Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM

GABRIEL RYAN*, Columbia University, USA
SIDDHARTHA JAIN†, MINGYUE SHANG, SHIQI WANG, XIAOFEI MA, MURALI KRISHNA RAMANATHAN, and BAISHAKHI RAY, AWS AI Labs, USA

Testing plays a pivotal role in ensuring software quality, yet conventional Search Based Software Testing (SBST) methods often struggle with complex software units, achieving suboptimal test coverage. Recent work using large language models (LLMs) for test generation have focused on improving generation quality through optimizing the test generation context and correcting errors in model outputs, but use fixed prompting strategies that prompt the model to generate tests without additional guidance. As a result LLM-generated testsuites still suffer from low coverage.

In this paper, we present SymPrompt, a code-aware prompting strategy for LLMs in test generation. SymPrompt's approach is based on recent work that demonstrates LLMs can solve more complex logical problems when prompted to reason about the problem in a multi-step fashion. We apply this methodology to test generation by deconstructing the testsuite generation process into a multi-stage sequence, each of which is driven by a specific prompt aligned with the execution paths of the method under test, and exposing relevant type and dependency focal context to the model. Our approach enables pretrained LLMs to generate more complete test cases without any additional training. We implement SymPrompt using the TreeSitter parsing framework and evaluate on a benchmark challenging methods from open source Python projects. SymPrompt enhances correct test generations by a factor of 5 and bolsters relative coverage by 26% for CodeGen2. Notably, when applied to GPT-4, SymPrompt improves coverage by over 2× compared to baseline prompting strategies.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies** → Artificial intelligence;

Additional Key Words and Phrases: Test Generation, Large Language Models

ACM Reference Format:
Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (July 2024), 21 pages. https://doi.org/10.1145/3643769

1 INTRODUCTION

Testing is an essential component of software development that allows developers to catch bugs early in the development lifecycle and prevents costly releases of buggy software [31]. However, manual test writing can be time-consuming, taking up more than 15% of development time on average [10]. Extensive research has therefore been devoted to developing automated test generation approaches, which can automate the process of writing testsuites for software units under development. Automated test generation for the purposes of generating a testsuite that becomes

*Work done while the author was an intern at AWS AI Labs
†Corresponding author

Authors' addresses: Gabriel Ryan, gabe@cs.columbia.edu, Columbia University, USA; Siddhartha Jain, siddjin@amazon.com; Mingyue Shang, myshang@amazon.com; Shiqi Wang, wshiqi@amazon.com; Xiaofei Ma, xiaofeim@amazon.com; Murali Krishna Ramanathan, mkraman@amazon.com; Baishakhi Ray, rabaisha@amazon.com, AWS AI Labs, USA.

ACM 2994-970X/2024/7-ART43
https://doi.org/10.1145/3643769

part of development codebase is typically performed in a *regression setting*, which assumes that the code currently under test is implemented correctly, and the objective is to generate a suite of tests that will effectively detect *future* bugs that may be introduced in to the codebase during later development, causing a regression [42].

Limitation of Existing Approaches. Widely used test-suite generation tools such as Evosuite [13] use a Search Based Software Testing (SBST) approach in which test inputs are randomly generated and mutated to maximize coverage of the software unit under test. However, SBST approaches struggle to generate high coverage test inputs in many cases, such as when branch conditions depend on specific values or states that are difficult to resolve with randomized inputs and heuristics. In a large scale study of SBST on 110 widely used open source projects, Fraser et al. observed that more than 25% of the tested software classes had less than 20% coverage [15].

The limitations of SBST have motivated recent work using large language models (LLMs) for automated testsuite generation [36]. In this setting, LLMs are typically instructed to generate test cases by supplying the source code of the focal method and optionally some additional code context. The instruction with which as user interacts with an LLM is commonly referred to as *prompt*.

Unlike SBST approaches that reason about the execution behavior of a method under test (commonly called *focal method*), LLMs approximate the overall functionalities of the focal method based on the natural naming convention [2, 4] of its implementation (e.g., meaningful method name, variable names, etc.), API usage, and calling context. This capability of LLMs can be harnessed to create test cases, specifically generating inputs targeting branch conditions that necessitate particular input values or states. However, when tasked with generating test inputs for methods with challenging or complex branch conditions, LLMs usually succeed in producing inputs for the easy branches, leaving behind the branches with more complex conditions. Consequently, for hard real-world use cases, LLMs usually struggle to generate high coverage testsuites [3, 12].

Figure 1 illustrates how a focal method can pose challenges for both SBST and LLM-based test generation approaches. The method takes a string representing a filesystem object as input and categorizes the type of filesystem object based on an external method call. Generating high coverage test inputs for this method is very difficult for a SBST approach because generating strings that represent different filesystem devices is extremely unlikely with randomized input generation, and in practice it will only test paths for which it has preprogrammed heuristics to generate string inputs such as directories (see Figure 1b). Conversely, an LLM trained on a large code corpus should in theory have the knowledge to generate strings representing filesystem objects like block devices and sockets, but in practice, when given an open ended prompt to implement a set of tests for the method, will only generate tests for relatively simple and common inputs for a filesystem utility function, such as temporary files and directories (see Figure 1c).

Our Solution. In this work we introduce a novel approach, SymPrompt, to constructing prompts leveraging different code properties, which enables LLMs to generate test inputs for more complex focal methods. Recent research employing LLMs for logical problem-solving demonstrates that when LLMs are prompted to decompose the problem into multiple stages of reasoning first instead of directly trying to generate the answer, they exhibit greatly enhanced capability in solving more complex problems [37, 38, 41]. We build on this concept devising a unit test specific decomposition framework and integrating it into a novel multi-stage prompting strategy. Our key insight is that the process of generating a test-suite to fully cover a focal method can be broken down into a sequence of logical problems that are posed to the LLM prompts. For each prompt, the LLM will generate an appropriate test case, and thus, will eventually generate a series of test cases with higher code coverage to test the diverse behavior of the focal method.

At a high level, SymPrompt works in three stages: (i) Given a focal method, SymPrompt tries to statically capture the execution behavior of the method by collecting approximate path constraints and return values for each execution path. (ii) SymPrompt collects some properties of the focal method including argument types, external library dependencies, code context, etc. (iii) For each execution path, SymPrompt constructs a prompt using the collected information and solicits the LLM to generate a test input that will execute the corresponding path. This generation is carried out iteratively, and the test cases generated in previous iterations are appended to subsequent prompts to provide further guidance to the LLM in generating a thorough testsuite.

Conceptually, our approach resembles symbolic analysis based test generation techniques. However, traditional symbolic analysis is known to suffer in real-world code with complex data types, external dependencies, and complex branching behavior. The proposed work addresses these limitations in three ways. First, SymPrompt collects approximate constraints based on static code rather than attempting to resolve all data types and unresolved dependencies while collecting path constraints. For instance, instead of attempting to symbolically reason about a call like `os.is_dir()`, we keep it as it is and rely on the LLM to reason about the underlying functionality of the method call based on its name and usage. Second, to mitigate computational overhead arising from numerous branching conditions, we focus only on the paths with unique branching conditions. Finally, instead of relying on a solver to resolve underlying constraints, we employ an LLM to generate test cases based on the prompt. The insight here is that, although the LLM may not precisely reason about all possible constraints, it typically extracts sufficient information from the code context and approximate path constraints to generate meaningful test inputs.

Our step-by-step prompting approach further facilitates the test generation process. Without this type of prompting, the LLM would need to (1) infer what branch the previous test it generated covered, (2) search for a new branch to address, and (3) create a test for it. In our approach, we delegate steps 1 and 2 to traditional static analysis, allowing the LLM to focus solely on step 3.

Results. We implement SymPrompt for Python using the TreeSitter parsing framework and evaluate on a benchmark of 897 focal methods that are challenging for existing SBST test generation. We prototype our technique on open source CodeGen2 16B LLM: SymPrompt improves the ratio of correct test generations by a factor of 5 and improve relative coverage by 26%. To check generalizability of our technique, we further evaluate SymPrompt with a state-of-the-art LLM, GPT-4: SymPrompt improves relative coverage by 105% over tests generated with a baseline prompting strategy.

In summary, this paper makes the following contributions:

(1) We introduce *Path Constraint Prompting*, a novel, code-aware prompting strategy for LLM test generation that breaks the process of generating a testsuite into a multi-stage procedure of generating test inputs for each execution path in the method under test.
(2) We implement our approach in SymPrompt using the TreeSitter parsing framework with integrations for both open-source transformers and GPT models. SymPrompt is currently a proprietary research prototype and we are working with our legal team to make the code publicly available.
(3) We evaluate SymPrompt on a benchmark of 897 methods that are challenging for existing SBST approaches from widely used Python projects and show it improves relative coverage by 26% for CodeGen2 and by a factor of more than 2× (105% relative improvement) when used with GPT-4.

```python
def exists_as(path: _PATH) -> str:
    path = normalize_path(path)

    if path.is_dir():
        return 'directory'
    if path.is_file():
        return 'file'
    if path.is_block_device():
        return 'block device'
    if path.is_char_device():
        return 'char device'
    if path.is_fifo():
        return 'FIFO'
    if path.is_socket():
        return 'socket'
    return ''
```

(a) Focal method.

```python
# Automatically generated by Pynguin.
import flutils.pathutils as module_0

def test_case_0():
    str_0 = '#O01E'
    str_1 = module_0.exists_as(str_0)

def test_case_1():
    str_0 = '/pynguin'
    str_1 = module_0.exists_as(str_0)
```

(b) SBST test generations.

```python
# test case for function exists_as
def test_exists_as():
    assert exists_as('.') == 'directory'
    assert exists_as(Path('./tmp')) == 'file'
    assert exists_as('./tmp/foobar') == 'file'
```

(c) LLM test generations.

```python
def test_case_1():
    """
    Testcase 1 for exists_as(path: _PATH) -> str:
    test case where path.is_dir()
    returns: 'directory'
    """
    assert exists_as('/tmp') == 'directory'

def test_case_2():
    """
    Testcase 2 for exists_as(path: _PATH) -> str:
    test case where not (path.is_dir()),
                    and path.is_file()
    returns: 'file'
    """
    assert exists_as('/etc/passwd') == 'file'

def test_case_3():
    """
    Testcase 3 for exists_as(path: _PATH) -> str:
    test case where not (path.is_dir()),
                    not (path.is_file()),
                    and path.is_block_device()
    returns: 'block device'
    """
    assert exists_as('/dev/ttyS0') == 'block device'
```

(d) SymPrompt test generations.

Fig. 1. Example test generations from an SBST tool (Pynguin), zero shot LLM (CodeGen2), and SymPrompt prompts for focal method `exists_as` in the `flutils` open source Python project. An SBST approach is unable to generate full coverage tests for this method without special configuration because it is unable to generate strings that represent specific types of filesystem objects (e.g., block devices). An LLM conversely is able to generate input strings associated with filesystem objects such as block devices, but in practice will only test a small subset of use cases based on the most likely usage scenarios such as paths to files and directories. SymPrompt constructs path specific prompts to guide the model to generate high coverage testsuites.

2 WORKING EXAMPLE

In this section we provide a working example of how path constraint prompts are constructed and used to guide an LLM generate high coverage tests in a regression setting. Figure 1a illustrates a focal method, named `exists_as`, extracted from our evaluation within the `flutils.path_utils` module. Testing this method poses a significant challenge for both SBST (Search-Based Software Testing) and LLM-based approaches due to its extensive branching structure, which requires tailored input data to achieve full coverage of test cases. Each branch in this method demands specific input values that satisfy precise constraints—particular string inputs that reference filesystem objects like directories and block devices.

SBST-based Test Generation. Figure 1b shows a suite of regression tests generated with the Python SBST tool `Pynguin`. Since SBST approaches generate inputs randomly according pre-programmed heuristics, they are unlikely to generate inputs that represent specific filesystem objects such as block devices and sockets unless the tool was specifically configured to generate strings representing these objects as inputs. In the case of Pynguin, the input strings it generated in Figure 1b represent a nonexistent device (`'#O01E'`) and a directory that is defined in the Pynguin test environment (`'/pynguin'`) that cover two of return behaviors in the focal method, but do not

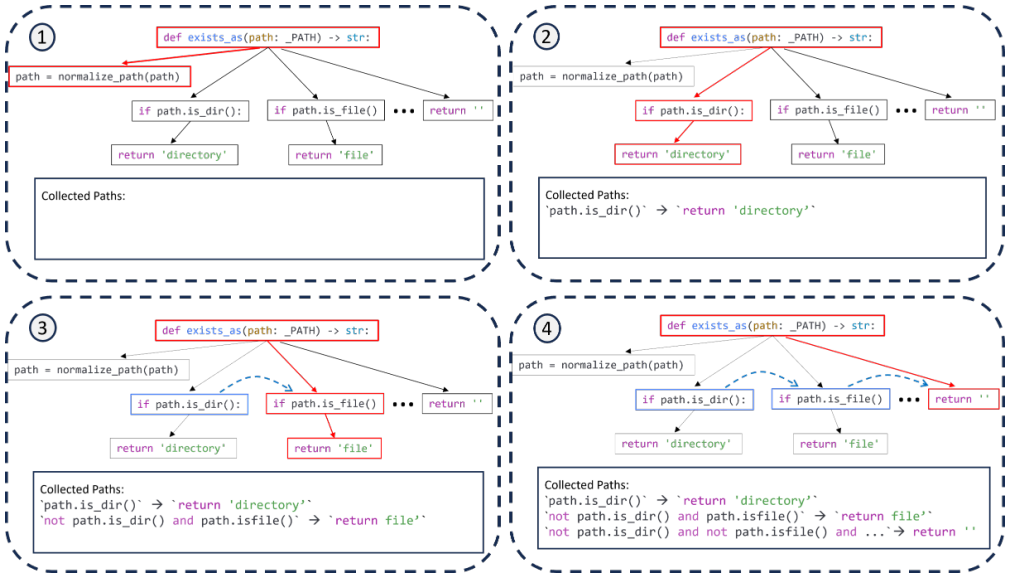

Fig. 2. Workflow for generating path constraint prompts. The focal method shown in Figure 1a is first parsed and its abstract syntax tree is traversed in preorder. In step ①, the traversal first visits the first method statement, `normalize_path(path)`, but does not record any information since it is not a branch constraint. In step ②, it then traverses to the first `if` statement, and records that there is a constraint `path.is_dir()` that must be satisfied to execute the current path on the AST. It then reaches the `return 'directory'` under the first `if` check, and records that there is an execution path where 'directory' is returned when `path.is_dir()` is true. The preorder traversal next visits the `if path.is_file()`, `return 'file'` branch of the AST in ③ and records a second path where `path.is_dir()` is `false` and `path.is_file()` is `true`, and the return behavior is `'file'`. This traveral continues until in step ④, the final `return` statement is reached, based on an execution path where none of the branch constraints are `true`. Each collected execution path and return value is then used to construct prompts for test generations that specifies both the path constraints and return behavior for the target test case.

test the other use cases that require other specific input values. Pynguin is not capable of generating input strings that test these other use cases without special configuration. In addition, it generates tests that do not follow common usage patterns in developer written tests, which makes the tests more difficult to maintain [36].

LLM-based Test Generation. Figure 1c shows a set of regression test calls generated using a standard code-completion testing prompt used by Lemieux et al. [22]. Unlike an SBST tool such as Pynguin, an LLM has an approximate domain understanding to reason that a branch constraint like `path.is_file()` will likely be tested by an input like `'./tmp'` and can generate associated input strings to test these use cases, even without observing the definitions of `normalize_path` and `is_file` on the first and second lines of code in `exists_as` in Figure 1a. However, in cases where focal methods have many different paths, and some paths represent less common use cases (e.g., if the input string is a block device), LLMs will only generate test cases for the most common use cases of a focal method, even if many test generations are sampled. In this case, the LLM (CodeGen2) only generates tests for the two most common uses, where the input is either a directory or a file.

SymPrompt. Our approach works in three steps: (i) *Collecting Approximate Path Constraints.* This step is the core to our approach. Figure 2 shows how path constraints are collected and used to

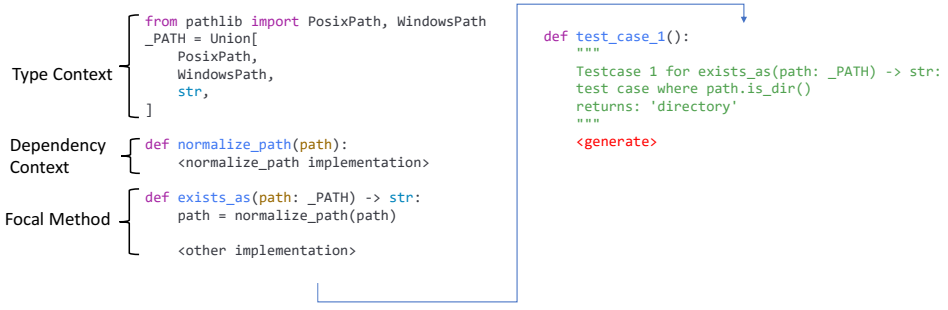

Fig. 3. Example of generation used by SymPrompt. The prompt exposes both the type and dependency context of the focal method to the model in addition to the path constraint prompt.

prompt an LLM to generate high coverage tests. Possible execution paths in the focal method are collected by traversing the method's abstract syntax tree (AST) in preorder and recording branch conditions on each possible execution path, where if a branch is not taken its branch condition is inverted on a given path. When a return statement is reached, the set of branch constraints that need to be satisfied to reach that return statement is recorded, along with the returned value. Each recorded path constraint and return value is then used to generate a prompt, which instructs the LLM to generate a test case targeting the corresponding path.

(ii) *Context Construction.* Besides the path constraint, each prompt includes the focal method signature, which guides the LLM to generate a correct focal method call. We further include additional focal context for generation based on the types and methods that appear in the focal method that exposes relevant data structure and external library dependencies the model may need to reference for effective test generation, as shown in Figure 3. For the focal method `exists_as`, we include the definition of the input parameter `_PATH` and the external method call `normalize_path`. One such prompt for our working example is shown in Figure 3.

(iii) *Test Generation.* For the focal method `exists_as`, path constraint prompting guides the LLM to generate tests for use cases it does not normally cover without specific prompting. In particular, path constraint prompts guide the model to generate test strings that satisfy the constraints for `is_block_device()`, `is_char_device()`, `is_fifo()`, and `is_socket()`. When specifically prompted, CodeGen2 is able to generate correct test inputs in three of these four cases in our evaluation, more than doubling the number of tested branches covered in the generated testsuite (see Figure 1d).

Path constraint prompting, in conjunction with type-aware focal contexts, allows us to leverage advantages of LLMs in deriving meaningful test inputs from context, understanding how to correctly initialize complex input types, utilizing relevant external API calls in testing, and resolving difficult-to-cover branch constraints, while deriving the advantages of a coverage-aware testing strategy similar to SBST by prompting for a set of high coverage tests.

3 METHODOLOGY

In this section, we elaborate on our approach, SymPrompt, to generating test cases for a given focal method in a regression setting. At the core of our approach is crafting tailored prompts that break the problem of test generation into multiple stages of reasoning based on the possible execution paths in focal method. These prompts are designed to instruct the model to generate test cases for

a specific set of execution paths within the focal method that will ensure comprehensive branch and line coverage.

3.1 Overview

Our approach to constructing multi-stage reasoning prompts is based on static symbolic analysis techniques [6, 17, 20] to reason about underlying program behavior. At a high level, for each control path (ρ) of a focal method, we statically collect a path constraint Φ_ρ that will steer the program execution along ρ.

Once the path constraints are collected, in the conventional symbolic analysis-based testing approach, a systematic search algorithm is employed to enumerate all the path constraints [18]. Feasible paths are those for which the corresponding Φ_ρ is satisfiable—any concrete solution that meets the conditions of Φ_ρ can be used to execute and test the path ρ in a regression setting.

However, when dealing with complex real-world code (for instance, when input parameters of the focal method are of complex data types, or when focal methods rely on external dependencies and API calls), the traditional symbolic analysis-based techniques often encounters difficulty in finding a solution. Furthermore, in cases where focal methods feature numerous nested branches and conditions, symbolic execution frequently experiences significant computational overhead.

In this work, we overcome the above mentioned shortcomings in three ways. First, instead of trying to resolve all the data types and unresolved dependencies while collecting path constraints, we allow approximations. For example, if the method `path.is_dir()` in Figure 2 cannot be reasoned about symbolically, we leave the condition as it is while collecting ϕ_ρ. Second, to address computational overhead with many branching conditions, we only focus on the paths having unique branching conditions. This design decision significantly helps to reduce the number of paths. Finally, instead of using a solver to solve the underlying constraints we leverage an LLM to generate test cases based on the collected path constraints. Our insight is that, although the LLM may not reason precisely about all possible constraints, it usually derives enough information from code context and the approximate path constraints to generate meaningful test inputs.

To this end, our test generation contains three steps: (Step-I) collecting approximate path constraints and return expressions for program paths, (Step-II) capturing relevant code context, and (Step-III) generating prompts amenable to a LLM by leveraging the path constrains and context identified in the above step. This entire process is done statically without executing the focal method. Figure 4 gives a high level overview of how we use SymPrompt to generate tests.

3.2 Step-I: Collecting Approximate Path Constraints

The objective of SymPrompt is to generate test inputs that will follow specific execution paths in the focal method. To facilitate this, we construct prompts to expose the model to relevant path constraints. This step describes in details how we collect approximate path constraints by statically traversing the abstract parse tree (AST) of the focal method.

3.2.1 Path Constraint Collection. Our approach to collecting path constraints is similar to static symbolic execution. We perform a preorder traversal of the focal method abstract parse tree and collect constraints for each branch and loop condition while maintaining a set of all constraints that appear on each possible execution path. When the traversal encounters a `return` statement, we record the path constraints for the paths that terminate at that `return`, along with the return value expression. To prevent an exponential explosion of paths when collecting path constraints, we minimize the path set on each traversal step to only include paths that increase overall branch coverage (see Section 3.2.2).

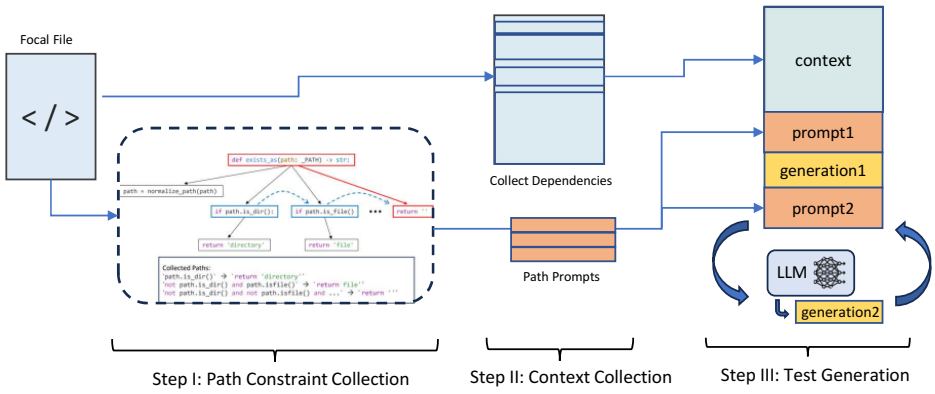

Fig. 4. Overview of SymPrompt's framework for test generation. In Step-I, path constraint collection is performed on the focal method. In Step-II, the type and dependency context from the focal method are parsed from the focal file along with the focal method itself. Finally, in Step-III, prompts for each set of path constraints are then constructed and iteratively passed to the model to generate test cases.

```
1: procedure MINIMIZEPATHS(paths)
2:     minconstraints = {}
3:     minpaths = {}
4:     for path in paths do
5:         path_constraints = splitConstraints(path)
6:         if any of path_constraints is not in minconstraints then
7:             minconstraints = minconstraints ∩ path_constraints
8:             minpaths = minpaths ∩ path
9:     return minpaths
```

(a) Path Minimization Algorithm.

```
def method(args):

    if cond1:
        <blk1>
    else:
        <blk2>

    if cond2:
        <blk3>
    else:
        <blk4>

    if cond3:
        <blk5>
    else:
        <blk6>

    return <return>
```

```
path1: cond1, cond2, cond3
path2: not cond1, cond2, cond3
path3: cond1, not cond2, cond3
path4: cond1, cond2, not cond3
path5: not cond1, not cond2, cond3
path6: not cond1, cond2, not cond3
path7: cond1, not cond2, not cond3
path8: not cond1, not cond2, not cond3
```

Minimized Paths

(b) Example on three branches.

Fig. 5. Path minimization algorithmic definition and illustration of how path minimization prevents the number of paths from growing exponentially in the number of branches. A method with $n = 3$ `if` conditions will have $2^n = 8$ possible execution paths, but applying the algorithm in 5a reduces the number of paths that need to be tested to at most $n + 1 = 4$, each of which covers a unique branch condition.

We provide a detailed description of the path constraint collection procedure in Appendix A in our supplemental materials.

3.2.2 Path Minimization. One challenge in enumerating execution paths in a method is that the number of possible execution paths grows exponentially in the number of branches. Therefore collecting path constraints can result in a very large number of paths to test. This is undesirable because most of the paths will usually not add additional line or branch coverage and therefore are redundant from the perspective of a developer. Therefore, when collecting path constraints to construct path constraint prompts we only collect a *linearly independent subset* of basis path constraints for use in prompting.

Basis paths were first proposed as a measure of method complexity [26] and are referred to a basis paths because they form a linear basis for all paths when expressed as a set of columns in the adjacency matrix of the method control-flow-graph. For testing purposes, basis paths are

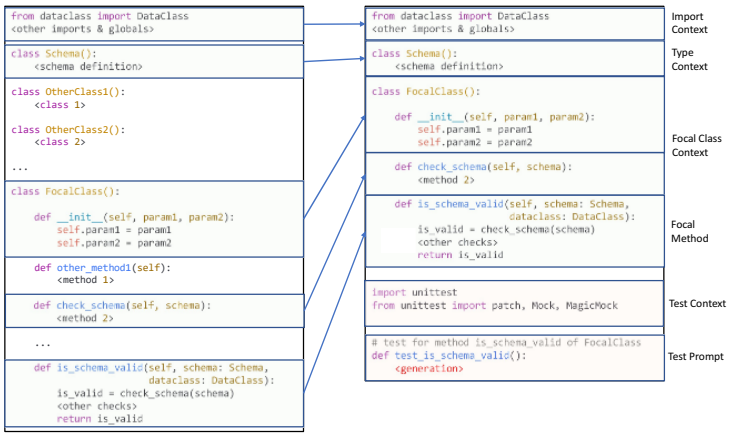

Fig. 6. Illustration of context construction on a focal method `is_schema_valid`. The context is composed of 5 components: 1. *Imports and Globals.* Imported modules and classes that appear in the focal file, along with global variables defined in focal file. 2. *Type Context.* Definitions of types that are used in the focal method and defined in the focal file. 3. *Focal Class Type Context.* The definition of the focal class type signature and initialization. 4. *Focal Class Method Context.* Definitions of any focal class methods that are called in the focal method. 5. *Focal Method.* The definition of the focal method to be tested. Constructing the generation context to expose relevant types and dependencies helps the model to attend to relevant definitions when generating. All objects that are included in the context are included in the test execution context and override any import statements generated by the model. This is particularly beneficial for test generations with GPT models, which we found are prone to generating hallucinated import statements (see Section 4.4).

convenient because a set of tests that execute a set of basis paths in a method will achieve full branch coverage on that method.

Algorithm 5a describes how we compute basis paths from a set of path constraints. We first split each path into its constituent branch constraints, and check if any of the path's constraints are not in the set of linearized path constraints. If a path includes a branch constraint that is not in the linearized constraint set, we add it to the set of linearized paths and add its branch constraints to the set of linearized path constraints.

Figure 5 illustrates how path minimization reduces the number of collected paths for a simple example method with three sequential `if-else` branches. The total number of paths in the method is $2 \times 2 \times 2 = 8$, but applying Algorithm 5a reduces the number of paths to at most 4 (one initial path, +1 for each branch). We apply path minimization after collecting new path constraints from any `if` or `while` statement while traversing the focal method AST.

3.3 Step-II. Context Construction

Prior work in test generation with LLMs have demonstrated that including additional context to the focal method definition is beneficial to the quality of test generations [36]. For SymPrompt, we focus on exposing two types of context to the model that are beneficial to generating correct test cases: (i) *Type context.* Context that includes type definitions for focal method parameters guides the model to generate correct test inputs, particularly when the input involves complex data structures. (ii) *Dependency Context.* Dependency context includes imports and other definitions that are used in the focal method, such as API calls and global data structures. We construct the generation focal context selectively to include only type context and dependency context. In addition, we construct

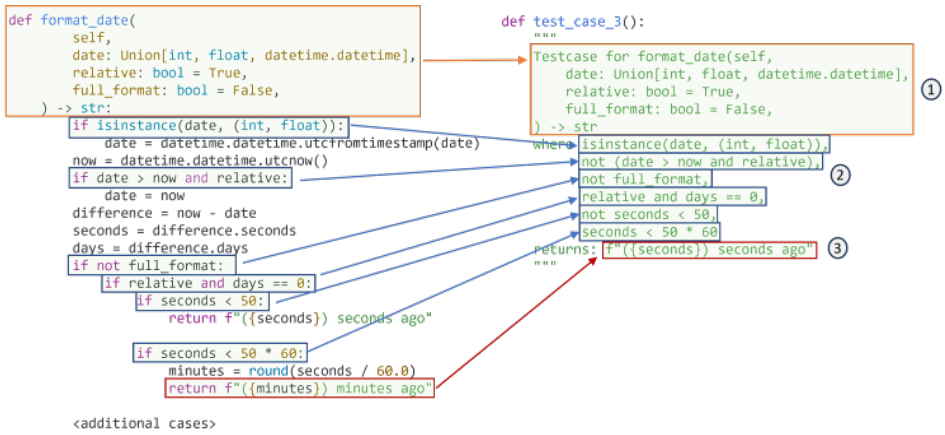

Fig. 7. Example of path constraint prompt construction on a simplified focal method in the `tornado` project. Each path prompt has three components: ① **Method Signature.** The prompt specifies the test case is for the focal method, including its full signature. ② **Path Constraints.** Next, the prompt specifies what path constraints should be satisfied by the inputs in the given testcase in order to follow the desired execution path. ③ **Return Behavior.** The prompt specifies what return behavior, if any, is expected on the specified execution path. This can guide correct generation of assertions for specific return cases.

the test execution context based on the focal context to override any model-generated imports and type definitions. We found that this prevents many common errors, particularly when generating tests with chat-tuned models such as GPT-4 (See Section 4.4).

Figure 6 illustrates how we construct the generation context. We first parse the focal file and extract all import statements, global definitions, class definitions, and method definitions. We first add all import statements and global definitions to the context. We then check which classes and functions are used in the focal method and add their definitions to the context window if they are defined in the focal file. If the focal method is defined in a class, we extract the class's type signature and constructor definition, and then include the definitions of any methods in the focal class that called in the focal method. Finally, we append the full focal method definition at the end of the context window immediately before the test context and test generation prompt.

3.4 Step-III. Test Generation

Once a set of path constraints have been collected for a given focal method, we construct prompts to generate tests for each path based on the focal method signature, path constraints, and return behavior on each collected execution path. Figure 7 shows how a prompt is constructed for a single path in a simplified focal method for serializing a timestamp value. Each prompt first specifies the focal method the testcase is for along with the focal method signature, which serves to guide the model to correctly generate the focal method call in the test. The path constraints are combined to indicate that all constraints should be applied when generating the given test case. If the path used to generate the prompt terminates in a return statement, a `returns: <return_value>` specifier is added to the prompt to guide assertion generation on the return value.

After constructing a set of path constraint prompts, we use these to iteratively prompt the model to generate tests for each path, and include the generated tests in the prompt for the next generation. This iterative prompting procedure, illustrated in Figure 4, reduces the challenging problem of generating a high coverage testsuite into a multistage reasoning procedure based on

individual execution paths and test inputs. Following common practice in code generation, for each generation, we check if the output code can be parsed, and if there are parse errors delete lines from the end of the generated code until it parses without errors. This eliminates errors caused by truncated model outputs.

Once a full set of tests have been generated, we construct the test execution context by importing all defined classes and variables that appear in the generation context. In addition, we copy all of the import statements that appear in the generation context and include them in the execution context. If the model generated any import statements for objects that are imported in the execution context, we remove the model-generated import.

4 EVALUATION

We address the following research questions in our evaluation:

- **RQ1. Performance Impact.** How does SymPrompt affect testing performance over simple test generation prompting methods in a regression setting?
- **RQ2. Training Data Memorization.** How does SymPrompt perform on projects that do not appear in the model training data?
- **RQ3. Design Choices.** How do path constraints and calling contexts each contribute to the performance gains achieved by SymPrompt?
- **RQ4. Performance Impact on Large Models.** Does SymPrompt still improve performance on very large models?

Experiment Setting. We perform all evaluations on an AWS p4d.24xl instance with 8 Nvidia A100 GPUs and 96 vCPUs. We use Python 10 and Pytorch 1.13 with the Huggingface transformers framework to run local models.

Evaluation Metrics. Based on prior work on test generation [32, 36], we use the following metrics in evaluation:

(1) **Pass@1:** The average number of tests that run without errors and pass in each generated testsuite for each focal method when executed.
(2) **FM Call@1:** The average number of tests that correctly call the focal method in each generated testsuite for each focal method.
(3) **Correct@1:** The average number of tests that both pass and correctly call the focal method for each generated testsuite.
(4) **Line & Branch Coverage:** Average line and branch coverage on the focal method for each testsuite generation.

These metrics are computed similarly to the standard Pass@1 metric used in program generation benchmarks such as HumanEval [9] and MBPP [5], but may contain partially passing rates on each generation, since a single generated testsuite may contain both passing and failing tests.

Benchmark Programs. We evaluate on 897 focal methods drawn from 26 open source projects used in benchmarks in prior work [22, 39]. We select focal methods from these projects where Pynguin [24] (an SBST tool for Python) was unable to achieve full coverage on the focal method during 10 runs, indicating the focal method poses a challenge for existing automated test generation tools. In this dataset, Pynguin had an average line coverage of 72.4% with std. deviation of 12.7%.

Evaluated Models. Following the literature on test generation with language models [3, 36], we evaluate SymPrompt with a single recent open source model, CodeGen2 [27]. To measure SymPrompt's ability to generalize to larger closed source models, we additionally perform an evaluation with GPT-4 [28].

```
<focal method & context>

def test_focalmethod():
    pass
```

(a) No-Op test.

```
<focal method & context>

# unit test for function <focalmethod>:
def test_focalmethod():
    <generate>
```

(b) Baseline test prompt.

```
<focal method & context>

Please infer the intention of <focal_method> defined above.
<generate1>

You are a professional Python developer who writes unit tests.
Write a set of tests for <focal_method>.
<generate2>
```

(c) GPT Describe-Generate prompt.

Fig. 8. Baseline prompts used in evaluation.

4.1 RQ1: Performance Improvement

We first evaluate how SymPrompt affects testing performance over simple test generation prompting methods in a regression setting.

Evaluation. We evaluate RQ1 on CodeGen2 [27], a recently released open source code model with 16 Billion parameters and a mixed causal training objective with span corruption and infilling. We use two baselines: the pynguin-generated test cases, and and a test completion prompt based on prior work using Codex for test generation [22]. For each focal method and prompting strategy, we performed 10 generations with CodeGen2 and averaged the results for each focal method over the 10 generations.

We also include as a baseline No-Op tests which simply load the target module in our execution framework and then return without explicitly running any test cases. In many cases loading the module containing the focal method will cause a significant proportion of the focal method to be covered, since both the method signature and docstring are counted as covered on load. The No-Op test format is shown in Figure 8a. The test completion prompt is formatted as a partial test function that the model fills in. First a comment specifies the method is a test of the focal method, followed by the function signature for the a test of the focal method, as shown in Figure 8b.

Observations. Table 1 summarizes our results for the evaluation of RQ1. Overall, we found that SymPrompt significantly improves performance both in generating tests that execute with/without errors and call the focal method, and also significantly improve coverage over the model tests generated with the baseline prompt. We found the SymPrompt is beneficial in two ways: first, the structure of path constraint prompts guides the model to generate tests by placing relevant information about how to call the focal method correctly based on its signature and how to generate correct assertions based on the expected method return behavior. This contributes to tests generated with SymPrompt improving pass rate by a factor of nearly 4× and correct rate, where the test both passes and calls the focal method, by a factor 5×. We found that while the model could often generate focal method calls 34% of tests on average with the baseline prompt, most of these generations had incorrect focal method usage or other errors that prevented them passing, resulting in an overall correct generation of only 3% on average.

The tests generated by SymPrompt also achieve a 10% improvement in line coverage and 4% in branch coverage over the baseline prompts, indicating that path constraint prompts are also effective at guiding the model to generate higher coverage tests that exercise more distinct use cases for the target method. Moreover, if the line coverage that occurs on module load is taken into account by measuring improvement over No-Op tests, then the baseline prompts only improve absolute coverage by 5% while SymPrompt improves coverage by 15%, a 3× improvement.

On average, Pynguin achieves 72% line coverage and 64% branch coverage this benchmark, which is still significantly higher than the coverage of the tests generated by CodeGen2 with SymPrompt. As has been observed in prior work, test generation model results are biased by errors in model generations that cause many test suites to fail before they can execute the focal method [33]. In

Table 1. **RQ 1. Results (LHS):** Results for evaluation of CodeGen2 test generation on 897 hard-to-test focal methods are shown in the left 4 columns. SymPrompt is effective at guiding the model to call the focal method correctly and generate passing tests, as well as covering a wider range of use cases in its generated tests. These result in relative improvements of 5× more correct test generations (which call the focal method and pass) and a 10% improvement in coverage over tests generated with a baseline test completion prompt. SymPrompt Filtered shows results when test generations with errors are discarded to provide a controlled comparison with Pynguin. Under this setting, SymPrompt compares favorably with Pynguin, achieving marginally better line and branch coverage. **RQ 2. Results (RHS):** Results for evaluation on focal methods unseen in training data (based on AmIInTheStack tool [1]) are shown in the right 4 columns. Compared to the full benchmark results, both the baseline prompts and SymPrompt have lower rates of correct test generations and coverage. However, tests generated with SymPrompt still have significantly better correct generation rates (4× improvement) and coverage over the baseline prompt generated tests.

	Full Benchmark					Unseen Projects				
Method	Pass@1	FM Call@1	Cor-rect@1	Line Cov.	Branch Cov.	Pass@1	FM Call@1	Cor-rect@1	Line Cov.	Branch Cov.
No-Op Tests	1.00	0.00	0.00	0.33	0.33	1.00	0.00	0.00	0.26	0.36
Baseline Prompt	0.12	0.34	0.03	0.38	0.40	0.12	0.32	0.03	0.32	0.45
SymPrompt	0.41	0.49	0.15	0.48	0.44	0.41	0.35	0.12	0.36	0.35
Pynguin	-	-	-	0.72	0.64	-	-	-	0.68	0.57
SymPrompt Filtered	0.81	1.00	0.81	0.77	0.66	0.83	1.00	0.83	0.71	0.57

contrast, during Pynguin's mutation-based testing process, it randomly generates and executes many different test candidates, and mutations that fail to improve coverage (e.g., to errors) are discarded. Therefore, we also run a version of SymPrompt (Symprompt Filtered) where test suites with no working tests (i.e., that do not execute and call the focal method) are discarded. Under this setting, CodeGen2 SymPrompt compares favorably with Pynguin (77% line coverage, 66% branch coverage on average).

Case Studies. We found that SymPrompt improved test generation in two ways: it reduced errors in calling the focal method by giving more precise guidance, and it helped to generate higher coverage test cases by testing more paths. The case study shown in Figure 9 illustrates both of these cases for a focal method `burp` from the `pytutils` project. The test shown in Figure 9a only tests one of the two execution paths in the `burp` method, where the input parameter `filename` is set to a normal value, and has an error where the focal method is called with an incorrect parameter. In contrast, the SymPrompt generated tests call `burp` correctly and use test inputs to cover both the regular and `filename='-'` paths, even though `'-'` is not a natural name for a file.

Figure 10 illustrates how SymPrompt can benefit testing a method that is challenging for an SBST approach due to specific type requirements for its input parameters. The input types are specified in a comment that an SBST tool cannot leverage, and as a result it generates test inputs that cause an exception on the first line of the method (see Figure 10b). A language model however can use the comments for type hints along with the method signature to generate a correct method call that executes without errors (see Figure 10b).

4.2 RQ2: Training Data Memorization

Since training data memorization can bias results with language model evaluations, we evaluate performance separately on a subset of projects that are not included in CodeGen2's training data.

Evaluation. To evaluate the potential impact of training data memorization on the results shown in RQ1, we conduct a seperate performance evaluation exclusively on projects that were excluded from CodeGen2's training data. Since CodeGen2 is trained on a subset of the Stack [21], we use

```python
# test case for function burp
def test_burp():
    b = b'hello world!\n'
    f = 'temp.txt'

    burp(f, b, 'wb')

    # make sure we can write
    burp(f, b, 'wb', iter_by=1)
```

(a) Baseline test generation.

```python
def test_case_1():
    """
    Testcase 1 for burp(filename, contents, mode='w'):
    test case where filename == '-' and allow_stdout
    """
    burp('-', 'hello world')

def test_case_2():
    """
    Testcase 2 for burp(filename, contents, mode='w'):
    test case where not (filename == '-' and allow_stdout)
    """
    burp('/tmp/foo', 'hello world')
```

(b) SymPrompt test generation.

Fig. 9. Case study of a simple focal method `burp` in the `pytutils` project with two main execution paths, based on whether the input filename is `-` or not. A test generation using the baseline prompt executes the method twice, but does not check for `filename='-'` as a test input. Moreover, the send focal method call uses a nonexistent parameter, `iter_by`, preventing the test from fully executing. In contrast, SymPrompt tests both paths and uses method correctly. Note that in this case the method does not return a value, therefore SymPrompt does not prompt, *correctly*, for assertions on the return statement.

```python
def aes_cbc_decrypt(data, key, iv):
    """
    Decrypt with aes in CBC mode

    @param {int[]} data
    @param {int[]} key
    @param {int[]} iv
    @returns {int[]}
    """
    expanded_key = key_expansion(key)

    decrypted_data = []


    <decrypt_implementation>


    return decrypted_data
```

(a) Focal method.

```python
def test_case_1(self):
    """
    Test for aes_cbc_decrypt(data, key, iv):
    returns: decrypted_data
    """
    data = [1, 2, 3, 4, 5]
    key = [1, 2, 3, 4, 5]
    iv = [1, 2, 3, 4, 5]
    actual = aes_cbc_decrypt(data, key, iv)
```

(b) SymPrompt test generation.

```python
def test_case_1():
    try:
        bool_0 = True
        set_0 = {bool_0}
        str_0 = '?sIz0v</HZFS!_$'
        tuple_0 = ()
        var_0 = aes_cbc_decrypt(set_0, str_0, tuple_0)
    except BaseException:
        pass
```

(c) SBST test generation.

Fig. 10. Example test generations from an SBST tool (Pynguin) and LLM (CodeGen2) for focal method `aes_cbc_decrypt`. The method operates on arrays of ints, but an SBST approach is unable to infer types (even if they are specified in the comments, and generates test inputs that immediately raise an exception on the `key_expansion(key)` call. An LLM can infer input types from context and comments and therefore generate a test case with correctly typed inputs that execute the entire method without errors.

the AmIInTheStack tool [1] to identify three projects in the evaluation set that not included in the stack and evaluate on the focal methods drawn from these projects.

Observations. Table 1 shows results of this evaluation. Compared to the large scale evaluation results, CodeGen2's test generations with both baseline prompts and SymPrompt have lower rates of correct test generations and coverage. However, the tests generated with SymPrompt still have significantly higher rates of correct test generations ($\frac{.12}{.3} = 4\times$) and has 12.5% higher line coverage relative to the LLM baseline. These results indicate that SymPrompt is beneficial for generating more correct tests with higher coverage when focal methods are different from those seen in the training data.

Table 2. **RQ 3. Results (LHS):** Ablation results evaluated on 100 randomly sampled focal methods from the benchmark used in RQ1. The ablation results demonstrate that both type and dependency in calling context and path constraint prompts improve correct generations and coverage relative to baseline prompts, but path constraint prompts contribute significantly more to the overall improvement in correct generations and coverage demonstrated by SymPrompt for generations with CodeGen2. **RQ 4. Results (RHS):** Results with GPT. When evaluating GPT we use as a baseline a two stage prompt in which the model is first prompted to describe the method under test, and then generate tests based on both the method definition and the previously generated description. Although performance is similar for both prompting approaches without calling context, when SymPrompt is used without ablation it gives a relative improvement of 178% in Correct@1 rate and 1.05% relative improvement in coverage over baseline prompts.

	CodeGen2					GPT-4				
Method	Pass@1	FM Call@1	Correct@1	Line Cov.	Branch Cov.	Pass@1	FM Call@1	Correct@1	Line Cov.	Branch Cov.
Baseline Prompt	0.09	0.37	0.04	0.30	0.29	0.14	0.12	0.09	0.36	0.40
Constraints Only	0.27	0.49	0.17	0.49	0.38	0.15	0.15	0.10	0.39	0.43
Context Only	0.16	0.40	0.06	0.42	0.35	0.38	0.28	0.18	0.43	0.47
SymPrompt	0.50	0.65	0.26	0.53	0.42	0.46	0.39	0.25	0.74	0.74

4.3 RQ3: Design Choices

SymPrompt uses both selective type and dependency focal context and path constraint prompts to improve performance over baseline test completion prompts. We evaluate the impact of each of these components on SymPrompt's performance.

Evaluation. We conduct an ablation to evaluate how each of these methods contributes to performance improvements in isolation. When ablating focal context, we use the local context around the focal method based on prior work [22]. We evaluate on 100 randomly sampled focal methods from the benchmark used in RQ1.

Observations. Table 2 (LHS) shows results of the ablations. SymPrompt with no ablations achieves an overall 26% correct generation rate and 53% coverage on average. Ablating path constraint prompts but retaining calling context reduces the FM call rate and substantially reduces pass rate, indicating that the additional guidance from path constraint prompting is very beneficial for generating tests with correct focal method calls. The path constraint ablation also has significantly lower coverage, indicating that the path constraints serve to generate more thorough test cases.

The ablation of calling context also reduces the pass rate and FM call rate of generated tests, although much less than ablating path constraint prompting, and still has significantly better performance than a full ablation of both methods. These results indicate that while test generations with path constraint prompts benefit from using calling context, the additional guidance and structure provided to the model in the symbolic prompts are crucial to the performance improvements in correct generations and coverage by SymPrompt over baseline test generation prompts.

4.4 RQ4: Large Model Performance Impact

In addition to evaluating on an open source 16B parameter model, we also evaluate if the prompting strategy used by SymPrompt can benefit test generations with a larger and better trained model. Recent work has shown that increasingly large scale language models exhibit *emergent abilities* that are completely absent in smaller scale models [37]. In this evaluation we show that a significantly larger language model, GPT-4, exhibits the ability to reason precisely about path constraints in a focal method and generate its own path constraint prompts in a zero-shot setting. We find that prompting the model to approach test generation in this way is very beneficial for generating high coverage testsuites.

Evaluation. We evaluate with GPT-4 [28], a significantly larger model than CodeGen2 that benefits from much more extensive training. We found that GPT-4 was capable of generating precise descriptions of the execution paths and constraints in focal methods given a 0-shot prompt, so instead of generating path prompts with static analysis, we prompt the model to describe execution paths and then embed each path description in a test docstring (See Figure 8c).

Recent works have demonstrated that test generation with GPT models benefit from multi-stage prompts that incorporate description and planning [23, 43], therefore as a baseline we use a 2 stage prompt that first asks the model to describe the intent of the method under test and then to generate a testsuite based on both the description and the focal context based on [43].

Observations. Table 2 (RHS) summarizes the results of our evaluation with GPT-4. We found that calling contexts in the generated tests were especially important for GPT-4s generations, since the model was prone to hullucinating incorrect import statements or using undefined classes and objects that were defined in the focal context. Overwriting the model-generated imports with the classes and objects identified in the calling context greatly reduced errors when the tests were executed. Using the path constraint prompts generated by GPT-4 in isolation did not lead to significant performance improvements over the baseline describe and generate prompts, but when used in conjunction with calling context the path constraint prompts improved the average coverage of generated tests over the baseline describe-generate prompt by a factor of more than 2, from 36% to 74%. We hypothesize that the significant performance difference occurs because path constraint prompts are effective for generating more high coverage testsuites, but most of those tests fail due to errors with imports and undefined variables when calling contexts are not used.

GPT vs. Analysis-Generated Path Prompts. Figure 11 shows a comparison of the path descriptions generated by GPT-4 and SymPrompt's static analysis on the method `_serialize` shown in Figure 12a. We found that, compared to the static analysis, GPT-4 was able to generate more natural path constraints while still giving precise descriptions. Figure 12 shows the test cases generated for two GPT path prompts on the method `_serialize`.

Path-Following Generation Accuracy. In addition to measuring the impact of SymPrompt on overall coverage, we conducted a small study of how effective the models are at generating tests that follow the specific paths specified in the prompts by manually examining 10 generated tests in the GPT-generated set. Of these, 6 out of 10 followed their specified paths. The missing four cases are either due to deeply nested branching or exception handling–the model either did not generate correct preconditions for deeply nested branches or error-handling paths. We also observed qualitatively that CodeGen2 usually generates test inputs that follow specified paths when the constraint involves an input parameter, as shown in Figure 1, while GPT-4 is also able to generate test inputs for more complex path constraints involving class variables and external function calls, as shown in Figure 12.

5 THREATS TO VALIDITY & DISCUSSION

Model and Benchmark Validity: Our evaluations focus on open source Python projects and utilize specific language models (CodeGen2 and GPT-4). This restricts the generalizability of our findings to other languages or models. However, these are two large state-of-the-art models. Also, none of the methods are specific to Python. So we expect the findings will hold consistently in other settings.

Memorization Validity: Although we use the AmIInTheStack tool to prevent training data memorization from biasing our RQ 2 results, the possibility remains that CodeGen2 could have seen some of the focal methods or similar code in its training data. However, since these models are not explicitly trained for test generation tasks, we think this threat is minimal.

```
def test_serialize_1(self):
    """
    testcase for _serialize(self, value, attr, obj, **kwargs)
    where allow_none == True and value is None
    returns: None
    """
```

(a) SymPrompt path prompt.

```
def test_serialize_1(self):
    """
    1. **Path 1**
       - The `allow_none` attribute of the instance is `True` and `value` is `None`.
         - In this case, the method returns `None`.
    """
```

(b) GPT-4 path prompt.

Fig. 11. Comparison of SymPrompt-generated path prompt to GPT-4 generated path prompt. GPT-4 is capable of generating precise execution path descriptions with more natural language. We construct path prompts by using a markdown parser to extract each path description and embed them as docstrings for each test generation.

Metric Validity: Our evaluations are based on the metrics Pass@1, FM Call@1, Correct@1, and Line Coverage. However, these metrics might not capture the full complexity or usefulness of a generated test case.

Test Generation in a Regression Setting: In this paper, we operated under the assumption that the tests are generated within a regression setting, assuming the correctness of the underlying focal method implementation. As a result, our generated tests may not uncover any implementation bugs. This approach to testing also has limitations, particularly when the implementation of the focal method is not yet finalized, a scenario commonly encountered in continuous development environments.

6 RELATED WORK

Our work relates to the following areas: Search Based Software Testing, Symbolic Test Generation, Test Generation with LLMs, and Hybrid LLM-SBST test generation.

Search Based Software Testing (SBST) and Symbolic Approaches. Evosuite is an SBST regression testing framework for java that generates regression tests based mutation testing and coverage guided randomized test generation [13, 14, 16]. Randoop uses coverage guided randomized test generation for Java in conjunction with sanity checking oracles to check for common classes of bugs like NullException errors [29, 30]. Pynguin applies SBST to generate regression tests for Python [24, 25]. PeX is a whitebox test generation with concolic execution and constraint solver [35]. Korat tests based on a formal specification [8]. Dart performs concolic test generation [18]. Cute performs concolic testing for c [34]. Our approach is conceptually related to coverage driven SBST approaches and Concolic Execution because it formulates test generation as a constraint solving problem for the LLM, where the LLM is guided to generate a test that will follow a specific execution path. However, instead of performing symbolic constraint solving to follow specific execution paths we give the LLM access to the focal method source code.

LLM Regression Test generation. Athenatest finetune pretrained transformers on paired method-test data and show including more focal context in prompt leads to higher coverage test generations [36]. Bariess et. al. compare Codex-generated tests to randoop based on a one-shot test example and show the codex-generated tests achieve better coverage(arxiv) [7]. A3Test apply post-processing to correct test naming errors in model generations using PLBart(arxiv) [3]. Hashtroudi

```
        def _serialize(self, value, attr, obj, **kwargs):
            if self.allow_none and value is None:
                return None
            for type_, schema_ in self.desc.items():
                if _issubclass_safe(type(value), type_):
                    if is_dataclass(value):
                        res = schema_._serialize(value, attr, obj, **kwargs)
                        res['__type'] = str(type_.__name__)
                        return res
                    break
                elif isinstance(value, _get_type_origin(type_)):
                    return schema_._serialize(value, attr, obj, **kwargs)
            else:
                warnings.warn(
                    f'The type "{type(value).__name__}" (value: "{value}") '
                    f'is not in the list of possible types of typing.Union '
                    f'(dataclass: {self.cls.__name__}, field: {self.field.name}). '
                    f'Value cannot be serialized properly.')
            return super()._serialize(value, attr, obj, **kwargs)
```

(a) Focal method.

```
class TestUnionFieldSerialize(unittest.TestCase):

    def test_serialize_return_none(self):
        """
        1. **Path 1**
           - The `allow_none` attribute of the instance is `True` and `value` is `None`.
            - In this case, the method returns `None`.
        """
        mock_field = _UnionField({}, None, None, allow_none=True)
        self.assertIsNone(mock_field._serialize(None, None, None))

    def test_serialize_dataclass(self):
        """
        2. **Path 2**
           - The `allow_none` attribute of the instance is `False` or `value` is not `None`.
           - For each `type_`, `schema_` pair in `self.desc.items()`,
`_issubclass_safe(type(value), type_)` is `True`.
            - If `value` is a dataclass, the method serializes `value` using
`schema_._serialize()`, adds a `__type` field to the result, and returns the result.
            - If `value` is not a dataclass, the loop breaks and the method proceeds to the next
execution path.
        """
        @dataclasses.dataclass
        class MockDataClass:
            pass

        mock_schema = MagicMock()
        mock_schema._serialize.return_value = {}

        mock_field = _UnionField({MockDataClass: mock_schema}, None, None)
        result = mock_field._serialize(MockDataClass(), None, None)
        self.assertEqual(result, {'__type': 'MockDataClass'})

<more tests follow>
```

(b) GPT-4 generated tests.

Fig. 12. Case study showing GPT generations with path constraint prompts.

et al. show that models finetuned on existing project testsuites output higher quality test generations(arxiv) [19]. MuTAP uses mutation testing to guide LLM test generations towards tests that are more likely to detect bugs and show improved bug detection with Codex and Llama2

on Defects4j(arxiv) [11]. These approaches use fixed prompting strategies, our work focuses on developing code-aware prompts that guide the model to generate a high coverage set of tests.
ChatGPT test generation. Testpilot generates javascript tests with ChatGPT 3.5 using a 0-shot test prompt and then iteratively adds additional code and documentation context if the generated tests fail(arxiv) [32]. ChatUnitTest generates tests using adaptive focal context based on the maximum focal length and then applies both rules-based repair and self-debugging based on error messages when generated tests fail [40]. ChatTester similarly generates tests based on the focal context and then prompts ChatGPT with error messages when generated tests fail(arxiv) [43]. These approaches all focus on constructing the focal context and then fixing ChatGPT's generations. In contrast our work focuses on developing prompts to guide the model to test each execution path in the focal method, improving testsuite coverage.
Hybrid SBST-LLM regression test generation. Codamosa runs Pynguin, an SBST tool for python, and iteratively calls Codex to generate additional testcases for methods with low coverage [22]. Our work focuses on improving LLM testsuite generations instead.

7 CONCLUSION

This paper introduces SymPrompt, a novel approach to test generation with LLMs by decomposing the test suite generation process into a structured, code-aware sequence of prompts. SymPrompt significantly enhances the generation of comprehensive test suites with a recent open source Code LLM, CodeGen2, and achieves substantial improvements in the ratio of correct test generations and coverage. Moreover, we show that when given a specific instruction prompt to analyze execution path constraints, GPT-4 is capable of generating its own path constraint prompts, which improves the coverage of its generating testsuites by a factor of 2× over prompting strategies from recent prior work.

REFERENCES

[1] [n. d.]. Am I in the stack? https://huggingface.co/spaces/bigcode/in-the-stack.
[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4998–5007. https://doi.org/10.18653/v1/2020.acl-main.449
[3] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).
[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 38–49.
[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[7] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).
[8] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 123–133.
[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[10] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211.
[11] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2023. Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. *arXiv preprint arXiv:2308.16557* (2023).

[12] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[13] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[14] Gordon Fraser and Andrea Arcuri. 2013. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE sixth international conference on software testing, verification and validation*. IEEE, 362–369.

[15] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[16] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering* 20 (2015), 611–639.

[17] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 47–54.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[19] Sepehr Hashtroudi, Jiho Shin, Hadi Hemmati, and Song Wang. 2023. Automated Test Case Generation Using Code Models and Domain Adaptation. *arXiv preprint arXiv:2308.08033* (2023).

[20] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[21] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[22] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.

[23] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, and Shing-Chi Cheung. 2023. Finding Failure-Inducing Test Cases with ChatGPT. *arXiv preprint arXiv:2304.11686* (2023).

[24] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[25] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated unit test generation for python. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 9–24.

[26] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2 (1976), 308–320. https://api.semanticscholar.org/CorpusID:9116234

[27] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).

[28] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[29] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[30] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[31] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* 1 (2002).

[32] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

[33] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[34] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[35] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In *International conference on tests and proofs*. Springer, 134–153.

[36] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[37] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing*

Systems 35 (2022), 24824–24837.

[39] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943

[40] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[41] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).

[42] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[43] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023).

Received 2023-09-29; accepted 2024-01-23