After the statements before the while loop are executed, the state displayed to the user is labelled as F0 in Fig. 9. The state at this point meets the user’s expectation; the user decides to commit it and then enters the loop. The user next executes all the loop statements. The state at the end of the first iteration is labelled as F4 of Fig. 9. Due to the bug in the loop, the obtained state, F4, was not as expected, and hence the user makes a change to the state before the second commit. The updated state after the change is labelled as B4 in Fig. 9, and user asks WOLVERINE2 for a repair.

The tool now employs a backward traversal (via the backward semantics) from the correct state, B4, provided by the user, to localize the bug. Figure 6a shows the intermediate representation (IR) on which bug localization module operates. The first column in Fig. 9 shows the states in forward execution, and the second column shows the states in backward
execution at each statement between the committed states. The third column computes the difference in the states at respective forward and backward execution (the distance in the map representing the states), while the fourth column shows the gradient of the difference of the states corresponding to each program statement. In this example, only one statement has a non-zero gradient: the second statement of the loop (shown in the red circle in the last column), which is indeed the buggy statement.

Now, WOLVERINE2 produces a transformed abstract program (shown in Fig. 6b) with a smaller repair space, providing it to the repair algorithm, which synthesizes a repair in a mere 0.2s while the original program required 26s for the repair without localization (speedup of 130×).

5.4 Algorithm

We repeat the notations used for the reader’s convenience: $S$ represents a state of a program with a set of variables $\sigma_V$ and a set of heap nodes $\sigma_H$ with fields $\sigma_F$ as $V \times H$; the state of the program variables, $V$, is a map $\sigma_V \rightarrow D$ and the program heap is represented by $H$ as a map $\sigma_H \times \sigma_F \rightarrow D$. The domain of possible values, $D$, is $\mathcal{I} \cup \sigma_H$ where $\mathcal{I}$ is the set of integers. For simplicity, we constrain the discussions in this paper to only two data-types: integers and pointers. We use the function $\Upsilon(e)$ to fetch the type of a program entity; a program entity $e \in \mathcal{E}$ is either a variable $v \in \sigma_V$ or a field of a heap node $h \in \sigma_H \times \sigma_F$. Also, pointers can only point to heap nodes as we do not allow taking reference to variables.
\begin{itemize}
\item a = a->next;
\item a = curr;
\item if (curr != NULL) {
\item b = b->next;  
\item b = curr;  
\item curr = curr->next;
}\end{itemize}

(a) Input program

\begin{itemize}
\item (true) : a = a.next
\item (true) : a = curr
\item (true) : b0 = (curr != NULL)
\item (b0) : b = b.next
\item (b0) : b = curr
\item (b0) : curr = curr.next
\end{itemize}

(b) IR of the program shown in Figure 10a

\textbf{5.4.1 Forward execution}

Let $\Delta$ denote the operational semantics in the forward execution.

\[ \Delta : \text{State} \times \text{statement} \rightarrow \text{State} \]

The (repaired) program must satisfy the correctness criterion: $\Delta(\omega_{\text{pre}}, \text{stmt}) = \omega_{\text{post}}$

where stmt is a program statement and $\omega_{\text{pre}}$, $\omega_{\text{post}}$ are the precondition (state before executing statement “stmt”) and postcondition (state after executing statement “stmt”).

Given a program trace of $n$ statements $[s_0, s_1, \ldots, s_n]$, we define $\Delta_i$ as the transition function for the $i$th statement ($s_i$) in the trace. Subsequently, we denote $\Delta_{i,j}$ to denote the forward transition function for statements from statements $s_i$ to $s_j$.

\[ \Delta_{i,j} = \Delta_i \circ \Delta_{i+1} \circ \Delta_{i+2} \circ \cdots \circ \Delta_j \]

In particular, a transition function for a complete trace of $n$ statements can be written as:

\[ \Delta_{1,n} = \Delta_1 \circ \Delta_2 \circ \Delta_3 \circ \cdots \circ \Delta_n \]

Forward execution follows the forward semantics (Fig. 5). We illustrate forward execution via Fig. 9: we use the getfld rule to execute $fast = temp.next$; the node pointed by fast is updated to n1 (state F1). We, then, use the asgn rule to execute $temp = fast$, which changes the node pointed by temp from n2 to n1(state F2). Then, we again use the asgn rule to update the value of prev to n3, followed by getfld rule, to revise value of slow to the node n2 (F3 and F4 respectively).

\textbf{5.4.2 Backward execution}

We use $\nabla$ to denote the operational semantics for the backward execution.

\[ \nabla : \text{State} \times \text{statement} \rightarrow 2^{\text{State}} \]

In this case, we may get a set of states instead of a single state (as the same state could be reached by multiple input states).

\[ \nabla(\omega_{\text{post}}, \text{stmt}) = \{\omega_{\text{pre1}}, \omega_{\text{pre2}}, \omega_{\text{pre3}}, \ldots\} \]
\[ \nabla_{n,0}(\omega_{\text{post}}) = \nabla_{n,n-1} \circ \nabla_{n-1,n-2} \circ \nabla_{n-2,n-3} \circ \cdots \circ \nabla_{1,0} = \{\Omega_{in1}, \Omega_{in2}, \ldots\} \]

We define the backward execution in Fig. 11. Let us illustrate it using the program shown in Fig. 10a (and its IR in Fig. 10b). This program has no bug, and we only use it for elucidation. Backward execution uses the rules described in Fig. 12. P0 represents the program state before the start of the execution, and P6 shows the program state after all statements have been executed. We have copied the state P0 to Initial State and state P6 to B6 on the right-hand side for convenience. $N$ is the set of all nodes (including null) in the program at a particular
program point. Starting from state B6, it uses getfld₁ rule, (because the LHS of statement \( curr = curr \rightarrow next \) does not appear in any previous program statement) and this updates the value of \( curr \) to its value in the Initial State (node n₁), as shown in state B5. We define two program statements as matching if their LHS writes to the same program variable or to a field of the same program variable. Now, since the LHS of statement \( b = curr \) is same as LHS of previous statement \( b = b \rightarrow next \), therefore we cannot read the value of \( b \) from the initial state. In this case, \( b \) is the variable to which both of these statements are writing. The value of \( b \) has changed (from that in initial state P₀) due to the presence of a matching statement. Since we cannot be sure of the actual value of \( b \) at this program point, to perform backward execution, we assign all possible node values (including null) to \( b \). We call this process “splitting of states”. This is the use of \( \text{assign}_2 \) rule and the set of possible states is B₄(a), B₄(b) and B₄(c).

For backward executing \( b = b \rightarrow next \), WOLVERINE₂ uses getfld₁ rule to update the value of \( b \). This rule was used because, now, no previous program statement has the same LHS (b). Therefore, we update all the states (B₄(a), B₄(b) and B₄(c)) with the value of \( b \) in the initial state (node n₁) to give state B₅. Although we maintain all the split states, we only show unique states in the figure. Next, the \( \text{assign} \) rule is used, which performs no updates to the state (shown in state B₂). Now, we will again have to split states as the LHS of statement \( a = curr \) and \( a = a \rightarrow next \) are the same. WOLVERINE₂ uses the \( \text{assign}_2 \) rule to assign all possible node values to \( a \), and the obtained set of states are shown in B₁(a), B₁(b) and B₁(c). Finally, the getfld₁ rule is used to revise the value of \( a \), and this time, it can be read from the initial state (node n₂); this state is labelled as B₀. Note that, in this case, state B₀ converges to the Initial State, showing that the program does not have a bug.

Fig. 11 Forward and backward execution of IR code shown in Fig. 10b
a set of states, while \( \omega \) represents a single state.

**Lemma 1** If \( i \)th statement is upward exposed then, \( \Delta_{0,i-1}(\omega_{pre}, P)[LHS(s_i)] = \omega_{pre}[LHS(s_i)] \).

**Proof** Since \( i \)th statement is upward exposed, no other preceding statement or aliased variable can change the value of \( LHS(s_i) \), hence its value remains the same as in precondition (\( \omega_{pre} \)).

**Lemma 2** If \( i \)th statement is downward exposed then, \( \nabla_{n,i+1}(\omega_{post}, P)[LHS(s_i)] = \omega_{post}[LHS(s_i)] \).

**Proof** Since \( i \)th statement is downward exposed, no other succeeding statement or alias variable changes the value of \( LHS(s_i) \), hence its value remains the same as in postcondition (\( \omega_{post} \)) during backward execution. There has not been any splitting of states for \( LHS(s_i) \) before \( i \)th statement is backward traversed.

We provide our complete bug localization algorithm (Algorithm 4) for detecting suspicious program statements. Line 2 in Algorithm 4 is the forward trace, that is, the sequence of states obtained from the forward execution of the program. Line 3 is the backward trace, or the sequence of states obtained from backward execution; as we have seen, spitting can happen in the backward execution leading to several states at a program point (so \( \Omega_i \) is a set of states). Since the splitting is uniform, we can count the number of children of a particular state in the backward trace by dividing the final number of states at the end of backward execution (denoted by \( M \)) and the number of states at that program point. This process gives a multiplying factor at each program point. Line 4 calculates the distance between the state in forward execution and the state(s) in backward execution, multiplied by the multiplying factor pertaining to that point. This operation is denoted by \( \otimes \) in the algorithm. Each \( \Gamma_i \) stores the result of this operation. Starting from the last program statement, if the pairwise
Algorithm 4: Pick suspicious statements

\textbf{Input:} $\phi \mapsto [(P, n, \Omega_{\text{pre}}, \Omega_{\text{post}})]$

1 $\text{SuspiciousSet} = \emptyset$

2 $[\omega_f^0, \omega_f^1, \omega_f^2, \ldots, \omega_f^n] \leftarrow [I, \Delta_1, \Delta_2, \ldots, \Delta_n](\omega_{\text{pre}}, P)$

3 $[\Omega_{b}^0, \Omega_{b}^1, \Omega_{b}^2, \ldots, \Omega_{b}^n] \leftarrow [\nabla_{n,1}, \nabla_{n,2}, \ldots, \nabla_{n,n}, I](\omega_{\text{post}}, P)$

4 $[\Gamma^0, \Gamma_1, \Gamma_2, \ldots, \Gamma_n] \leftarrow [\omega_f^0, \omega_f^1, \omega_f^2, \ldots, \omega_f^n] \otimes [\Omega_{b}^0, \Omega_{b}^1, \Omega_{b}^2, \ldots, \Omega_{b}^n]$

5 \textbf{for} $i \in \{1 \ldots n\}$ \textbf{do}

6 \hspace{1em} \textbf{if} $\Gamma_{i-1} \neq \Gamma_i$ \textbf{then}

7 \hspace{2em} $\text{SuspiciousSet} = \text{SuspiciousSet} \cup \{i\}$

8 \hspace{1em} \textbf{end}

9 \textbf{return} $\text{SuspiciousSet}$

difference of the $\Gamma_i$’s is non-zero (this is the gradient), the $i$th statement is added to the set of suspicious statements (Line 5-7), which is returned at the end (Line 9).

Let the number of states in $\nabla(P, \Omega_{\text{post}}, i)$ be $p$ and the number of states at the end of backward execution $\nabla_{n,0}(P, \Omega_{\text{post}})$ be $M$.

Let $\nabla_{n,i} = \Omega_{b}^i$ and $\Delta_{1,i} = \omega_f^i$. Then,

$$\Gamma_i = \omega_f^i \otimes \Omega_{b}^i = \frac{M}{|\Omega_{b}^i|} \ast (|\Omega_{b}^i[n] - \omega_f^i| + |\omega_f^i[n] - \Omega_{b}^i|)$$

In Fig. 11, upon complete backward execution, we get a total of 9 states. The number of states at every program point is shown on the right-hand side (in the aligned box) Fig. 11. The multiplying factor for a state can be determined by dividing 9 by its number of states.

$\Gamma_1$ at P1 is the sum of the difference between the nine states in backward execution (right side), out of which only three are shown (as others are duplicated). With the state in forward execution (left side) at P1, the sum of differences is 6.

We provide the theoretical analysis of this algorithm in the Appendix.

6 Advanced debugging/repair

In this section, we show how skilled engineer can employ the features in WOLVERINE2 for effective debug-repair sessions.

6.1 Specification refinement

WOLVERINE2 is designed to model heap manipulations; however, WOLVERINE2 can use the concrete($\zeta$) statement in its intermediate representation as an abstraction of any statement $\zeta$ that it does not model. On hitting a concrete($\zeta$) statement, WOLVERINE2 uses gdb to concretely execute the statement and updates its symbolic state from the concrete states provided by gdb. Figure 13 shows an instance where we wrap the $i=i+1$ statement in an concrete execution; WOLVERINE2 translates this statement to a string of gdb commands, and the symbolic state is updated with the value of $i$ from the concrete state that gdb returns after executing the statement. Hence, although WOLVERINE2 is specifically targeted at heap manipulations, it can also be used to debug/repair programs containing other constructs as
long as the bug is in heap manipulation statements. We refer to this technique of reconstructing the symbolic specification by running the statement concretely as specification refinement. Specification refinement can be used in creative ways by skilled engineers. In Fig. 13, the programmer decided to wrap a complete function call \((\text{foo}())\) within the \texttt{concrete()} construct, allowing \textsc{Wolverine2} to reconstruct the effect of the function call via concrete execution without having to model it. This strategy can fetch significant speedups for repair: let us assume that, in Fig. 1, the programmer uses her domain knowledge to localize the fault to Lines 6–8; she can pass this information to \textsc{Wolverine2} by wrapping the other statements in the loop (lines 5,9) in \texttt{concrete} statements; this hint brings down the repair time on the full program on the complete execution from 6.0s to 1.5s, i.e., achieving a \(4 \times\) speedup (on our machine). For this experiment, we turned off the fault localizer in \textsc{Wolverine2}. This is understandable as each instruction that is modeled can increase the search space exponentially.

### 6.2 Checkpoint-based hopping

This feature comes in handy when the programmer herself is not sure about the correctness of the expected specification she is asserting. For example, Fig. 14 shows the code for reversal...
of a doubly-linked list, with bugs in two statements in the while loop (these bugs are different from the ones in Fig. 1).

The programmer steps into the reverse() function after creating the doubly linked-list.

```c
(Wolverine2) start
Starting program...
push(2)
(Wolverine2) next; next; next; next;
push(4);
...
reverse();
(Wolverine2) step

current = head;
(Wolverine2) next
```

She executes the first statement in reverse() and then asserts the program state. Whenever the user asserts a state, WOLVERINE2 creates a checkpoint of this state at that program point; checkpointing memorizes important events during a debug run, allowing the user to resume a new direction of debugging from this location, if required (illustrated later). A Checkpoint ID, which keeps a count of the checkpoints (0 in this case), is returned. This ID can be used to resume debugging from this corresponding checkpoint/

```c
while(current != NULL)
(Wolverine2) spec
Program states added – Checkpoint 0
(Wolverine2) enter

She employs the next command to execute till the end of the loop.

```c
temp1 = current->prev;
(Wolverine2) next; next; next; next; next;
...
while(current != NULL)
```

The program state displayed to her is shown in Fig. 15a: due to the fact that the effect of Bug#1 and Bug#2 cancel out, she observes that the data-structure has not changed except the current pointer. Since she expected the reversal of the first node by the end of this loop iteration, she issues the desired changes and asserts the state (Checkpoint 1); the updated heap is shown in Fig. 15b.

```c
(Wolverine2) change n4 -> prev n3
(Wolverine2) change current -> n3
```

```c
while(current != NULL)
(Wolverine2) spec
Program states added – Checkpoint 1
```

She now enters the loop for the second time.

```c
(Wolverine2) enter
while(current != NULL)
```

At the end of this iteration, the user again finds an unexpected state and issues necessary changes to reverse the next node.

```c
(Wolverine2) change current n2
(Wolverine2) change n3 -> prev n2
(Wolverine2) change n3 -> next n4
```

Satisfied with the updated state (shown in Fig. 15c), she asserts it (Checkpoint 2).

```c
while(current != NULL)
(Wolverine2) spec
Program states added – Checkpoint 2
```
Fig. 15  Figures for checkpoint-based hopping example

She, now, feels less convinced about her hypothesis regarding the correct run of the program and, thus, about the asserted program states and, therefore, decides to try out another direction of investigation. This would have required her to abandon the current session and spawn a new session; not only will it require her to resume debugging from the beginning, but she would also lose the current debugging session, preventing her from resuming in case she changes her mind again. WOLVERINE2 packages feature for such a scenario where a programmer may be interested in exploring multiples directions, allowing them to save and restore among these sessions at will. In this case, instead of exploring from the initial state, the programmer adds a checkpoint for the current state (Checkpoint 2) and returns to Checkpoint 0 by issuing 
\begin{verbatim}
while(current != NULL)
(Wolverine2) resume 0
Program resumed at checkpoint 0
\end{verbatim}

She executes through the first loop iteration in a similar manner (thus, the obtained state is the same as shown in Fig. 15a). She issues the following changes to match her expectations and asserts the updated program state (Checkpoint 3).

\begin{verbatim}
(Wolverine2) change current n3
(Wolverine2) change n3 -> prev n4
(Wolverine2) change n3 -> next null
\end{verbatim}

\begin{verbatim}
while(current != NULL)
(Wolverine2) spec
Program states added – Checkpoint 3
\end{verbatim}
To her surprise, the state shown to her remains unchanged after these modifications (barring the current pointer). She is now more confident that the states she had asserted in the previous session were correct and now wants to return to it. She issues \texttt{resume 2}.

\textsc{Wolverine2} essentially maintains the constraints corresponding to the checkpointed states along with the different debugging states in a directed-acyclic graph; when the programmer resumes from checkpoint 2, \textsc{Wolverine2} pops off all the constraints that were asserted till the first common ancestor (Checkpoint 0) of the current (Checkpoint 3) and the requested checkpointed state (Checkpoint 2) from the solver, and then, pushes the constraints till the resumed checkpoint (Checkpoint 2). They have been shown as red and green dotted lines, respectively, in Fig. 15d.

\section{7 Experiments}

We built \textsc{Wolverine2} using the \texttt{gdb} Python bindings \cite{16}, the C-to-AST compiler uses \texttt{pycparser} \cite{5}, the visualization module uses \texttt{igraph} \cite{30} to construct the box and arrow diagrams and the repair module uses the Z3 \cite{11} theorem prover to solve the SMT constraints. We conduct our experiments on an Intel(R) Xeon(R) CPU @ 2.00GHz machine with 32 GB RAM. To evaluate our implementation, we attempt to answer the following research questions:

\begin{itemize}
  \item \textbf{RQ1.} Is our repair algorithm able to fix different types and combinations of bugs in a variety of data-structures?
  \item \textbf{RQ2.} Can our repair algorithm fix these bugs in a reasonable time?
  \item \textbf{RQ3.} How does our repair algorithm scale as the number of bugs is increased?
  \item \textbf{RQ4.} What is the accuracy of our localization algorithm with respect to other localization algorithms?
  \item \textbf{RQ5.} What is the impact of our algorithm (localization + repair) on the repair time of \textsc{Wolverine2}?
  \item \textbf{RQ6.} Is \textsc{Wolverine2} capable of debugging/fixing real bugs?
\end{itemize}

We conduct our study on 40 heap manipulating programs (Table 5) from online sources \cite{15} for a variety of data-structures like singly, doubly, and circular linked lists, AVL trees, Red-Black trees, Splay Trees, and Binary Search Trees.

Though there has been a large body of work on automated debugging and repair \cite{3,7,8,36,41,42,46–51,56,63,64,68,69}, these techniques cannot tackle repair over deep properties like functional correctness of heap data-structures. Our work is more in line with the following papers involving synthesis and repair of heap manipulating programs \cite{18,28,43,44,53,57,62} or that involve functional correctness of student programs \cite{6,24,26,29,54,55,61,66,67}. Hence, our collection of benchmarks are similar to the above contributions.

We divide our benchmarks into \texttt{Small} and \texttt{Large} benchmarks: the \texttt{Large} benchmarks involve more complex control-flow (nested conditions and complex Boolean guards) and are about $3 \times$ larger than the \texttt{Small} programs (in terms of the number of IR instructions).

\subsection{7.1 Experiments with fault-injection}

We create buggy versions via an in-house fault injection engine that automatically injects bugs (at random), thereby eliminating possibilities of human bias. For each program, we control our fault-injection engine to introduce a given number of bugs. We characterize a buggy version by $\langle x, y \rangle$, implying that the program requires mutation of $x$ (randomly selected)
Table 5 Our two sets of benchmarks [15]

Small benchmarks

| B1    | Reverse singly linked-list          |
| B2    | Reverse doubly linked-list          |
| B3    | Deletion from singly linked-list    |
| B4    | Creation of circular linked-list    |
| B5    | Sorted Insertion singly linked list |
| B6    | Insertion in single linked list     |
| B7    | Swapping nodes singly linked list   |
| B8    | Splaytree Left Rotation             |
| B9    | Minimum in Binary Search Tree       |
| B10   | Find Length of singly linked list   |
| B11   | Print all nodes singly linked list  |
| B12   | Splaying of circular linked list    |
| B13   | AVL tree right rotation             |
| B14   | AVL tree left-right rotation        |
| B15   | AVL tree left rotation              |
| B16   | AVL tree right-left rotation        |
| B17   | Red-Black tree left rotate          |
| B18   | Red-Black tree right rotate         |
| B19   | Enqueue using linked-list           |
| B20   | Splaytree Right Rotated             |

Large benchmarks

| L1    | Delete middle of singly linked-list|
| L2    | Remove duplicate in singly linked-list|
| L3    | Last node to first singly linked-list|
| L4    | Intersection of two singly linked-list|
| L5    | Split singly linked-list into two lists|
| L6    | Value-based partition of singly linked-list|
| L7    | Delete specific node in singly linked-list|
| L8    | Splitting of circular linked-list|
| L9    | Middle node as head singly linked-list|
| L10   | Merge alternate nodes in singly linked-list|
| L11   | Delete node of specific value in singly linked-list|
| L12   | Splitting of doubly linked-list|
| L13   | Pairwise swap of nodes in singly linked-list|
| L14   | Rearranging singly linked-list|
| L15   | Absolute sort of nodes in singly linked-list|
| L16   | Quicksort of singly linked-list|
| L17   | Delete specific node in doubly linked-list|
| L18   | Sorted insertion in singly linked-list|
| L19   | Remove duplicate nodes in doubly linked-list|
| L20   | Constrained deletion in singly linked-list|

program expressions and the insertion of \( y \) newly synthesized program statements. The value of \( x \) is determined by the number of mutations in the correct program. A mutation consists of replacing a program variable or the field of a variable with another variable or another field (randomly chosen from the program space) in a program statement or a guard. The value of \( y \) is determined by the number of program statements the engine deletes from the correct program. WOLVERINE2 is unaware of the modifications and deletions when fed the modified program.

For the experiments, WOLVERINE2 makes ten attempts at repairing a program, each attempt followed by \textit{proof-directed search space widening}; each attempt is run with a timeout of 30s. The experiment was conducted in the following manner:

1. We evaluate each benchmark (in Table 5) for four bug classes: Class1 (\((1, 0)\)), Class2(\((1, 1)\)), Class3(\((2, 0)\)) and Class4(\((2, 1)\));
2. For each benchmark \( B_i \), at each bug configuration \((x, y)\), we run our fault injection engine to create \(40\) buggy versions with \( x \) errors that require modification of an IR instruction and \( y \) errors that require insertion of a new statement;
3. Each of the above buggy programs is run twice to amortize the run time variability.

For RQ1 and RQ2, we use the 20 Small programs as these experiments involve only the repair tool (sans the localizer). Figure 16 shows the average time taken to repair a buggy configuration over the 20 buggy variants, which were themselves run twice (the reported time shows the average time taken for the successful repairs only). We report the time taken for our main algorithm (in Algorithm 3) and its variant AlgVar (discussed in the last paragraph.
in Sect. 4). Our primary algorithm performs quite well, fixing most of the repair instances in less than 5 s; understandably, the bug classes that require insertion of new instructions (Classes 2 and 4) take longer. There were about 1–4 widenings for bugs in class 1, 2, 3; the bugs in class 4 were more challenging, needing 2–6 widenings.

In terms of the success rate, our primary algorithm was able to repair all the buggy instances. However, Fig. 16c shows the success rate for each bug configuration for AlgVar; the success rate is computed as the fraction of buggy instances (of the given buggy configuration) that could be repaired by Wolverine2 (in any of the two attempts).

The inferior performance of the variant of our main algorithm shows that the quality of the unsat cores is generally poor, while the performance of our primary algorithm demonstrates that even these unsat cores can be used creatively to design an excellent algorithm.

Figure 16d answers RQ3 by demonstrating the scalability of Wolverine2 with respect to the number of bugs on (randomly selected) five of our smaller benchmarks. We see that in most of the benchmarks, the time taken for repair grows somewhat linearly with the number of bugs, though (in theory), the search space grows exponentially. Also, one can see that more complex manipulations like left-rotation in a red-black tree (B17) are affected more as a larger number of bugs are introduced compared to simpler manipulations like inserting a node in a sorted linked list (B5).
Table 6  Coverage statistics of the
tests generated for comparison
with other bug localization
techniques in Fig. 17

| Benchmark | # Tests | Line coverage | Branch coverage |
|-----------|---------|---------------|----------------|
| B1        | 17.0    | 100.0         | 100.0          |
| B2        | 61.0    | 89.0          | 89.0           |
| B3        | 17.0    | 99.0          | 100.0          |
| B4        | 335.0   | 100.0         | 100.0          |
| B5        | 17.0    | 100.0         | 98.0           |
| B6        | 121.0   | 100.0         | 100.0          |
| B7        | 61.0    | 93.0          | 89.0           |
| B8        | 17.0    | 98.0          | 99.0           |
| B9        | 13.0    | 97.0          | 87.0           |
| B10       | 17.0    | 93.0          | 96.0           |

The variance in the runtimes for the different buggy versions, even for those corresponding
to the same buggy configuration, was found to be high. This is understandable as SMT solvers
often find some instances much easier to solve than others, even when the size of the respective
constraint systems is similar.

For RQ4, we compared our algorithm with the state-of-the-art bug localization algorithms
we found in literature [1,32,38,39]. All these algorithms require the creation of test cases
that differentiate between the correct and incorrect behavior of the program; our algorithm,
on the other hand, does not require a test suite and localizes the bug using a single trace. To
compare our algorithm against the existing algorithms, we developed a test generation engine
for heap manipulating programs. We randomly selected 10 heap manipulating programs from
our larger benchmarks (in Table 5). Bugs were injected using our fault injection engine. We
created 20 versions of the program, each having a single bug for all the selected benchmarks
(creating a total of 200 buggy programs). For each benchmark, Table 6 shows the average
number of test cases generated, and the average statement and branch coverage produced by
these test cases averaged over the 20 buggy versions.

To compare across benchmarks of differing sizes, we normalize the average rank of the
buggy statement produced by these algorithms with the program size; hence, we report the
developer effort, i.e., the percentage of the lines of code to be examined before the faulty line
is encountered.

Figure 17 shows the line plot of the number of programs (out of 200) in which the average
rank of the buggy statement produced by different algorithms was within a given effort
threshold (% of program length on the x-axis). The plot shows that our algorithm is able to
rank the ground truth repair in 150 of the 200 programs to within 10% of the program size,
and all the 200 programs to 25% of the program size. On the other hand, the best performing
metric among the other algorithms (Tarantula) is only able to rank 33 of the 200 programs
to 25% of the program size.

For RQ5, we choose all the 40 heap manipulating programs (in Table 5): for each bench-
mark, $B_i$, at each of the four configurations, we create 40 buggy versions. We compare our
tool with two configurations of the repair tool (sans localization):

1. **Section-wise repair**: when the user has some prior information about the bug and confines
   the repair tool’s search to only a section of the program (like the loop head, a loop body,
   etc.); a section can contain multiple nested control-flow statements but does not cross
   loop boundaries;
2. **Unconfined repair**: the repair tool is unleashed on the whole program.

Figure 18a compares the repair time (averaged over the 40 buggy versions for single bug configuration) required for WOLVERINE2 compared in three cases: (1) when localization is used, (2) when the bug is naively localized to a section of the code, and (3) when the bug localization is not used at all. The plot is in log scale. It clearly shows that using our bug localization algorithm reduces repair time by several orders of magnitude. On average over all benchmarks we were $50 \times$ faster compared to the section-wise repair and $190 \times$ (upto $779 \times$ on some benchmarks) faster than in the unconfined repair setting.

Figure 18b shows the average repair time compared for the three cases for double bugs (two bugs in the program). Figure 18c shows the average repair time compared for the three cases for single bug and one insert slot configuration. Figure 18d shows the average repair time compared for the three cases for double bug and one insert slot configuration.

We show a summary of the average speedup results in Table 7, both for the Small and Large instances (we do not provide plots for the smaller instances); understandably, localization benefits the Large benchmarks more than the Small benchmarks, illustrating the effectiveness of our localization algorithm in the repair of larger, more complex instances. For the programs that timed-out, we consider their runtimes as the timeout period (300s).

### 7.2 Experiments with student submissions

In order to answer RQ6, we collected 247 buggy submissions from students corresponding to 5 programming problems on heap manipulations from an introductory programming course [10].

We attempted repairing these submissions and categorized a submission into one of the following categories (shown in Table 8):

- **Fixed**: These are the cases where WOLVERINE2 could automatically fix the errors.
- **Implementation Limitations**: These are cases where, though our algorithm supports these repairs, the current state of our implementation could not support automatic repair.
- **Out of scope**: The bug in the submission did not occur in a heap-manipulating statement.
Log scaled timing across localized, sectional-wise and unconfined repair (Class1)

Log scaled timing across localized, sectional-wise and unconfined repair (Class2)

Log scaled timing across localized, sectional-wise and unconfined repair (Class3)

Log scaled timing across localized, sectional-wise and unconfined repair (Class4)

Fig. 18 Time comparison with and without localization (section-wise and unconfined) repair cases (second set of benchmarks)

– **Vacuous**: In these submissions, the student, had hardly attempted the problem (i.e., the solution is almost empty).

Overall, we could automatically repair more than 80% of the submissions where the student has made some attempt at the problem (i.e., barring the vacuous cases).

8 Related work

Our proof guided repair algorithm is inspired by a model-checking technique for concurrent programs—referred to as underapproximation widening [23], that builds an underapproximate model of the program being verified by only allowing a specific set of thread interleavings by adding an underapproximation constraint that inhibit all others. If the verification instance finds a counterexample, a bug is found. If a proof is found which does not rely on the underapproximation constraint, the program is verified; else, it is an indication to relax the underapproximation constraint by allowing some more interleavings. Hence, the
Table 7: Average speedups gained when compared to case of section-wise repair and case of unconfined repair. The numbers in the parenthesis show the maximum speedup obtained in respective classes for the case of unconfined repair.

| Bug Configuration | Class1 | Class2 | Class3 | Class4 |
|-------------------|--------|--------|--------|--------|
| Section-wise repair | 14     | 4      | 10     | 8      |
| Unconfined repair  | 39 (151)| 12 (33)| 31 (134)| 22 (64)|

Table 8: Tool evaluation on student submissions.

| Id | Total | Fixed | Impl. Limit | Out of scope | Vacuous |
|----|-------|-------|-------------|--------------|---------|
| P1 | 47    | 30    | 2           | 8            | 7       |
| P2 | 48    | 29    | 3           | 8            | 8       |
| P3 | 48    | 36    | 0           | 5            | 7       |
| P4 | 61    | 46    | 0           | 6            | 9       |
| P5 | 43    | 25    | 0           | 4            | 14      |

algorithm can find a proof from underapproximate models without needing to create abstractions. To the best of our knowledge, ours is the first attempt at adapting this idea for repair. In the case of repairs, performing a proof-guided search allows us to work on smaller underapproximated search spaces that are widened on demand, guided by the proof; at the same time, it allows us to prioritize among multiple repair strategies like insertion, deletion, and mutation. There have also been some attempts at using proof artifacts, like unsat cores, for distributing large verification problems [9]. In the space of repairs, DirectFix [45] also builds a semantic model of a program but instead uses a MAXSAT solver to search for a repair. Invoking a MAXSAT solver is not only expensive, but a MAXSAT solver also does not allow prioritization among repair strategies. In DirectFix, it is not a problem as the tool only allows mutation of a statement for repair and does not insert new statements. Alternatively, one can use a weighted MAXSAT solver to prioritize repair actions, but it is prohibitively expensive; we are not aware of any repair algorithm that uses a weighted MAXSAT solver for repair.

Inspired by the success of WOLVERINE, there have been proposals at using proof-guided techniques for synthesis and repair: GAMBIT [64] uses a proof-guided strategy for debugging concurrent programs under relaxed memory models. It also provides an interactive debugging environment, similar to WOLVERINE2, but focuses its debugging/repair attempts at concurrent programs, operating under varying memory models. MANTHAN [20,21] uses a proof-guided approach to synthesis; instead of starting from a buggy program, it learns an initial version of the program from input-output examples. It, then, uses a similar repair engine as WOLVERINE and WOLVERINE2 to repair the candidate.

Zimmermann and Zeller [72] introduce memory graphs to visualize the state of a running program, and Zeller used memory graphs in his popular Delta Debugging algorithms [70,71] to localize faults. Our algorithm is also based on extracting these memory graphs from a concrete execution on gdb and employing its symbolic form for repair. The notion of
concrete statement in WOLVERINE2 bears resemblance to the concolic testing tools [19, 59].

Symbolic techniques [2,4,22,33,34,40,51,52,64] build a symbolic model of a program and use a model-checker or a symbolic execution engine to “execute” the program; they classify a statement buggy based on the “distances” of faulty executions from the successful ones. Angelic Debugging [7], instead, uses a symbolic execution engine for fault localization by exploring alternate executions on a set of suspicious locations, while Angelix [46,48] fuses angelic debugging-style fault localization with a component-based synthesis [31] framework to automatically synthesize fixes. There have also been regression aware strategies to localize/repair bugs [4]. There have also been proposals to use statistical techniques [38,39,47], evolutionary search [36,50,68,69] and probabilistic models [42] for program debugging. However, though quite effective for arithmetic programs, the above algorithms were not designed for debugging/repairing heap manipulations. There have been proposals that repair the state of a data-structure on-the-fly whenever any consistency check (from a set of checks provided by a user) is found to fail [12,13]. However, our work is directed towards fixing the bug in the source code rather than in the state of the program, which makes this direction of solutions completely unrelated to our problem. In the space of functional programs, there has been a proposal [14,35] to repair functional programs with unbounded data-types; however, such techniques are not applicable for debugging imperative programs. Finally, WOLVERINE2 uses a much lightweight technique for fault localization than expensive MAXSAT calls.

There has been some work in the space of synthesizing heap manipulations. The storyboard programming tool [62] uses abstract specifications provided by the user in three-valued logic to synthesize heap manipulations. As many users are averse to writing a formal specification, SYNBAD [57] allows the synthesis of programs from concrete examples; to amplify the user’s confidence in the program, it also includes a test-generation strategy on the synthesized program to guide refinement. SYNBAD inspires the intermediate representation of WOLVERINE2; WOLVERINE2 can also be extended with a test-generation strategy to validate the repair on a few more tests before exposing it to the programmer. SYNLIP [18] proposes a linear programming based synthesis strategy for heap manipulations. Feser et al. [14] propose techniques for synthesizing functional programs over recursive data structures. WOLVERINE2, on the other hand, attempts repairs; the primary difference between synthesis and repair is that, for a “good” repair, the tool must ensure that the suggested repair only makes “small” changes to the input program rather than providing a completely alternate solution. Other than synthesis of heap manipulations, program synthesis has seen success in many applications, from bit-manipulating programs [31], bug synthesis [58], parser synthesis [37,60] and even differentially private mechanisms [56]. Fault localization techniques have seen both statistical and formal algorithms. Statistical debugging techniques [1,8,32,38,39,47] have been highly popular for large code-bases. These techniques essentially attempt to discover correlations between executions of parts of the program and its failure. However, though these techniques work quite well for large codebases, they are not suitable for somewhat smaller, but tricky programs, like heap manipulations. Our experiments (RQ4) demonstrate this and thus motivate different fault localization techniques for such applications. Moreover, these techniques essentially provide a ranking of the suspicious locations and hence are somewhat difficult to adopt with repair techniques. On the other hand, our localization algorithm provides a sound reduction in the repair space, thereby fitting quite naturally with the repair.
9 Discussion and conclusion

We believe that tighter integration of dynamic analysis (enabled by a debugger) and static analysis (via symbolic techniques) can open new avenues for debugging tools. This work demonstrates that a concrete execution on a debugger to collect the potentially buggy execution and the user-intuitions on the desired fixes, fed to a bug localizer that contracts the repair space, and a proof-directed repair algorithm on the reduced search space, is capable of synthesizing repairs on non-trivial programs in a complex domain of heap-manipulations. We are interested in investigating more in this direction.

There exist threats to validity to our experimental results, in particular from the choice of the buggy programs and how the bugs were injected. We were careful to select a variety of data-structures and injected bugs via an automated fault injection engine to eliminate human bias; nevertheless, more extensive experiments can be conducted.

Author Contributions Not applicable.

Funding Not applicable.

Availability of data and material To be made available post-publication.

Declarations

Conflicts of interest The second author has visits and collaborations with NUS Singapore, UW-Madison, Microsoft, University of Lisbon and NYU.

Code availability To be made available post-publication.

A Theoretical analysis of the bug localization algorithm

Let \( I \) be the identity function that copies the \( \omega_{\text{pre}} \) or \( \omega_{\text{post}} \). Then,

\[
[\omega_0^f, \omega_1^f, \omega_2^f, \ldots, \omega_n^f] \leftarrow [I, \Delta_{1,1}, \Delta_{1,2}, \ldots \Delta_{1,n}](\omega_{\text{pre}}, P)
\]

\[
[\Omega_0^b, \Omega_1^b, \Omega_2^b, \ldots, \Omega_n^b] \leftarrow [\nabla_{n,1}, \nabla_{n,2}, \ldots \nabla_{n,n}, I](\omega_{\text{post}}, P)
\]

\[
[\Gamma_0, \Gamma_1, \Gamma_2, \ldots, \Gamma_n] \leftarrow [\omega_0^f, \omega_1^f, \omega_2^f, \ldots, \omega_n^f] \otimes [\Omega_0^b, \Omega_1^b, \Omega_2^b, \ldots, \Omega_n^b]; \Omega \text{ is a set of states while } \omega \text{ is a single state.}
\]

**Theorem 1** If a real bug is at location \( i \), it implies that \( i \) is in the suspicious set

**Proof** Let \( \hat{P} \) and \( P \) denote the correct and buggy programs, respectively.

**Assumption:** Assume that in the trace \( P_n \) (sequence of statements of length \( n \)), the \( i \)th statement has a real bug and \( LHS(s_i) \) be \( x \). Let us define the correct execution as the forward execution states of \( \hat{P} \) at each program point. Let \( \Delta_{0,j}(\omega_{\text{pre}}, \hat{P})[x] = n1, \Delta_{0,j}(\omega_{\text{pre}}, P)[x] = n2, \omega_{\text{post}}[x] = n3 \) and \( \omega_{\text{pre}}[x] = n0 \). Therefore, the number of nodes and points-to pairs in the datastructure, \( n \geq 2 \).

**Case 1** The \( i \)th statement is upward exposed: this is illustrated in Fig. 19a.

1. \( \Delta_{0,i-1}(\omega_{\text{pre}}, P) = \Delta_{0,i-1}(\omega_{\text{pre}}, \hat{P}) \) [Since the bug is in \( i \)th stmt]
2. \( \nabla_{n,i}(\omega_{\text{post}}, P)[x] = \omega_{\text{pre}}[x] \) [asgn\(_1\) rule in Fig. 12]
3. \( \Delta_{0,i-1}(\omega_{\text{pre}}, P)[x] = \omega_{\text{pre}}[x] \) [Lemma 1]
(a) Illustrations for case 1

Program P

\[ \bar{I}(pre)[x] = n_0 \]

\[ \bar{I}(\text{post})[x] = n_3 \]

Forward Execution

Buggy

\[ x = \ldots \]

Backward Execution

Subcase 1

\[ (0, i) \bar{1}[x] = n_0 \]

\[ (n, i-1) \]

\[ (i \bar{1}) [x] = n_0 \]

\[ (i) [x] = 0 \]

Non-zero gradient

Subcase 2

\[ (0, i) [x] = n_1 \]

\[ (n, i-1) \]

In all P states, \( x \rightarrow n_3 \) \( \Rightarrow \) \( (i) [x] > 0 \)

At least 1 state: \( x \not= n_1 \) \( \Rightarrow \) \( (i) [x] > 0 \)

(b) Illustrations for case 3

Program P

\[ \bar{I}(pre)[x] = n_0 \]

\[ \bar{I}(\text{post})[x] = n_3 \]

Forward Execution

Buggy

\[ x = \ldots \]

Backward Execution

\( \{s_1, s_2, \ldots \} : n^s \) states finally

Split for \( x \)

\[ (0, i \bar{1}) \]

\[ (n, i) [x] \]

P\( ^n \) states with \( x \) pointing to each node \( P \) times

\[ \bar{I}(i-1) = (n \bar{1})^s \ast p \]

Non-zero gradient

\[ (0, i) [x] = n_1 \]

\[ (n, i-1) \]

In all P states, \( x \rightarrow n_3 \) \( \Rightarrow \) \( (i) = n^s \ast p \)

Fig. 19 Illustrations for the proof of theorem 1
4. \( \nabla_{n,i}(\omega_{\text{post}}, P)[x] = \Delta_{0,i-1}(\omega_{\text{pre}}, P)[x] \) \[From 2 and 3\]
5. \( \Gamma_{i-1}[x] = 0 \) \[Since \( \Gamma_{i-1} = (\nabla_{n,i} - \Delta_{0,i-1}) \ast (\text{multiplying factor}) \) and \( \nabla_{n,i} - \Delta_{0,i-1} = 0 \)]
6. Also since \( i \)th statement is upward exposed there is no previous splitting of states when this statement is backtracked. \[Backward Semantics in Fig. 12\]
7. Without loss of generality, let\( s \) there are already \( p \) backtracked states before \( i \)th statement is backtracked.
8. Subcase 1: There has not been any splitting for \( x \) in the backward execution. Therefore for all the \( p \) states (in \( \nabla_{n,i+1} \)), \( x \) points-to \( n_3 \) \[Stmt is both upward and downward exposed\]
9. \( \Gamma_i[x] = p \ast \text{multiplying factor} > 0 \) \[From 8 and since \( \Gamma_i = (\nabla_{n,i+1} - \Delta_{0,i}) \ast (\text{multiplying factor}) \)]
10. Subcase 2: There has been a previous splitting for \( x \) in the backward execution. Therefore in atleast one of the \( p \) states, the value of \( x \neq n_1 \). This is because splitting happens uniformly across all nodes and there are atleast 2 nodes in the data-structure. \[Rules in Fig. 12\]
11. \( \Gamma_i[x] > 0 \) \[From 10\]
12. Hence for loop at step 5 in Algorithm 4, since \( \Gamma_{i-1} \neq \Gamma_i, i^{th} \) statement will be added to suspicious set.

**Case 2** The \( i \)th statement is sandwiched. We add all sandwiched statements to the suspicious set since the program loses all information regarding the correctness of such a statement.

**Case 3** The \( i \)th statement is downward exposed and not upward exposed: this is illustrated in Fig. 19b.

1. There will be splitting of states when \( i \)th statement is backtracked. \[asgn_2 \] rule in Fig. 12
2. Without loss of generality, let\( s \) there are \( p \) backtracked states before \( i \)th statement is backward executed.
3. In all these \( p \) states, the value of variable \( x = n_3 \). \[Lemma 2\]
4. Let there be \( s \) splits for \( x \) until the full backtrack of the program.
5. The final number of states after backtracking is \( n^s = M \), where \( n \) is the number of nodes and points-to pairs \[Rules in Fig. 12\]
6. Each of the \( p \) states contribute a difference of 1 in \( \Gamma_i = p \ast \frac{n^s}{p} = n^s \) \[\Gamma_i = (\nabla_{n,i+1} - \Delta_{0,i}) \ast (\text{multiplying factor})\]
7. \( \Gamma_{i-1} = \frac{n^s}{pm} \ast (n - 1) \ast p = n^s - n^{s-1} \) \[Out of \( n \) nodes, 1 will match with a node in \( \Delta_{0,i}(\omega_{\text{pre}}, P) \); uniform splitting\]
8. \( \Gamma_{i-1} - \Gamma_i = n^s - n^{s-1} - n^s \neq 0, \) for \( n > 1 \) \[Since number of nodes and points-to pairs > 1\]
9. Hence, for the loop at step 5 in Algorithm 4, since \( \Gamma_{i-1} \neq \Gamma_i, i^{th} \) statement will be added to suspicious set.

In all cases, Algorithm 4 successfully catches the buggy statement.

**Theorem 2** If \( C \) is the set of suspicious locations captured by Algorithm 4, then replacing all statements \( s \not\in C \) by \( lhs = \ast \) maintains the ground truth repair in the proof-guided repair algorithm.

**Proof** Any statement which depends on a buggy statement is also put in the suspicious set by our algorithm. This is natural because upon execution of the buggy statement, an incorrect
state is achieved, and any statements which further depend on this state would be buggy with respect to the ground truth state. Hence, we can slice away the statement from repair by making them non-deterministic.

References

1. Abreu R, Zoeteweij P, Golsteijn R, Van Gemund AJ (2009) A practical evaluation of spectrum-based fault localization. J Syst Softw 82(11):1780–1792
2. Ball T, Naik M, Rajamani SK (2003) From symptom to cause: localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL ’03, New York, NY, USA. ACM
3. Bavishi R, Pandey A, Roy S (2016) Regression aware debugging for mobile applications. In: Mobile! 2016: proceedings of the 1st international workshop on mobile development (invited paper), Mobile! 2016, New York, NY, USA., pp 21–22 Association for Computing Machinery
4. Bavishi R, Pandey A, Roy S (2016) To be precise: regression aware debugging. In: Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications, OOPSLA 2016, New York, NY, USA. ACM
5. Bendersky E. Pycparser: C parser in Python. https://pypi.python.org/pypi/pycparser. Accessed 24 Jan 2017
6. Bhattacharyya A, Kohli P, Singh R (2018) Neuro-symbolic program corrector for introductory programming assignments. In: Proceedings of the 40th international conference on software engineering, ICSE ’18, New York, NY, USA. Association for Computing Machinery
7. Chandra S, Torlak E, Barman S, Bodik R (2011) Angelic debugging. In: Proceedings of the 33rd international conference on software engineering, ICSE ’11, New York, NY, USA. ACM
8. Chatterjee P, Chatterjee A, Campos J, Abreu R, Roy S (2020) Diagnosing software faults using multiverse analysis. In: Bessiere C (ed) Proceedings of the twenty-ninth international joint conference on artificial intelligence, IJCAI-20, vol 7. International Joint Conferences on Artificial Intelligence Organization, pp 1629–1635. Main track
9. Chatterjee P, Roy S, Diep BP, Lal A (2020) Distributed bounded model checking. In: FMCAD, July 2020
10. Das R, Ahmed UZ, Karkare A, Gulwani S (2016) Prutor: a system for tutoring CS1 and collecting student programs for analysis. CoRR, arXiv:1608.03828
11. De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems, Berlin, Heidelberg. Springer-Verlag. pp 337–340
12. Demsky B, Rinard M (2003) Automatic detection and repair of errors in data structures. In: Proceedings of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA ’03, New York, NY, USA. ACM
13. Elkahkab B. Khurshid S (2008) Juizi: a tool for repairing complex data structures. In: Proceedings of the 30th international conference on software engineering, ICSE ’08, New York, NY, USA. ACM
14. Feser JK, Chaudhuri S, Dillig I (2015) Synthesizing data structure transformations from input–output examples. In: Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation, PLDI ’15, New York, NY, USA. ACM
15. Geeks for Geeks. Data structures. http://www.geeksforgeeks.org/data-structures/. Accessed 24 Jan 2017
16. Free Software Foundation. GDB Python API. https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html. Accessed 25 Jan 2017
17. Free Software Foundation. GDB: the GNU project debugger. https://sourceware.org/gdb/. Accessed 24 Jan 2017
18. Garg A, Roy S (2015) Sythesizing heap manipulations via integer linear programming. In: Blazy S, Jensen T (eds) Static analysis, SAS 2015. Proceedings. Springer, Berlin
19. Godefroid P, Klarlund N, Sen K (2005) DART: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, PLDI ’05, New York, NY, USA. ACM
20. Golia P, Roy S, Meel KS (2020) Manthan: a data-driven approach for Boolean function synthesis. In: Lahiri SK, Wang C (eds) Computer aided verification (CAV). Springer, Cham, pp 611–633
21. Golia P, Roy S, Slivovsky F, Meel KS (2021) Engineering an efficient Boolean functional synthesis engine. In: ICCAD
22. Groce A, Chaki S, Kroening D, Strichman O (2006) Error explanation with distance metrics. Int J Softw Tools Technol Transf 8(3):229–247
23. Grumberg O, Lerda F, Strichman O, Theobald M (2005) Proof-guided underapproximation-widening for multi-process systems. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL ’05, New York, NY, USA. ACM

24. Gulwani S, Radčić E, Zuleger F (2018) Automated clustering and program repair for introductory programming assignments. In: Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation, PLDI 2018, New York, NY, USA. Association for Computing Machinery, pp 465–480

25. Guo P (2013) Online python tutor: embeddable web-based program visualization for CS education. In: Proceeding of the 44th ACM technical symposium on computer science education, SIGCSE ’13, New York, NY, USA. ACM

26. Gupta R, Pal S, Kanade A, Shevade S (2017) Deepfix: fixing common C language errors by deep learning. In: Proceedings of the Thirty-First AAAI conference on artificial intelligence, AAAI’17. AAAI Press, pp 1345–1351

27. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580

28. Hu Q, Samanta R, Singh R, D’Antoni L (2019) Direct manipulation for imperative programs. In: Chang BY (ed) Static analysis. Springer, Cham, pp 347–367

29. Hu Y, Ahmed UZ, Mechtaev S, Leong B, Roychoudhury A (2019) Re-factoring based program repair applied to programming assignments. In: ASE ’19. IEEE Press, pp 388–398

30. igraph—the network analysis package. http://igraph.org/python/. Accessed 24 Jan 2017

31. Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: Proceedings of the 32Nd ACM/IEEE international conference on software engineering, ICSE ’10, New York, NY, USA, vol 1. ACM

32. Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering, ASE ’05, New York, NY, USA. ACM

33. Jose M, Majumdar R (2011) Bug-assist: assisting fault localization in ANSI-C programs. In: Proceedings of the 23rd international conference on computer aided verification, CAV’11, Berlin, Heidelberg. Springer-Verlag

34. Jose M, Majumdar R (2011) Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI ’11, New York, NY, USA. ACM

35. Kneuss E, Koukoutos M, Kuncak V (2015) Deductive program repair. In: CA V

36. Le Goues C, Nguyen T, Forrest S, Weimer W (2012) GenProg: a generic method for automatic software repair. IEEE Trans Softw Eng 38:54–72

37. Leung A, Sarracino J, Lerner S (2015) Interactive parser synthesis by example. SIGPLAN Not 50(6):565–574

38. Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI (2005) Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, PLDI ’05, New York, NY, USA. ACM

39. Liu C, Yan X, Fei L, Han J, Midkiff SP (2005) SOBER: statistical model-based bug localization. In: ESEC/FSE-13, New York, NY, USA. ACM

40. Liu Y, Li B (2010) Automated program debugging via multiple predicate switching. In: Proceedings of the twenty-fourth AAAI conference on artificial intelligence, AAAI’10. AAAI Press

41. Loncaric C, Ernst MD, Torlak E (2018) Generalized data structure synthesis. In: Proceedings of the 40th international conference on software engineering, ICSE ’18, New York, NY, USA. Association for Computing Machinery, pp 958–968

42. Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL ’16, New York, NY, USA. ACM

43. Malik MZ, Siddiqui JH, Khurshid S (2011) Constraint-based program debugging using data structure repair. In: 2011 fourth IEEE international conference on software testing, verification and validation, March 2011

44. Malik MZ, Ghori K, Elkarablieh B, Khurshid S (2009) A case for automated debugging using data structure repair. In: Proceedings of the 2009 IEEE/ACM international conference on automated software engineering, ASE ’09, USA. IEEE Computer Society

45. Mechtaev S, Yi J, Roychoudhury A (2015) DirectFix: looking for simple program repairs. In: Proceedings of the 37th international conference on software engineering, ICSE ’15, Piscataway, NJ, USA, vol 1, pp 448–458
46. Mechteev S, Yi J, Roychoudhury A (2016) Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th international conference on software engineering, ICSE ’16, New York, NY, USA. ACM
47. Modi V, Roy S, Aggarwal SK (2013) Exploring program phases for statistical bug localization. In: Proceedings of the 11th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, PASTE ’13, New York, NY, USA. ACM
48. Nguyen HD, Qi D, Roychoudhury A, Chandra S (2013) SemFix: program repair via semantic analysis. In: Proceedings of the 2013 international conference on software engineering, ICSE ’13, Piscataway, NJ, USA. IEEE Press
49. Nguyen T-T, Ta Q-T, Chin W-N (2019) Automatic program repair using formal verification and expression templates. In: VMCAI
50. Nguyen T, Weimer W, Le Goues C, Forrest S (2009) Using execution paths to evolve software patches. In: International conference on software testing, verification and validation workshops, 2009. ICSTW’09, IEEE, pp 152–153
51. Pandey A, Kotcharakota PR, Roy S (2019) Deferred concretization in symbolic execution via fuzzing. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2019, New York, NY, USA. Association for Computing Machinery, pp 228–238
52. Pham V-T, Khurana S, Roy S, Roychoudhury A (2017) Bucketing failing tests via symbolic analysis. In: Huismans M, Rubin J (eds) Fundamental approaches to software engineering. Springer, Berlin, pp 43–59
53. Polikarpova N, Sergey I (2019) Structuring the synthesis of heap-manipulating programs. Proc ACM Program Lang 3:1–30
54. Pu Y, Narasimhan K, Solar-Lezama A, Barzilay R (2016) Sk_p: a neural program corrector for moocs. In: Companion proceedings of the 2016 ACM SIGPLAN international conference on systems, programming, languages and applications: software for humanity, SPLASH Companion 2016, page 39-40, New York, NY, USA. Association for Computing Machinery
55. Rolim R, Soares G, D’Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B (2017) Learning syntactic program transformations from examples. In: Proceedings of the 39th international conference on software engineering, ICSE ’17. IEEE Press
56. Roy S, Hsu J, Albarghouthi A (2021) Learning differentially private mechanisms. In: 2021 IEEE symposium on security and privacy (SP), Los Alamitos, CA, USA, May 2021. IEEE Computer Society, pp 852–865
57. Roy S (2013) From concrete examples to heap manipulating programs. In: Logozzo F, Fähndrich M (eds) Static analysis: 20th international symposium, SAS 2013, Seattle, WA, USA, 20–22 June 2013. Proceedings. Springer, Berlin
58. Roy S, Pandey A, Dolan-Gavitt B, Hu Y (2018) Bug synthesis: challenging bug-finding tools with deep faults. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018, New York, NY, USA. Association for Computing Machinery, pp 224–234
59. Sen K, Marinov D, Agha G (2005) CUTE: a concolic unit testing engine for C. In: ESEC/FSE-13, New York, NY, USA. ACM
60. Singal D, Agarwal P, Jhunjhunwala S, Roy S (2018) Parse condition: symbolic encoding of LL(1) parsing. In: Barthe G, Sutcliffe G, Veanes M (eds) LPAR-22. 22nd international conference on logic for programming, artificial intelligence and reasoning. EPiC series in computing, vol 57. EasyChair, pp 637–655
61. Singh R, Gulwani S, Solar-Lezama A (2013) Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation, PLDI ’13, New York, NY, USA. Association for Computing Machinery
62. Singh R, Solar-Lezama A (2011) Synthesizing data structure manipulations from storyboards. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, ESEC/FSE ’11, New York, NY, USA. ACM
63. Tan SH, Yoshida H, Prasad MR, Roychoudhury A (2016) Anti-patterns in search-based program repair. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, New York, NY, USA. Association for Computing Machinery, pp 727–738
64. Verma A, Kalita PK, Pandey A, Roy S (2020) Interactive debugging of concurrent programs under relaxed memory models. In: Proceedings of the 18th ACM/IEEE international symposium on code generation and optimization, CGO 2020, New York, NY, USA. Association for Computing Machinery, pp 68–80
65. Verma S, Roy S (2017) Synergistic debug-repair of heap manipulations. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017, New York, NY, USA. ACM
66. Wang K, Singh R, Su Z (2017) Dynamic neural program embedding for program repair. arXiv, arXiv:1711.07163
67. Wang K, Singh R, Su Z (2018) Search, align, and repair: data-driven feedback generation for introductory programming exercises. In: Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation, PLDI 2018, New York, NY, USA. Association for Computing Machinery pp 481–495

68. Weimer W (2006) Patches as better bug reports. In: Proceedings of the 5th international conference on generative programming and component engineering, GPCE ’06, New York, NY, USA. ACM

69. Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: Proceedings of the 31st international conference on software engineering, ICSE ’09, Washington, DC, USA. IEEE Computer Society

70. Zeller A (2002) Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT symposium on foundations of software engineering, SIGSOFT ’02/FSE-10, New York, NY, USA. ACM

71. Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200

72. Zimmermann T, Zeller A (2002) Visualizing memory graphs. In: Diehl S (ed) Revised lectures on software visualization, international seminar. Springer-Verlag, London

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.