International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-026-00844-2

GENERAL

Special Issue CSV 2025

LLMs and fuzzing in tandem: a new approach to automatically generating weakest preconditions

Daragh King[1] · **Vasileios Koutavas**[1] · **Laura Kovács**[2]

Accepted: 20 February 2026

Abstract

The weakest precondition (WP) of a program describes the largest set of initial states from which all terminating executions of the program satisfy a given postcondition. The generation of WPs is an important task with practical applications in areas ranging from verification to run-time error checking. This paper proposes the combination of Large Language Models (LLMs) and fuzz testing for generating WPs. In pursuit of this goal, we introduce *Fuzzing Guidance* (FG); FG acts as a means of directing LLMs towards correct WPs using program execution feedback. FG utilises fuzz testing for approximately checking the validity and weakness of candidate WPs, this information is then fed back to the LLM as a means of context refinement. We demonstrate the effectiveness of our approach on a comprehensive benchmark set of deterministic array programs in Java. Our experiments indicate that LLMs are capable of producing viable candidate WPs, and that this ability can be practically enhanced through FG.

Keywords Weakest precondition · Fuzzing · Correctness · LLMs · Verification

1 Introduction

Pre- and postconditions are fundamental formalisms for specifying program behaviour, originating from the seminal work of Floyd [1] and Hoare [2]. Beyond Hoare-logic itself, these formalisms are a cornerstone in many software engineering, verification, and debugging techniques; these include design-by-contract [3], abstract interpretation [4], program refinement [5], symbolic execution [6], and run-time assertion checking [7].

The postcondition of a program method specifies the desired property of the program state upon termination. On the other hand, the precondition specifies the required property of the initial state for the method to behave as intended.[1] Knowing the pre- and postconditions of a method suffice to ensure its correctness to external calling methods – thereby enabling compositional program reasoning.

There can be many pre- and postconditions for a given method m, but finding the *right* ones is crucial for effective compositional reasoning. If m's precondition is too permissive (allowing too many initial states) or its postcondition too restrictive (rejecting too many final states), reasoning about m's behaviour may fail. Conversely, if m's precondition is too restrictive or postcondition too permissive, reasoning about methods calling m may fail.

This need for precision has led to the notion of a *weakest precondition* (WP), i.e. the most permissive precondition for which m's executions lead to final states satisfying its postcondition.[2] The weakest precondition calculus [8] was the first manual method for deriving the weakest precondition from a method's code and desired postcondition. Subsequent techniques have sought to automatically derive preconditions, either sufficient (though often not weakest) [9–23] or necessary ones [24]. However, since the problem

✉ D. King
kingd6@tcd.ie

V. Koutavas
vasileios.koutavas@tcd.ie

L. Kovács
laura.kovacs@tuwien.ac.at

[1] School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland

[2] Faculty of Informatics, TU Wien, Vienna, Austria

[1] In total correctness specifications, a precondition also guarantees termination of the method; in this paper we restrict our attention to programs that terminate for any input.

[2] And its dual, the strongest postcondition.

Published online: 12 March 2026

is undecidable in the presence of loops, user intervention remains an essential fallback.

Studies and tools regarding the generation of WPs (and their requisite loop invariants) typically reside on the more mathematical side of computer science, relying on formal methods. On the other end of the spectrum lie more heuristic-based, data driven fields, such as fuzz testing [25] and machine learning [26]. Fuzz testing generates pseudo random inputs for a program under test in an attempt to trigger buggy or unwanted behaviour. Fuzz testing has seen far-reaching success [27], and is actively deployed in industry [28]. While fuzz testing has grown in popularity, machine learning has given rise to LLMs [29].

In recent years, LLMs have found extensive and successful applications to software engineering, including code generation [30], code repair [31] and invariant inference [32]. Although LLMs suffer from inherent drawbacks (e.g. stochastic outputs, hallucinations, etc.), it is natural to question the extent to which these techniques can contribute to the inherently manual process of WP generation.

This paper broaches a *new frontier in the automatic generation of WPs*. First, we employ LLMs to act as generators for candidate WPs. Second, we introduce Fuzzing Guidance (FG) as a means for refining these candidate WPs and steering LLMs towards generating correct WPs. FG relies upon two distinct phases of fuzz testing to validate and ensure the weakness of an LLM-generated WP; the results of these fuzzing phases are then fed back to the LLM as a means of context refinement. The benefits of this combined approach are:

- *Superior performance in WP generation*: The combination of LLMs and FG can greatly improve the quality/correctness of generated WPs when compared to zero-shot prompting; this is especially true for (typically) cheaper non-reasoning models (e.g. GPT-4o [33]), enabling them to achieve comparable performance to their (typically) more expensive reasoning counterparts (e.g. O4-mini [34]).
- *Increased scope*: Unlike previous works in WP inference that rely on formal methods, our use of LLMs for generating WPs has no inherent limitation in the programs it can handle. This increases the potential scope and use of WPs in practice. Indeed, our experiments include programs which fall outside the scope of previous work – these include variants of problems revolving around the use of binary search and sorting (see the supplementary material link for example-four).
- *Model-driven evolution*: Although the general performance progress of LLMs is not guaranteed, whatever evolutions do occur in the future will be readily reflected in our approach. Relying on LLMs enables this background-improvement to materialise in our method for WP generation.

Section 2 describes the methodology of FG; outlining a high-level structure of the programs used, the prompts utilised, and the constituent phases of fuzz testing. Following this, Sect. 3 depicts a motivating example; here, GPT-4o and FG interact to produce the correct WP for a particular program.

Section 4 details our experimental analysis. We introduce the four benchmark sets of Java programs used in our work, and outline the LLM-FG configurations utilised for generating WPs. Detailing our results, we explore if LLMs can generate sensible WP candidates and whether FG can aid them in doing so by enabling iterative phases of context refinement.

Proceeding to the tail-end of the paper, we provide a discussion of our work in Sect. 5 – detailing the relevant threats to validity and the limitations of our approach. Section 6 overviews related work. Section 7 outlines future directions, and Sect. 8 concludes.

2 Method

This section outlines our methodology for the generation of Weakest Preconditions (WPs) using LLMs and Fuzzing Guidance (FG). The overarching workflow for this process can be seen in Fig. 1 – the fuzzing phases and prompts depicted are detailed in the Sect. 2.2 and Sect. 2.3 respectively.

2.1 Input programs

The input programs considered for this work follow a common high-level structure (an example can be seen in Fig. 5

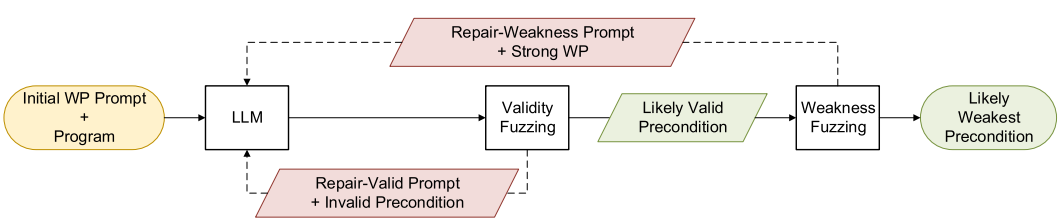

Fig. 1 Fuzzing Guidance Workflow

in Sect. 3). Specifically, the programs considered are implemented in Java and feature a deterministic looping computation over arrays; these programs consist of:

- A single 'static int' method named 'foo' which accepts three arrays a, b, c as inputs.
- A postcondition implemented in code which checks a desired property for some subset of the input arrays; an exception is thrown if the postcondition is not satisfied.
- A successful return value of 0.

The looping computations contained within each program range over a varied set of tasks, e.g. array copying, arithmetic, sorting, search, etc. For example, the program in Fig. 5 of Sect. 3, copies elements of array a into array b.

Additionally, it is worth noting that each program's single method is named 'foo' to prevent any unnecessary direction being given to the LLM being prompted.

2.2 Fuzzing guidance phases

FG utilises the JQF fuzz testing tool [35] to carry out two distinct phases of fuzz testing for candidate WPs. These phases are termed *validity-fuzzing* and *weakness-fuzzing*; the outputs of which act as feedback to the LLM being utilised for WP generation.

2.2.1 Validity-fuzzing

This phase of fuzzing is employed to help check that a candidate WP is valid. A candidate WP is valid if all initial program states which satisfy the WP wind up satisfying the program's given postcondition after it is executed.

Validity-fuzzing attempts to generate pseudo-random initial states which adhere to a candidate WP, but which do **not** satisfy a program's postcondition; i.e. they cause the program to throw an exception. If such an input is found then we can conclude that the candidate WP is not valid; otherwise we can posit that the WP is *likely-valid*.

2.2.2 Weakness-fuzzing

Given a WP that is already deemed to be likely-valid, weakness-fuzzing attempts to ensure that the same WP is *likely-weakest*. A candidate WP is weakest if it accounts for all initial program states which result in the program's postcondition being satisfied after it is executed.

Weakness-fuzzing attempts to generate pseudo-random initial states which do *not* adhere to a candidate WP, but which do satisfy a program's postcondition; i.e. no exception is thrown. If such an input is found then we can conclude that the candidate WP is not weakest; otherwise we can posit that the WP is likely-weakest.

```
Context: You have the following Java program containing a method named 'foo'.
You must determine the weakest precondition for 'foo' so that:
    - The method's postcondition is always satisfied.
    - No exceptions or errors are reachable.

Task A: Identify this weakest precondition.
Task B: Create a public Java method named 'precondition' that checks this
weakest precondition.

Adhere to the following requirements:
    - It must accept the same arguments as 'foo'.
    - It must return 'false' if the precondition is not satisfied and 'true'
    otherwise.
    - It may include a for loop if necessary.
    - Before this method, provide a single brief comment describing the
    precondition in pseudo-code.
    - Integrate this 'precondition' method into the original Java program and
    return the complete updated program.

Output Format: Return only the updated Java program (do not include any other
additional text), which includes
    - The original 'foo' method - this should be unchanged.
    - The newly added 'precondition' method.

Provide no additional explanations beyond the program code and the required
comment. Reason through your solution internally.
```

Fig. 2 LLM prompt to generate WPs, i.e. the initial-WP prompt

```
You have a Java program that contains:
    - A method called 'foo', which may reach an error state under certain
    inputs.
    - A method called 'precondition', intended to represent the weakest
    precondition for 'foo'.

The current implementation of 'precondition' is incorrect. Specifically, there
is a known set of input values for which 'precondition' returns 'true'
(indicating the precondition is satisfied), but passing these same inputs to
'foo' actually triggers an error state. The aforementioned inputs follow:
[Insert Inputs]

Task:
    1. Analyze the existing 'foo' method and the incorrect 'precondition'
    method.
    2. Determine why the current 'precondition' method fails to exclude the
    problematic inputs.
    3. Rewrite or adjust 'precondition' so that it correctly represents the
    true
    weakest precondition for 'foo', ensuring that when 'precondition' returns
    'true', 'foo' will not reach an error with the same inputs.

Goal:
Provide the corrected 'precondition' method so that 'foo' never encounters an
error when the corrected 'precondition' returns 'true'.

Output Format: Return only the updated Java program (do not include any other
additional text), which includes
    - The original 'foo' method - this should be unchanged.
    - The corrected 'precondition' method.

Provide no additional explanations beyond the program code and the required
comment. Reason through your solution internally.
```

Fig. 3 LLM prompt to repair or replace invalid WPs, i.e. the repair-validity prompt

2.3 Prompting

Our method for generating WPs relies on three underlying prompts for LLM interactions; these will be referred to as the *initial-WP prompt* (Fig. 2), the *repair-validity prompt* (Fig. 3), and the *repair-weakness prompt* (Fig. 4).

The initial-WP prompt tasks the LLM with WP generation for a particular program. In isolation (i.e. when FG is not employed) this prompting method can be viewed as a zero-shot prompting strategy [36]. The outcomes to this zero-shot strategy are revealed through the results for the *GPT-4o* and *O4-mini* experimental configurations in Sect. 4.

```
You have a Java program that contains:
    - A method called 'foo', which may reach an error state under certain
    inputs.
    - A method called 'precondition', intended to represent the weakest
    precondition for 'foo'.

The current implementation of 'precondition' is incorrect. Specifically, there
is a known set of input values for which 'foo' returns successfully and
'precondition' returns 'false' - indicating that the inputs do not satisfy the
given precondition, but still result in a succesful execution of 'foo'.
This implies that the precondition defined by the 'precondition' method may
not be weakest. The aforementioned inputs follow:
[Insert Inputs]

Task:
    1. Analyze the existing 'foo' method and the incorrect 'precondition'
    method.
    2. Determine why the current 'precondition' method fails to correctly
    account for the above inputs.
    3. Rewrite or adjust 'precondition' so that it correctly represents the
    true weakest precondition for 'foo'.

Goal:
Provide the corrected 'precondition' method so that 'foo' never encounters
an error when the corrected 'precondition' returns 'true', and that the
corrected 'precondition' method truly represents the weakest precondition for
'foo'.

Output Format: Return only the updated Java program (do not include any other
additional text), which includes
    - The original 'foo' method - this should be unchanged.
    - The corrected 'precondition' method.

Provide no additional explanations beyond the program code and the required
comment. Reason through your solution internally.
```

Fig. 4 LLM prompt to repair or replace strong WPs, i.e. the repair-weakness prompt

The repair-validity and repair-weakness prompts expect to deal with an already existing candidate WP. The repair-validity prompt provides the LLM with inputs which highlight that the current candidate WP is not valid (as derived through validity-fuzzing, Sect. 2.2.1); the LLM is then tasked with repairing the current WP or replacing it with a newly generated WP. Similarly, the repair-weakness prompt provides the LLM with inputs which highlight that the current candidate WP is not the weakest possible (as derived through weakness-fuzzing, Sect. 2.2.2); the same task of WP repair or generation is then also put forth to the LLM.

Besides intention and functionality, there are additional aspects to our prompts which are worth discussing. Each of our prompts ends with 'Reason through your solution internally'. This is done to encourage the LLM to utilise a chain-of-thought [36] process that does not reveal these 'thoughts' in the corresponding output; this makes the parsing of LLM's responses more straightforward. One might argue that this restriction on the LLM's outputs might affect how they can be interpreted; we mitigate this issue through the use of the following phrase in our initial-WP prompt – 'provide a single brief comment describing the precondition in pseudo-code'. This addendum ensures that 1) there is an easily-read reasoning behind the LLM's response, and 2) there exists a counter-point to the code generated when it does not align with expectations.

2.4 Fuzzing guidance cycles

The outputs derived from the individual validity- and weakness-fuzzing phases (Sect. 2.2) are combined with their respective repair prompts (Sect. 2.3) in an effort to steer the LLM towards generating higher-quality candidate WPs in terms of validity and weakness respectively. This describes the essence of FG, but a little more needs to be said when considering how it is used in practice – we must define how the two phases of FG combine to define an FG-cycle. An FG-cycle consists of:

1. A maximum of X iterations of validity-fuzzing and repair.
2. A single iteration of weakness-fuzzing and repair.

Step 2 follows the successful completion of Step 1 (cf. Fig. 1). Multiple FG-cycles can be used in practice to generate WPs.

```
public static int foo(int[] a, int[] b, int[] c) {
    int N = a.length;
    if (N == 0 || b.length != N) {
        throw new RuntimeException();
    }

    int i = 0;
    while (i < N) {
        b[i] = a[i];
        i++;
    }

    int[] b_clone = b.clone();
    sort(b_clone);

    i = 0;
    while (i < N) {
        if (b[i] != b_clone[i])
            throw new RuntimeException();
        i++;
    }

    return 0;
}
```

Fig. 5 Example program P from the 'Sorting' benchmark set (Sect. 4.1)

3 Motivating example

This section outlines the generation of a correct weakest precondition using an LLM (GPT-4o [33]) and Fuzzing Guidance (FG). We will do so for an example program P (Fig. 5) from our 'Sorting' benchmark set (Sect. 4.1). (See the supplementary material link for example-one.)

Before proceeding, it is worth re-noting that the comments in the following code snippets are generated by the LLM in question (Sect. 2.3).

3.1 Initial weakest precondition generation

First, we prompt the LLM to generate its initial candidate WP by appending program P to the initial-WP prompt (Fig. 2). This results in the following 'precondition' method being generated (i.e. the initial candidate WP) in Fig. 6.

```
// Precondition: 'a' and 'b' must have the same
// non-zero length,
// and 'c' should be at least of length 'a'.
public static boolean precondition
(int[] a, int[] b, int[] c) {
    return a.length > 0 && a.length == b.length
    && c.length >= a.length;
}
```

Fig. 6 Initial candidate WP generated for P (Fig. 5)

3.2 Ensuring validity

Given the initial candidate WP generated (Fig. 6) for program P, we can now proceed to checking its validity using validity-fuzzing (Sect. 2.2.1). Validity-fuzzing quickly highlights an issue with the given candidate; the following inputs satisfy the candidate WP (i.e. 'precondition' in Fig. 6 returns true) but ultimately result in a failing execution of P.

```
int[] a = {1401919545, 267222264, 1358618764};
int[] b = {739288021, 776171394, -594227544};
int[] c = {427235608, -506959485, 1997723943,
          -498207265, 733576341};
```

The program P, along with the invalid candidate WP and the preceding inputs, are integrated into the repair-validity prompt (Fig. 3); using this prompt the LLM is tasked with amending the invalid candidate WP. This results in Fig. 7. Upon inspection, this 'precondition' method appears to be a valid candidate for P's WP; another round of validity fuzzing provides evidence that this is so. Now, we must attempt to see if this candidate WP is likely-weakest.

3.3 Ensuring weakness

Through validity-fuzzing, we now have a candidate WP (Fig. 7) that we believe to be a valid precondition for the program P. Our goal now is to check the weakness of the candidate WP with weakness-fuzzing (Sect. 2.2.2). Weakness-fuzzing proceeds to highlight a problem with the current candidate WP; the following inputs do *not* satisfy the WP (i.e. i.e. 'precondition' in Fig. 7 returns false) but result in a successful execution of P:

```
int[] a = {-1286467063};
int[] b = {0};
int[] c = {};
```

By accounting for the preceding inputs and inspecting P again, we can see that the restriction on the input array c in the current candidate WP (Fig. 7) is unnecessary. Therefore, through the processes of validity-fuzzing and weakness-fuzzing, we can determine that our current candidate WP is likely a valid precondition for P, but that it is demonstrably not the weakest (i.e. it is too strong) because of the unnecessary condition on the length of c.

The program P, along with the strong candidate WP and the preceding weakness-fuzzing outputs, are integrated into the repair-weakness prompt (Fig. 4); this prompt tasks the LLM with amending the strong WP – resulting in the candidate WP of Fig. 8.

Immediately we can observe that there appears to be something amiss with the latest candidate WP (Fig. 8). It appears simpler than the candidate in Fig. 7, and more akin to the invalid candidate depicted in Fig. 6 (although it does do away with the unnecessary restraints on the array c mentioned previously). We must undergo validity fuzzing again to ensure that the quality of the candidate WP has not regressed.

```
// Precondition: 'a' and 'b' must have the same
// non-zero length,
// 'c' should be at least of length 'a',
// and 'a' must be sorted in non-decreasing order
// to ensure 'b' remains sorted.
public static boolean precondition
(int[] a, int[] b, int[] c) {
    if (a.length == 0
    || a.length != b.length
    || c.length < a.length) {
        return false;
    }

    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i + 1]) {
            return false;
        }
    }

    return true;
}
```

Fig. 7 A likely-valid but too-strong candidate WP for P (Fig. 5)

```
// Precondition: 'a' and 'b' must have the same
// non-zero length,
// and 'c' must have a length that is
// irrelevant to the execution success of 'foo'.
// Additionally, 'a' naturally
// becomes sorted when it has identical elements.
public static boolean precondition
(int[] a, int[] b, int[] c) {
    return a.length > 0 && a.length == b.length;
}
```

Fig. 8 A step backwards for FG? Another candidate WP for P (Fig. 5)

```
// Precondition: 'a' and 'b' must have the same
// non-zero length,
// and 'a' must be sorted in non-decreasing order,
// since 'b' is a direct clone of 'a' before
// sorting the clone.
public static boolean precondition
(int[] a, int[] b, int[] c) {
    if (a.length == 0 || a.length != b.length) {
        return false;
    }

    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i + 1]) {
            return false;
        }
    }

    return true;
}
```

Fig. 9 A likely-valid and -weakest WP for P (Fig. 5)

3.4 Ensuring validity and weakness again

The current candidate WP (Fig. 8) must undergo validity-fuzzing again. As before in Sect. 3.2, validity-fuzzing finds a set of inputs which highlight the invalidity of the current candidate (these inputs are omitted for brevity). Once again, the repair of the current candidate WP must take place with the repair-validity prompt. This process results in the following candidate WP:

Given the latest candidate WP in Fig. 9, validity-fuzzing and weakness-fuzzing both succeed in succession; therefore this candidate is likely-valid and likely-weakest. Through manual inspection we can see that this is, in fact, the correct WP for program P.

3.5 Conclusion

Overall, we can see from this example how FG can enable the generation of correct WPs. Through several bouts of back-and-forth between LLM interations and different phases of fuzz testing, we arrived at a candidate WP that is correct.

3.6 Additional remarks

Notice that P (Fig. 5) calls an external method 'sort', and that no additional information was provided to the LLM about this method. Despite this, the approach taken here has arrived at the correct WP for P – from this, we can conclude that LLMs can reason about unknown but well-understood methods like 'sort'.

The above phenomena is of practical benefit in terms of prompting, but there is also a danger of missing important information regarding the behaviour of these unspecified method calls. If such a concern exists, one could augment the code with the pre- and post-conditions of these methods, allowing the LLM reason more robustly in a compositional manner.

4 Evaluation of weakest precondition generation

The core focus of this work is to deploy LLMs and fuzz testing towards the task of generating weakest preconditions (WPs); the method employed here is outlined in detail in Sect. 2 – in brief, LLM prompting for WP generation is augmented with Fuzzing Guidance (FG) in order to steer the LLM toward generating correct WPs. In this section we outline our results pertaining to this goal, comparing the results of WP generation with and without FG. In assessing and presenting these results we are primarily concerned with the three following criteria: *Correctness*, *Stability*, and the *Contribution of FG*.

Correctness: for a particular program, the correctness of a generated WP is defined in relation to the program's ground-truth WP. These ground-truth WPs are known a priori (through manual derivation) and the comparison with generated WPs is performed manually; if a generated WP is deemed semantically equivalent to its ground-truth counter part then it is correct.

Stability: we complete k iterations of WP generation for each benchmark set and experimental configuration. This allows us to look for consistency in the WPs generated and assess the stability of both the LLM outputs and FG usage.

Contribution of FG: as described in Sect. 2, FG attempts to steer the LLM towards generating correct WPs through program execution feedback. We are interested in observing how and when FG is deployed, and whether or not it can meaningfully guide the LLM in the aforementioned goal.

These criteria will be discussed in the per-configuration performance summaries provided in Sect. 4.4.

4.1 Benchmark sets

This work utilises four distinct sets of benchmark programs. Each program within these sets is implemented in Java and follows the high-level program structure outlined in Sect. 2.1. The following points will give a brief overview of each set:

- 'Existential' benchmarks: this set contains 20 programs; the correct WPs for the programs of this benchmark set require existential quantification over the array content. For example, a description of a correct WP for some program P in this set might look like – '*there exists* a valid index i for the input array a such that $a[i] = 100$'.
- 'Universal' benchmarks: this set contains 36 programs; the correct WPs for the programs of this benchmark set require universal quantification over the array content. For example, a description of a correct WP for some program P in this set could look like – '*for all* valid indices i of the input array b, $b[i] = 2i$'.
- 'Sorting' benchmarks: this set contains 8 programs. The computations performed in each program feature sorting in some manner. For example, the program in Fig. 5 (of Sect. 3) features a sorted array in its postcondition.
- 'Search' benchmarks: this set contains 6 programs; each of which feature binary-search over an input array. Again, these examples express properties with both existential and universal quantification over the array content.

4.2 Experimental set up and configurations

The set up for our experimental evaluations can be described through the following features:

- LLMs: We employ GPT-4o [33] and O4-mini [34] from OpenAI. These represent performant and accessible LLMs

from the categories of non-reasoning and reasoning models respectively.

- Benchmark Sets: The four sets of benchmark programs used are described in Sect. 4.1. The structure of these programs follows the description outlined in Sect. 2.1.
- Number of Iterations k: This defines the number of times WP-generation occurs for each program in each benchmark set. We fix $k = 5$ for all of the experiments described.
- Fuzzing Parameters: These parameters influence how FG is deployed. We allow a maximum of 10 attempts of validity-fuzzing (Sect. 2.2.1) for each FG-cycle (Sect. 2.4). We also set a maximum fuzzing time of $10s$ for each validity- and weakness-fuzzing (Sect. 2.2.2) invocation.

Overall, the four LLM and FG configurations investigated are: GPT-4o without FG (*GPT-4o*), GPT-4o with FG (*GPT-4o-FG*), O4-mini without FG (*O4-mini*), and O4-mini with FG (*O4-mini-FG*). The labels in parentheses correspond to the configuration labels in both the plots of Figs. 10, 11, 12, 13 and Table 1.

4.3 Description of the results table and figures

Before delving into the summaries of the obtained results, this section provides a brief descriptive aid to the figures and tables referenced in Sect. 4.4.

4.3.1 Tables

The results of Table 1 depict a set of performance statistics obtained from $k = 5$ iterations of evaluations. The majority of the tables headings are self-explanatory, though we believe some clarification regarding 'FG Usage' and 'FG Success' is warranted.

The 'FG Usage' statistics denote the min, max, and average number of programs where FG was employed in an attempt to direct the LLM in the task of WP generation. For example, on the 'Universal' benchmark set, *GPT-4o-FG* employed FG for a minimum of 6 programs, a maximum of 10 programs, and an average of 7.8 programs per iteration.

The 'FG Success' statistics denote the min, max, and average number of programs where FG was employed *successfully* – i.e. where FG enabled the generation of a correct WP. For example, on the 'Existential' benchmark set, *GPT-4o-FG* employed FG successfully for a minimum of 5 programs, a maximum of 6 programs, and an average of 5.6 programs per iteration.

From this table alone, we can already conclude that the use of FG has led to a marked improvement in the generation of correct WPs; this is true across all benchmark sets and LLMs used. For GPT-4o (a non-reasoning model), FG provides the greatest improvements in correctness; while in the case of O4-mini (a reasoning model), FG has enabled the achievement of a perfect correctness score across every iteration.

Table 1 Resulting statistics for each benchmark set and configuration, obtained across $k = 5$ iterations

Configuration	Benchmark	# Programs	# Correct WPs Min–Max (Avg., Avg.%)	FG Usage Min–Max (Avg.)	FG Success Min–Max (Avg.)
GPT-4o	Existential	20	10 – 12 (11, 55.00%)	–	–
	Universal	36	24 – 27 (26.2, 72.78%)	–	–
	Sorting	8	2 – 4 (2.8, 35.00%)	–	–
	Search	6	4 – 5 (4.2, 70.00%)	–	–
GPT-4o-FG	Existential	20	16 – 17 (16.6, 83.00%)	5 – 7 (6)	5 – 6 (5.6)
	Universal	36	32 – 34 (33, 91.67%)	6 – 10 (7.8)	6 – 9 (6.8)
	Sorting	8	4 – 6 (5.4, 67.50%)	4 – 6 (4.6)	2 – 4 (2.6)
	Search	6	5 – 6 (5.8, 96.67%)	1 – 2 (1.8)	1 – 2 (1.6)
O4-mini	Existential	20	18 – 19 (18.6, 93.00%)	–	–
	Universal	36	34 – 35 (34.8, 96.67%)	–	–
	Sorting	8	7 – 8 (7.8, 97.50%)	–	–
	Search	6	5 – 6 (5.4, 90.00%)	–	–
O4-mini-FG	Existential	20	20 – 20 (20, 100%)	1 – 2 (1.4)	1 – 2 (1.4)
	Universal	36	36 – 36 (36, 100%)	1 – 2 (1.2)	1 – 2 (1.2)
	Sorting	8	8 – 8 (8, 100%)	0 – 1 (0.2)	0 – 1 (0.2)
	Search	6	6 – 6 (6, 100%)	0 – 1 (0.6)	0 – 1 (0.6)

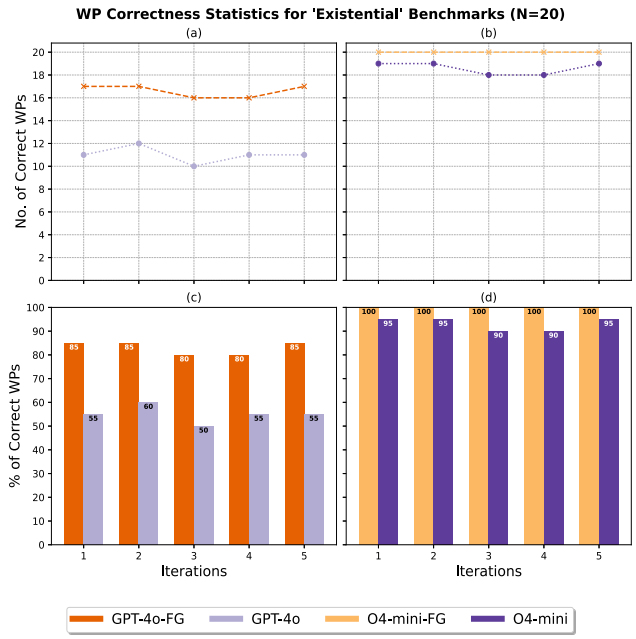

Fig. 10 WP correctness statistics for the 'Existential' benchmark set, in absolute and relative terms

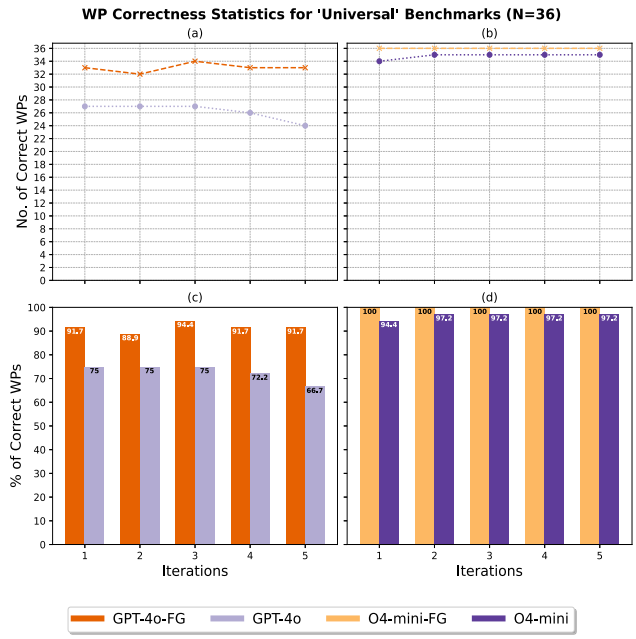

Fig. 11 WP correctness statistics for the 'Universal' benchmark set, in absolute and relative terms

4.3.2 Figures

Figures 10, 11, 12, 13 depict the correctness results obtained for each benchmark set and iteration. Each of these figures contains four plots (a)-(d).

Plots (a) and (b) outline the number of correct WPs generated; these line graphs can aid us in gleaning information

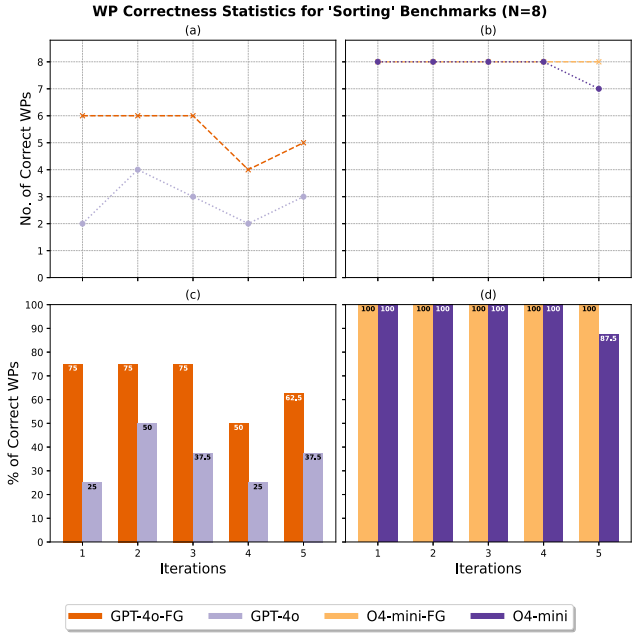

Fig. 12 WP correctness statistics for the 'Sorting' benchmark set, in absolute and relative terms

about the correctness results achieved per iteration and observe whether or not they are stable. For example, consider the line graph for *GPT-4o-FG* in plot (a) of Fig. 11, this shows us that the number of correct WPs generated by this config on the 'Universal' benchmark set hovers between 32 and 34 (out of a possible 36) – indicating result stability.

Plots (c) and (d) portray the same correctness information in terms of percentages; these bar charts enable us to more clearly see the relative performance differences between the experimental configurations. For example, observe the bar chart for *O4-mini* in plot (d) of Fig. 10, this shows us that the percentage of correct WPs generated by *O4-mini* on the 'Existential' benchmark set always underperforms *O4-mini-FG* in each iteration.

4.4 Result summaries per configuration

Resulting statistics for each benchmark set and experimental configuration can be seen in Table 1. The following provides a summary of these results, along with some observations – Table 1 is implicitly referenced throughout.

4.4.1 GPT-4o without FG (*GPT-4o*)

This configuration resulted in the lowest average correctness results for all benchmark sets. We can also observe a wide variance here – with an average correctness percentage of 35% and 72.78% respectively, *GPT-4o* performs poorly at attaining the correct WPs for the 'Sorting' benchmarks while performing relatively well on the 'Universal' benchmarks.

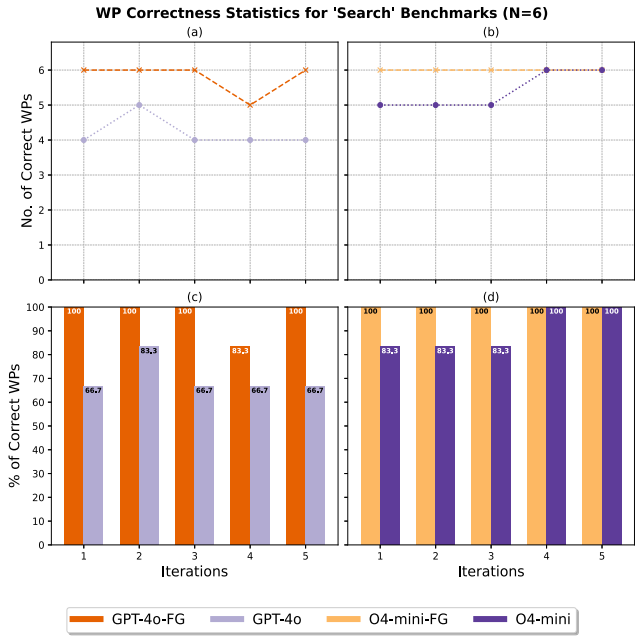

Fig. 13 WP correctness statistics for the 'Search' benchmark set, in absolute and relative terms

There is also evidence to suggest that the results of this set-up are consistent; this can be seen in the relative similarity of the min and max number of correct WPs generated per benchmark set – as a result, we can conclude that *GPT-4o* is consistently *incorrect* also. This notion of consistent incorrectness is also evident in plots (a) of Fig. 10 and Fig. 11 as the line-graphs appear relatively stable. When we look at the same plots in Fig. 12, and Fig. 13, we can observe a more erratic performance (exacerbated by the small size of the corresponding benchmark sets).

4.4.2 GPT-4o with FG (*GPT-4o-FG*)

Through the use of FG we can see how the performance of the base-line *GPT-4o* config can be significantly improved upon. This is made evident by the fact that *GPT-4o-FG* achieves higher average correctness results than *GPT-4o* across all benchmark sets. Additional evidence that FG is responsible for this performance boost can be seen in the 'FG Success' results. For example, the 'Sorting' benchmark set sees the largest increase in average correctness (from 35% for *GPT-4o* to 67.50% for *GPT-4o-FG*) and this coincides with the proportionally largest average value for 'FG Success'. Another point in support of the efficacy of FG can be seen through the consistent superiority of *GPT-4o-FG* when compared to *GPT-4o*; this is portrayed clearly in plots (a) and (c) of Figs. 10, 11, 12, 13.

4.4.3 O4-mini without FG (*O4-mini*)

Even without the use of FG, *O4-mini* performs exceptionally well in the task of generating correct WPs – in fact, the lowest average correctness result for *O4-mini* is 90.00% (on the 'Search' benchmark set). When considering experimental configurations that do not use FG, the superiority of *O4-mini* in this problem setting is reinforced by observing that the *lowest* average correctness score for *O4-mini* (90.00% on the 'Search' benchmark set) is still higher than the *highest* average correctness score achieved by *GPT-4o* (72.78% on the 'Universal' benchmark set). Although *O4-mini* is dominant over *GPT-4o*, *GPT-4o-FG* can appear to give *O4-mini* a 'run for its money' – this is evidenced by comparable average correctness scores for each benchmark set apart from 'Sorting'. This comparable performance between *GPT-4o-FG* and *O4-mini* is further highlighted in each relevant plot of Figs. 10, 11, 12, 13.

4.4.4 O4-mini with FG (*O4-mini-FG*)

When O4-mini is enabled with FG we can see that the correct WP is attained for every program, in every benchmark set, and for every iteration. Similar to what was observed in *GPT-4o-FG*'s summary (Sect. 4.4.1), the greatest increase in average correctness (over the corresponding base-line configuration *O4-mini*) also coincides with the largest proportional value of 'FG Success'.

4.5 Notable cases

4.5.1 GPT-4o can't swap

This case concerns three programs from the 'Existential' benchmark set and the *GPT-4o* configuration. These three programs each perform a value swap between two variables in their respective loop computations (see the supplementary material link for example-two). In each instance, *GPT-4o* produced a candidate WP which failed to account for this swapping of values – this was true for every evaluation carried out; in the end, *GPT-4o* never generated a correct WP for these programs.

Another interesting point regarding the aforementioned programs is that FG generally did *not* help. The WPs that were required by these programs postconditions were complex enough that validity-fuzzing (Sect. 2.2.1) could not produce any corresponding inputs for testing. This problem arises due to the random-input generators of the underlying fuzz testing tool JQF [35]. This phenomena resulted in validity-fuzzing phases 'passing' with invalid candidate WPs. Here, the combined inadequacies of GPT-4o and FG led to a perfect storm for poor performance.

4.5.2 O4-mini's trouble with integer arithmetic

This example regards a program in the 'Universal' benchmark set (see the supplementary material link for example-three). Here, O4-mini produced a candidate WP that is correct for mathematical arithmetic, but that is actually invalid when considering integer arithmetic in Java. The initial WP generated was susceptible to integer overflow and this flaw was picked up by the validity-fuzzing phase of FG. This feedback was then given to O4-mini and the LLM could then produce the correct WP; this behaviour was observed in each iteration carried out.

5 Discussion

5.1 Threats to validity

The main threats to validity arise from our use of LLMs. The outputs of LLMs are inherently stochastic in nature [37], and so the results that we have derived may not be *exactly* reproducible. We attempt to mitigate the severity of this threat by providing the prompts and logs of our interactions with the OpenAI API.

Another threat arises when we consider the range of LLMs employed. This work solely features models from a single vendor (OpenAI) and therefore the results that we have produced are biased in this respect; the argument could be made that other models may perform significantly better or worse, and that our results do not represent this potential variety.

Additionally, data leakage [38] is typically a concern for many studies which rely on LLMs. Data leakage occurs when the information from a test set (e.g. benchmark programs) is already present in the training data of an LLM. We believe we have significantly reduced the risk of this threat by creating our benchmark programs from scratch, and hosting them in private repositories.

As is common with many empirical studies in software engineering, our work is inherently limited by the scope of the experimental benchmarks used. We attempted to alleviate this concern by ensuring that our benchmark programs covered a range of common programming operations and tasks, e.g. arithmetic, sorting, search, looping computations, etc. However, we have only explored relatively small programs operating over integer arrays. A more extensive experimentation with real code is necessary to support the practical benefits of our technique, which we leave to future work.

6 Related work

The combination of LLMs with formal methods and additional software analysis techniques is leading to a burgeoning sub-field of computer science.

Recent works have employed LLMs with formal tools to generate program properties that are adjacent to weakest preconditions (WPs); much of this work has revolved around invariant generation – a key component for software verification. The authors of [32] use LLMs to generate loop invariants that are then validated by the Vampire theorem-prover [39]; these validated invariants are then used to replace loops within the context of bounded model checking [40]. The work of [41] introduces the Lemur framework where LLMs are used to generate and repair candidate invariants that are then used within the ESBMC [42] and UAtomizer [43] frameworks. The authors of [44] fine-tune LLMs in an attempt to infer invariants for Java programs (such that these invariants are likely to be Daikon [45] generated). In a task adjacent to invariant generation, the authors of [46] use LLMs to generate postconditions from natural language documentation. Additionally, [47] does not use LLMs, but instead opts to use deep-reinforcement-learning to synthesize program invariants.

The task of generating preconditions has also been approached on several fronts [9–23]. The authors of [23] and [21] use logical abductive inference to infer the provably weakest preconditions for a class of deterministic and non-deterministic array programs. The works of [18] and [19] employ a more dynamic approach, utilising data-driven execution loops in order to learn likely preconditions. Our work differs from the above in that we combine fuzzing and LLMs in a unique manner to derive WPs; additionally, our approach can handle problem variants (including sorting and search), that the aforementioned techniques cannot (see the supplementary material link for example-four).

7 Future work

There are several avenues for future work which arise from this current study, a number of which stem from the limitations of our current approach towards generating weakest preconditions (WPs). For example, the WPs that are derived through LLMs and Fuzzing Guidance (FG) are only ever *likely*-valid and *likely*-weakest (see Sect. 2.2); this is due to the inherently incomplete nature of fuzz testing (and thus FG). One direction for future work would seek to concretely verify the weakness and validity of a candidate WP that had been generated through an LLM with FG; this could be done through the use of a formal reasoning tool like Vampire [39].

Another path for future work would seek to expand the capabilities of the fuzz testing tool which undergirds FG – namely, JQF [35]. This work might include creating a more adept and flexible orchestrator for FG, such that FG could be applied to a greater range of programs beyond those studied here.

Bug-finding is another application of the approach outlined in this work. Anecdotally, we found that WPs could be generated for real-world programs which were similar to the benchmarks here; these WPs were then able to avoid known exception-triggering test cases and reveal other fault-inducing test cases that were not already known. We also found that buggy programs could be revealed in a roundabout manner when the WP generated by an LLM with FG does not fit with a programmer's expected WP.

8 Conclusion

This work investigated the use of LLMs (specifically GPT-4o and O4-mini) for generating weakest preconditions (WPs) and introduced our novel mechanism of Fuzzing Guidance (FG) for aiding LLMs towards this goal. The effectiveness of our approach was demonstrated on a comprehensive benchmark set of deterministic array programs in Java.

In concrete terms, we found that O4-mini was generally superior to GPT-4o in generating correct WPs. Additionally, we observed that FG enabled O4-mini to consistently generate the correct WPs for all of the benchmark programs studied. Our results also showed that a less performant model such as GPT-4o, when enabled with FG, could achieve competitive performance relative to the baseline O4-mini model (i.e. O4-mini without FG).

Overall, our results indicate that LLMs are capable of producing viable candidate WPs, and that this ability is enhanced through FG.

Supplementary material Log files detailing GPT interactions and fuzzing invocations: https://doi.org/10.5281/zenodo.17018494.

Acknowledgements This work was conducted with the financial support of the Research Ireland Centre for Research Training in Digitally-Enhanced Reality (d-real) under Grant No. 18/CRT/6224, Taighde Éireann – Research Ireland under Grant number 13/RC/2094/_2, and the ERC Consolidator Grant ARTIST 101002685. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

Funding information Open Access funding provided by the IReL Consortium.

References

1. Floyd, R.W.: Assigning meanings to programs. Math. Asp. Comput. Sci. **19**, 19 (1967)
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1969)
3. Meyer, B.: Applying 'design by contract'. Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '77, pp. 238–252. Association for Computing Machinery, New York (1977). https://doi.org/10.1145/512950.512973. https://doi-org.elib.tcd.ie/10.1145/512950.512973
5. Back, R.-J.J., Akademi, A., Wright, J.V., Schneider, F.B., Gries, D.: Refinement Calculus: A Systematic Introduction, 1st edn. Springer, Berlin (1998)
6. King, J.C.: A program verifier. PhD thesis, USA (1970). AAI7018026
7. Rosenblum, D.S.: A practical approach to programming with assertions. IEEE Trans. Softw. Eng. **21**(1), 19–31 (1995). https://doi.org/10.1109/32.341844
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
9. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. PLDI '93, pp. 46–55. Association for Computing Machinery, New York (1993). https://doi.org/10.1145/155090.155095. https://doi-org.elib.tcd.ie/10.1145/155090.155095
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Footprint analysis: a shape analysis that discovers preconditions. In: Nielson, H.R., Filé, G. (eds.) Static Analysis, pp. 402–418. Springer, Berlin (2007)
11. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) Compiler Construction, pp. 159–179. Springer, Berlin (2002)
12. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: De Nicola, R. (ed.) Programming Languages and Systems, pp. 253–267. Springer, Berlin (2007)
13. Moy, Y.: Sufficient preconditions for modular assertion checking. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 188–202. Springer, Berlin (2008)
14. Astorga, A., Madhusudan, P., Saha, S., Wang, S., Xie, T.: Learning stateful preconditions modulo a test generator. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 775–787 (2019)
15. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 150–168. Springer, Berlin (2011)
16. Gehr, T., Dimitrov, D., Vechev, M.: Learning commutativity specifications. In: Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I, vol. 27, pp. 307–323. Springer, Berlin (2015)
17. Moy, Y.: Sufficient preconditions for modular assertion checking. In: Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI'08, pp. 188–202. Springer, Berlin (2008)
18. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features. ACM SIGPLAN Not. **51**(6), 42–56 (2016)

19. Sankaranarayanan, S., Chaudhuri, S., Ivančić, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 295–306 (2008)
20. Seghir, M.N., Kroening, D.: Counterexample-guided precondition inference. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems, pp. 451–471. Springer, Berlin (2013)
21. Sumanth Prabhu, S., Fedyukovich, G., D'Souza, D.: Maximal quantified precondition synthesis for linear array loops. In: European Symposium on Programming, pp. 245–274. Springer, Berlin (2024)
22. Zhou, Z., Dickerson, R., Delaware, B., Jagannathan, S.: Data-driven abductive inference of library specifications. Proc. ACM Program. Lang. **5**(OOPSLA) (2021). https://doi.org/10.1145/3485493
23. Sumanth Prabhu, S., D'Souza, D., Chakraborty, S., Venkatesh, R., Fedyukovich, G.: Weakest precondition inference for non-deterministic linear array programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 175–195. Springer, Berlin (2024)
24. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 128–148. Springer, Berlin (2013)
25. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: a survey. IEEE Trans. Softw. Eng. **47**(11), 2312–2331 (2019)
26. Jordan, M.I., Mitchell, T.M.: Machine learning: trends, perspectives, and prospects. Science **349**(6245), 255–260 (2015)
27. Zhu, X., Wen, S., Camtepe, S., Xiang, Y.: Fuzzing: a survey for roadmap. ACM Comput. Surv. **54**(11s), 1–36 (2022)
28. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: whitebox fuzz testing in production. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 122–131. IEEE (2013)
29. Naveed, H., Khan, A.U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., Mian, A.: A comprehensive overview of large language models. arXiv preprint (2023). arXiv:2307.06435
30. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., Chen, W.: Codet: Code generation with generated tests. arXiv preprint (2022). arXiv:2207.10397
31. Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of code language models on automated program repair. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1430–1442. IEEE (2023)
32. Pirzada, M.A., Reger, G., Bhayat, A., Cordeiro, L.C.: Llm-generated invariants for bounded model checking without loop unrolling. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 1395–1407 (2024)
33. OpenAI: Hello GPT-4o (2025). https://openai.com/index/hello-gpt-4o/. Accessed 2025-27-05
34. OpenAI: introducing-o3-and-o4-mini (2025). https://openai.com/index/introducing-o3-and-o4-mini/. Accessed 2025-27-05
35. Padhye, R., Lemieux, C., Sen, K.: Jqf: coverage-guided property-based testing in Java. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 398–401 (2019)
36. Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C., Li, Y., Gupta, A., Han, H., Schulhoff, S., et al.: The prompt report: a systematic survey of prompting techniques. arXiv preprint (2024). arXiv:2406.06608
37. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30** (2017)
38. Inan, H.A., Ramadan, O., Wutschitz, L., Jones, D., Rühle, V., Withers, J., Sim, R.: Training data leakage analysis in language models. arXiv preprint (2021). arXiv:2101.05405
39. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: International Conference on Computer Aided Verification, pp. 1–35. Springer, Berlin (2013)
40. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
41. Wu, H., Barrett, C., Narodytska, N.: Lemur: Integrating large language models in automated program verification. arXiv preprint (2023). arXiv:2310.04870
42. Menezes, R.S., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., et al.: Esbmc v7.4: harnessing the power of intervals (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 376–380. Springer, Berlin (2024)
43. Heizmann, M., Dietsch, D., Greitschus, M., Leike, J., Musa, B., Schätzle, C., Podelski, A.: Ultimate automizer with two-track proofs: (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 950–953. Springer, Berlin (2016)
44. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: International Conference on Machine Learning, pp. 27496–27520. PMLR (2023)
45. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
46. Endres, M., Fakhoury, S., Chakraborty, S., Lahiri, S.K.: Can large language models transform natural language intent into formal method postconditions? In: Proceedings of the ACM on Software Engineering **1** (FSE), pp. 1889–1912 (2024)
47. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: a deep learning framework for program verification. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II, vol. 32, pp. 151–164. Springer, Berlin (2020)