STRUT: Structured Seed Case Guided Unit Test Generation for C Programs using LLMs

JINWEI LIU, Xidian University, China
CHAO LI, Beijing Institute of Control Engineering, China and Beijing Sunwise Information Technology, China
RUI CHEN, Beijing Institute of Control Engineering, China and Beijing Sunwise Information Technology, China
SHAOFENG LI, Xidian University, China
BIN GU, Beijing Institute of Control Engineering, China
MENGFEI YANG, China Academy of Space Technology, China

Unit testing plays a crucial role in bug detection and ensuring software correctness. It helps developers identify errors early in development, thereby reducing software defects. In recent years, large language models (LLMs) have demonstrated significant potential in automating unit test generation. However, using LLMs to generate unit tests faces many challenges. 1) The execution pass rate of the test cases generated by LLMs is low. 2) The test case coverage is inadequate, making it challenging to detect potential risks in the code. 3) Current research methods primarily focus on languages such as Java and Python, while studies on C programming are scarce, despite its importance in the real world. To address these challenges, we propose STRUT, a novel unit test generation method. STRUT utilizes structured test cases as a bridge between complex programming languages and LLMs. Instead of directly generating test code, STRUT guides LLMs to produce structured test cases, thereby alleviating the limitations of LLMs when generating code for programming languages with complex features. First, STRUT analyzes the context of focal methods and constructs structured seed test cases for them. These seed test cases then guide LLMs to generate a set of structured test cases. Subsequently, a rule-based approach is employed to convert the structured set of test cases into executable test code. We conducted a comprehensive evaluation of STRUT, which achieved an impressive execution pass rate of 96.01%, along with 77.67% line coverage and 63.60% branch coverage. This performance significantly surpasses that of the LLMs-based baseline methods and the symbolic execution tool SunwiseAUnit. These results highlight STRUT's superior capability in generating high-quality unit test cases by leveraging the strengths of LLMs while addressing their inherent limitations.

Additional Key Words and Phrases: Automatic Unit Test Generation, Large Language Models, Structured Test Cases

ACM Reference Format:
Jinwei Liu, Chao Li, Rui Chen, Shaofeng Li, Bin Gu, and Mengfei Yang. 2025. STRUT: Structured Seed Case Guided Unit Test Generation for C Programs using LLMs. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA093 (July 2025), 22 pages. https://doi.org/10.1145/3728970

Authors' addresses: Jinwei Liu, Xidian University, Xi'an, China, liujinwei@stu.xidian.edu.cn; Chao Li, Beijing Institute of Control Engineering, Beijing, China and Beijing Sunwise Information Technology, Beijing, China, lichao@sunwiseinfo.com; Rui Chen, Beijing Institute of Control Engineering, Beijing, China and Beijing Sunwise Information Technology, Beijing, China, chenrui@sunwiseinfo.com; Shaofeng Li, Xidian University, Xi'an, China, lishaofeng@xidian.edu.cn; Bin Gu, Beijing Institute of Control Engineering, Beijing, China, gubinbj@sina.com; Mengfei Yang, China Academy of Space Technology, Beijing, China, yangmf@bice.org.cn.

ACM 2994-970X/2025/7-ARTISSTA093
https://doi.org/10.1145/3728970

1 INTRODUCTION

Unit testing plays a crucial role in ensuring software correctness. It helps developers identify potential issues in the early stages of program development, effectively reducing the occurrence of defects and thereby lowering development costs. However, manually creating and maintaining unit test cases can be laborious and time-consuming[14, 24, 30].

To solve this problem, researchers have proposed various methods to automate the unit test generation process. Traditional unit testing tools adopt search-based[10, 17, 19] or constraint-based techniques[27, 32]. While these methods can generate test cases to enhance software test coverage, they have several key limitations in terms of test readability, scenario coverage, and the quality of assertions, which increases maintenance complexity. Deep learning-based approaches[9, 37] can learn from real-world focal methods and generate test cases that closely resemble those written by developers, offering improved readability and understandability. Nevertheless, they cannot correct simple errors through interaction with the model, resulting in incorrect test cases and a low pass rate for most generated test cases[21]. With the rise of LLMs, their excellent performance in tasks such as code comprehension[28] and code generation[20] has been widely recognized. Researchers are actively exploring leveraging LLMs to overcome the limitations of test generation techniques[21, 39].

Although existing LLMs-based methods can sometimes generate unit tests that are well-explained, easy to understand, and have a certain degree of accuracy and coverage, there are still significant limitations and drawbacks. First, existing test case generation works based on LLMs are targeted at specific languages such as Java and Python. However, C program language, which is widely used in critical real-world applications such as operating systems and network devices, lacks dedicated support. According to [40], there exists a substantial disparity in the performance of LLMs on different language tasks, making these methods potentially difficult to extend to C language. Second, when confronted with development tasks in real-world scenarios, existing works still grapple with issues such as low compilation success rate[39], low execution pass rate[21], and low coverage rate[38] of generated test cases, making it difficult to meet practical needs. For example, TestART[21] shows a pass rate of 78.55% for generating test cases for Java methods. When HITS[38] handles complex methods, the line and branch coverage ratios only reach 55.09% and 48.12%, respectively. Upon in-depth analysis, our key insight is that the fundamental limitations of existing LLMs-driven unit test generation methods primarily arise from their reliance on LLMs for code generation. These methods are hampered by the models' capacity to generate programs in specific programming languages. Specifically, when these languages exhibit complex features and test cases demand more advanced functionalities, the constraints become evident.

To overcome the aforementioned limitations and improve the effectiveness of unit testing, we have developed a method named STRUT which is based on LLMs. The key idea of STRUT is to introduce structured test cases with simple syntax and structured constraints as a bridge between complex program language and LLMs, thereby resolving the limitations of LLMs in language features. The structured testing seed case is modeled as a triple consisting of input, output, and stub functions. The inputs consists of the test data used in the test case, and the outputs represents the expected value for the test. To ensure the isolation of the unit test, we stub out functions related to the focal method, simulating their behavior within the stub functions to prevent any influence on the focal method. By adopting these structured test cases, STRUT mitigates the limitations of LLM in directly generating code with complex features of the program language. Subsequently, the structured test cases suite is transformed into executable test code through a rule-based approach. This transformation process ensures that the generated test cases meet the syntactic and semantic

requirements of the program language, thereby enhancing the correctness and reliability of test case generation.

To evaluate the effectiveness of the proposed method, we conducted a comprehensive assessment of STRUT. Our experiments utilized a diverse set of datasets, including three real-world industrial embedded software programs, four open-source software programs, and a carefully designed artificial dataset, with a total of 1709 focal methods. We compared the proposed method STRUT with SunwiseAUnit[6] and the naive context-guided method based on GPT-4o[8]. SunwiseAUnit is a widely used commercial C/C++ unit testing tool with a symbolic execution based test generation engine. GPT-4o currently represents one of the most advanced LLMs. The experimental results indicate that STRUT outperforms all baseline methods, achieving the highest line and branch coverage of 77.67% and 63.60%, respectively. Compared to GPT-4o, STRUT improves line coverage by 37% and branch coverage by 33%. Additionally, it achieved an impressive execution pass rate of 96.01%. These remarkable results clearly demonstrate that STRUT significantly enhances the unit testing performance of C programs by leveraging the power of LLMs.

Our contributions are summarized as follows:

- We are the first to propose generating structured test cases with simple syntax and structured constraints as a bridge between complex programming languages and LLMs, effectively addressing the limitations of LLMs in handling complex features of the program language.
- We developed STRUT and implemented it into a practical tool. The tool is designed based on the concept of structured test cases, enabling developers to easily generate high-quality unit tests.
- We conducted a comprehensive evaluation of STRUT using a diverse range of datasets. The evaluation results demonstrate the remarkable effectiveness of STRUT in generating unit tests.

2 APPROACH

In this section, we introduce STRUT, a methodology for generating unit tests guided by structured seed cases using LLMs. Although we focus on C programs in this paper, we believe that this approach can be extended to other programming languages. The core concept behind STRUT is the use of a structured representation of unit test cases to bridge the gap between complex C code and large language models, addressing LLMs limitations in test code generation. As illustrated in Figure 1, the STRUT workflow is divided into three phases: context building, test generation, and run & optimization.

The first phase applies a static analysis-based method to extract the necessary context to generate tests for the focal method. This phase aims primarily to provide sufficient contextual information for subsequent test generation using LLMs, with a structured seed test case as a crucial component guiding effective unit test generation. Next, using this contextual information, we create a prompt containing the focal method code block, its dependencies, and the structured seed test case, which guides the LLM in generating additional structured test cases through a one-shot learning approach. These structured test cases are then transformed into compilable code through a rule-based code conversion process. After compilation and execution, we further extend the test suite based on coverage information, ultimately achieving a test suite with higher coverage. The following sections provide a detailed explanation of each phase in the STRUT workflow.

2.1 Context Building

Constructing a concise and minimal context for focal methods is essential for LLMs to produce high-quality test cases. We utilize static analysis to extract the minimal necessary context for

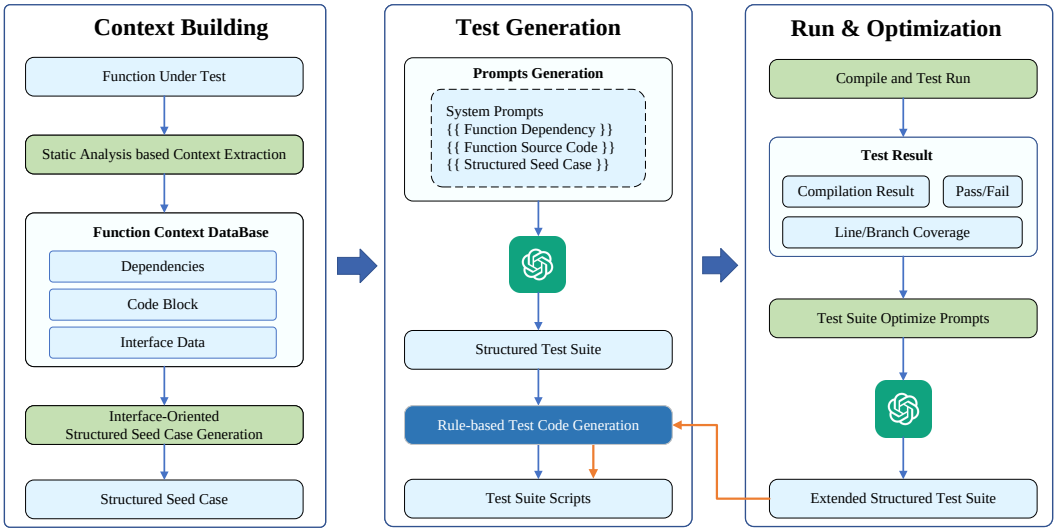

Fig. 1. Method Overview

each focal method and adopt an interface-oriented approach to construct structured seed cases, providing targeted guidance for LLMs to generate high-quality, structured test cases.

2.1.1 Static Analysis based Context Extraction. We employ static analysis techniques to extract the focal method's dependencies and store them in a context database. We build the context database from three critical areas: the focal method's dependencies, the code block of the focal method, and the interface data.

- Dependencies: The dependencies refer to the code segments required to compile the focal method independently. These include type declarations, global variable definitions, preprocessor directives, and function declarations involved in the focal method's implementation. Once all the dependencies are set up, the focal method can be compiled independently.
- Code Block: The code block is the sliced source code of the focal method extracted from the C source file. We slice each focal method from the source files and combine them with the dependencies code segments of the focal method to form a independently compilable C file.
- Interface Data: The interface data encompasses all externally exposed interfaces of the focal method. Includes metadata for the definition of the type, formal parameters, global variables, return values, and functions invoked within the focal method.

We use a dependency extraction algorithm to identify the focal method's dependencies, as detailed in Algorithm 1. The algorithm involves two main tasks: analyzing data types used by the focal method and acquiring related dependencies. Lines 1–5 focus on retrieving the focal method and analyzing its signature and type dependencies. In line 6, we obtain the function body, and from lines 7 to 21, we iterate over its statements. Lines 8–11 check for declarations, retrieving declared types and dependencies if found. Lines 12–17 handle variable references, identifying global variables and their dependencies. Lines 18–20 involve checking function calls and retrieving invoked function declarations. The method *FindTypeDependency* gathers relevant dependencies for a given type using provided type information. Line 25 checks if the type is declared externally, retrieving additional information if so. Line 27 determines if the type is composite, recursively searching for its subtypes' dependencies.

Algorithm 1 Dependency Extraction Algorithm

Input: $CFile$
Output: $FM, Deps, globalVars$

$FM \leftarrow GetFocalMethod(CFile)$
$FM_{\text{sig}} \leftarrow GetSignature(FM)$
$FM_{\text{sig\_type}} \leftarrow GetSignatureTypes(FM_{\text{sig}})$
$Deps \leftarrow [], globalVars \leftarrow []$
$Deps \leftarrow$ FindTypeDependency$(FM_{\text{sig_type}})$
$FM_{\text{body}} \leftarrow GetFunctionBody(FM)$
for each $stmt \in FM_{\text{body}}$ **do**
    if $isDeclaration(stmt)$ **then**
        $declType \leftarrow GetType(stmt)$
        $Deps \leftarrow Deps +$ FindTypeDependency$(declType)$
    end if
    if $isVarReferences(stmt)$ *and* $isGlobalVar(stmt)$ **then**
        $globalVar \leftarrow GetGlobalVar(stmt)$
        $globalVarType \leftarrow GetVarType(globalVar)$
        $Deps \leftarrow Deps +$ FindTypeDependency$(globalVarTpe)$
        $globalVars \leftarrow globalVars + globalVar$
    end if
    if $isFunctionCall(stmt)$ **then**
        $Deps \leftarrow getFunctionDeclaration(stmt)$
    end if
end for

function $FindTypeDependency(types)$
    $Deps \leftarrow []$
    for each $type \in \{types\} do$ **do**
        if $isExternalType(type)$ **then**
            $Deps \leftarrow Deps + FindExternalTypeDeclaration(type)$
            if $isCompositeType(type)$ **then**
                for each $child_{\text{type}} \in type$ **do**
                    $FindTypeDependency(child_{\text{type}})$
                end for
            end if
        end if
    end for
    return $Deps$
end function

For the interface data, we analyze the function signature to obtain the parameter list and return value type of the focal method. Additionally, we examine the function body to identify the global variables utilized and the function calls used within it. After collecting all relevant data, we extract the data types from the dependencies to construct the interface data. To fully understand the functions invoked, we also gather the interface data for the functions called within the focal method.

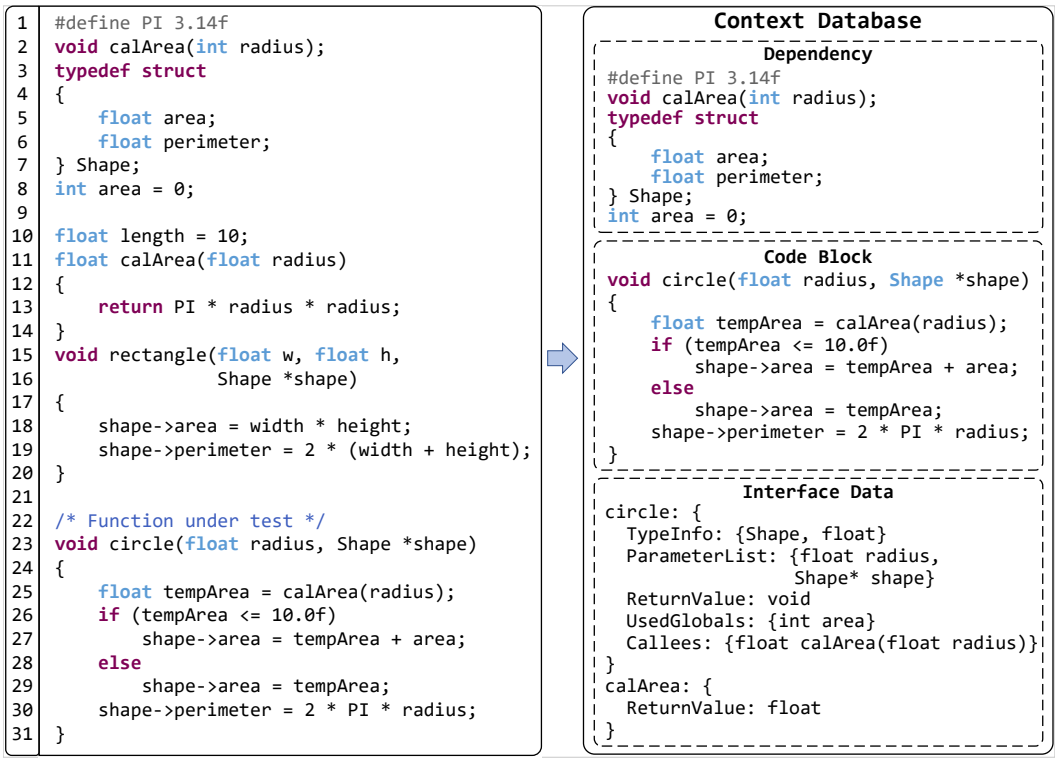

Fig. 2. Context Database Building Example

Figure 2 shows an example of context building. First, we employ Algorithm 1 to identify the dependencies of the focal method. Specifically, we extract the macro definition *PI*, the external type *Shape*, the global variable *area*, and the function *calArea* invoked by the target method. Subsequently, we slice the focal method code segment from the entire C file and combine it with the dependencies to create a standalone compilable program. Finally, we analyze the parameter list, return values, global variables, and their type information for the *circle* function. We conduct a similar analysis for the called function *calArea*, gathering the relevant interface data.

2.1.2 Interface-Oriented Structured Seed Case Generation. The structured seed case is the most critical component of STRUT. In this section, we introduce our seed case schema and then the construction methods for interface-oriented seed test cases.

Figure 3 illustrates the schema of structured test cases, which are defined as a ternary group consisting of Inputs, Outputs, and Stub functions.

- The *Inputs* consists of multiple pairs, each containing an *ExprInput*, *Type* and a *Value*. The *ExprInput* represents the expression for test input data, such as variable names or dereferenced pointers, the *Type* represents the data type for the *ExprInput*, and the *Value* represents the test data for the *ExprInput*.
- Similar to the *Inputs*, the *Outputs* includes *ExprOutput*, *Type*, and *Value*. *Exproutput* represents return values and data that change persistently during testing, such as pointer parameters and global variables.

- To ensure the isolation of unit tests, we stub all functions called by the focal method. The *Stubs* contains the signature information of the called functions, with each function including its return value or the values it may modify. The *StubFunctionSig* is the signature of the called function. We use *ExprStub* to describe the side effects when calling this function, such as modifications to persistent data or the return value.

We use an interface-oriented approach to construct structured seed test cases. In Figure 4, we present a structured seed case generated from the focal method shown in Figure 2. First, we scan the interface data for the parameters *radius*, shape, and global variable *area*. We use these data as test inputs. From the *Callee* data in the interface, we see that the focal method circle calls the *calArea* function. Based on our analysis of *calArea*, we find that only its return value affects the focal method *circle*, so we only need to include a return value example for *calArea* in the stubs. Finally, we find that the *shape's* members are modified during testing. Since this parameter is a pointer type, we provide output examples for the *shape's area* and *perimeter* members.

We generate a default value for each expression based on its type. These default values can help the LLM better understand the data types of the assignment targets. For each test data, we set the expressions and assign default values according to the following rules:

(1) Basic types: The variable name is used as the expression, with values assigned according to the type(e.g., int: 0, float: 0.0).
(2) Composite types: Internal members of the composite types are treated as expressions, and values are assigned based on the type of each member.
(3) Pointer types: If the pointer is valid, it is assigned a value corresponding to the target type. Otherwise, it is set to NULL.
(4) Array types: The first element of the array is initialized according to its type.

If a structured pointer contains nested structures, such as a linked list, we initialize the head node and recursively assign default values to subsequent nodes until a reasonable depth is reached. We set the assignment depth to 2 levels, which not only provides effective guidance to the LLMs but also ensures the efficiency of our algorithm.

2.2 Test Generation

The test generation phase aims to create prompts that help the LLM generate high-quality structured test cases. The prompt contains the focal method context, source code, and a structured seed case. STRUT employs a one-shot prompting strategy. We use the structured seed case generated in the context-building phase for each function, guiding the LLM in generating structured test cases.

2.2.1 Prompts Generation and Structured Test Suite. The prompt for generating test cases is presented in Figure 5. It consists of the context of the focal method, its definition, and a seed case. The prompt explicitly indicates the need to generate mock data for the called functions. In the seed cases, we provide data that might be altered by the called functions, allowing the LLM to determine assignments based on the test branches. We employ a one-shot learning approach, presenting the seed case as an example, to demonstrate the standard test case format to LLMs. The seed case explicitly identifies the objects that must be assigned values during testing and the expected test output format. The key to this strategy lies in offering the LLM a clear template, ensuring that the generated test cases accurately match the anticipated input and output formats. Furthermore, the generated test cases exhibit greater consistency and reliability by providing the model with sufficiently detailed context and examples. This approach enhances the LLM's precision in generating test data and minimizes ambiguous or uncertain results.

There may be a large amount of input data when dealing with functions with numerous branches. To alleviate the burden on the LLM and enhance the reliability of the output, we only require the

```
TestCase = {Inputs, Outputs, Stubs}
Inputs = [
    {ExprInput₁, Type₁, Value₁},
    {ExprInput₂, Type₂, Value₂},
    ...
]
Outputs = [
    {ExprOutput₁, Type₁, Value₁},
    {ExprOutput₂, Type₂, Value₂},
    ...
]
Stubs = [
    {
        StubFunctionSig₁,
        [
            {ExprStub₁, Type₁, Value₁},
            {ExprStub₂, Type₂, Value₂}
        ]
    },
    {
        StubFunctionSig₂,
        [
            {ExprStub₁, Type₁, Value₁},
            {ExprStub₂, Type₂, Value₂}
        ]
    },
    ...
]
```

Fig. 3. Structured Test Case Schema

```
{
  "cases":[{
      "inputs":[{
          "expr": "radius",
          "type": "float",
          "value": 0.0
      },{
          "expr": "shape->area",
          "type": "float",
          "value": 0.0
      },{
          "expr": "shape->perimeter",
          "type": "float",
          "value": 0.0
      },{
          "expr": "area",
          "type": "int",
          "value": 0
      }],
      "stubs":[{
          "called function":"float calArea(float radius)",
          "changed variable":[{
              "expr":"returnValue",
              "type": "float",
              "value":0.0
          }]
      }],
      "outputs":[{
          "expr":"shape->area",
          "type": "float",
          "value":0.0
      },{
          "expr":"shape->perimeter",
          "type": "float",
          "value":0.0
      }]
  }]
}
```

Fig. 4. A Structured Seed Case For *circle*

LLM to ensure that each test case conforms to the seed case format without organizing all outputs into a single JSON data structure. This can further reduce the likelihood of errors by the LLM.

Please generate some test cases for the following C function. I will provide you a test case example with json format. This example is divide into three parts: inputs, outputs, stubins. We use 'expr', 'type' and 'value' key to represent an input or output data. In the stubins list, I'll provide the called function signature as well as the called function return value or pointer type parameters that the called function may change, and you'll need to decide which data to assign values to based on the test case.
Please continue to generate with this format. Satisfy the following requirements.
1. Assign values to the data according to the situation you want to test.
2. Try to cover all branches.
3. Use the stub function to simulate the return value of the called function. I will provide you with all the data that the called function may change. It is up to you to decide which values to change in each test.
4. Output legal json format for each test case.

{{ context }}
{{ focal method }}
{{ seed case }}

Fig. 5. Test Cases Generation Prompts

2.2.2 Rule-based Test Code Generation. Once structured test cases are generated, STRUT employs a rule-based code transformation method to convert these cases into C test code. Each test case corresponds to two main components in the test code: stub functions and the test case function.

Stub functions are used to simulate the external calls that the focal method depends on, while the test case function is responsible for preparing the input data, executing the target function, and validating its output.

The generation rules for stub functions are as follows:

(1) A function with an empty body is created based on the "called function" from the stubs.
(2) According to the "changed variable" in the stubs, the stub function assigns values to variables or generates return values within the function body.

The generation rules for the test case function are as follows:

(1) Based on the *expr* in the *inputs*, create a variable declaration statement for the expression.
(2) Create an assignment statement for the expression based on the *value* in the *inputs*.
(3) Pass the created variables as parameters to the focal method in order to test it.
(4) Generate an assertion statement based on the *expr* and *value* in the *outputs* to check whether the variable meets the expected outcome.

These rules are designed based on the structured test case schema and are manually specified to ensure compatibility with C programs. Figure 6 illustrates an example of converting a structured test case into a corresponding test code. The left side of the figure shows the content of the structured test case, while the right side displays the resulting test code. For example, lines 2–4 of the test code show the *calArea* stub function generated from the *stub* section of the structured test case. This function returns 3.14, simulating the stub behavior specified in the test case. Lines 12–15 in the test code set the input variables, as indicated by the *inputs* section of the structured test case. Lines 19–20 represent the result validation section, corresponding to the *outputs* part of the structured test case, where the *ASSERT_FLOAT_EQ* macro checks whether *shape->area* is 8.14 and *shape->perimeter* is 6.18 after function execution.

The test code generated with this rule-based transformation approach forms a complete translation unit with the target function and its dependencies. This unit can be compiled and executed to produce the test results for the test case.

2.3 Run & Optimization

In the test run and optimization phase, according to Figure 1, we first compile and run the tests; we collect compilation and runtime results, as well as line and branch coverage metrics. Based on the coverage analysis results, we identify the need for optimization. Specifically, we optimize only focal methods with coverage below 80% once to achieve good testing efficiency and reduce costs. Then, we collect information on the uncovered branches and analyze whether the true or false condition caused the lack of coverage. The uncovered branch code and relevant information are integrated, and feedback is provided to the LLM, guiding it to regenerate test cases targeting that branch. Our optimized prompt is illustrated in Figure 7. In the prompt, we instruct the LLM not to generate duplicate test cases and allow it to independently determine the path conditions that must be satisfied to reach this branch. Since newly generated test cases may cover branches that have already been tested, to reduce redundancy, we discard test cases that do not increase branch coverage. In other words, we only add a generated test case to the test suite if it covers an uncovered branch.

To reduce the overhead of repeated LLMs requests during each feedback optimization, we implement a simplified strategy, performing only a single feedback iteration. This approach is informed by preliminary evaluation data, which show that additional feedback iterations have minimal impact on the final coverage. Our experimental evaluation results further validate this finding.

3 IMPLEMENTATION

We have implemented the STRUT methodology as a tool capable of testing real-world C programs. This tool is built upon the commercial C/C++ unit testing framework, SunwiseAUnit[6], as a plugin

Generated structured Test Case

```
{
  "cases":[{
      "inputs":[{
          "expr": "radius",
          "type": "float",
          "value": 1.0
      },{
          "expr": "shape->area",
          "type": "float",
          "value": 0.0
      },{
          "expr": "shape->perimeter",
          "type": "float",
          "value": 0.0
      },{
          "expr": "area",
          "type": "int",
          "value": 5
      }],
      "stubs":[{
          "called function":"float calArea(float radius)",
          "changed variable":[{
              "expr": "returnValue",
              "type": "float",
              "value": 3.14
          }]
      }],
      "outputs":[{
          "expr": "shape->area",
          "type": "float",
          "value": 8.14
      },{
          "expr": "shape->perimeter",
          "type": "float",
          "value": 6.18
      }]
  }]
}
```

Generated Test Code

```
1  /* stub function */
2  float calArea(float radius) {
3    return 3.14;
4  }
5
6  /* generated test case 0 for function circle */
7  void circle_test_0() {
8    /* data preparation */
9    float radius;
10   Shape* shape;
11   shape = (Shape *)malloc(sizeof(Shape));
12
13   /* input value assignment */
14   radius = 1.0;
15   shape->area = 0;
16   shape->perimeter = 0;
17   area = 5;
18
19   /* test execution */
20   circle(radius, shape);
21
22   /* result validation */
23   ASSERT_FLOAT_EQ(shape->area, 8.14);
24   ASSERT_FLOAT_EQ(shape->perimeter, 6.18)
25 }
```

Fig. 6. An Example of Rule-based Test Code Generation

> Well done. But the following branches are not covered. I will provide you with the uncovered conditions. Please generate test cases for the uncovered branches. The test cases you design need to be able to reach this branch and cover this branch. Do not output test cases that are repeated with the previous ones.
>
> 1. if (a > b): true condition uncover
> 2. if (x==1): false condition uncover

Fig. 7. Test Suite Optimization Prompts Used By STRUT

for a test case generation engine. By reusing SunwiseAUnit's features for project preprocessing, compilation, test execution, and coverage analysis, we significantly reduced the implementation cost. We describe some details of implementation in the following aspects.

Context Extraction. We used a robust, commercial-grade C parser based on ANTLR V3 to analyze the target C program, extracting each function's type metadata and interface data, which are then stored in a JSON file for persistence. The function's code block, along with its dependencies, is extracted using a Python script built on libclang. These two sets of information are merged and saved in a standalone .c file. The JSON file storing type metadata and interface data is then parsed to construct a structured seed test case, which is also persisted in JSON format. This component comprises approximately 1,000 lines of Python code and 2,000 lines of Java code.

Test Generation. This part includes two components. First, a Python script containing roughly 1,000 lines of code handles interactions with the large language model for test generation. For each

function under test, the script gathers the relevant context, synthesizes the final prompts according to a test case generation template, and sends them to the LLM, specifically GPT-4o, as the backend model. The LLM returns a JSON-formatted set of test cases as intermediate results. In this process, we did not use GPT-4o's JSON mode. Our early experiments revealed that requiring GPT-4o to produce strictly formatted JSON data increases the likelihood of errors, particularly in scenarios demanding a large number of test cases. To mitigate this issue, we only ensure that each individual test case is generated in JSON format.

These test cases are then parsed, and legal C test code is generated by leveraging the metadata from the context extraction phase. This component is implemented in approximately 6,400 lines of Java code. For simplicity, the example test code shown in *Figure 3* differs from the actual implementation. The actual test code is more complex, accommodating a wider range of inputs and handling stub functions for consistent results. In practice, we provide a graphical user interface that allows users to import JSON-formatted test cases manually or automatically into the tool, where they can seamlessly generate test code.

Test Case Optimization is also carried out through a simple Python script, similar to the LLM interactions in the test generation component. Test execution and coverage analysis are fully supported by the SunwiseAUnit framework.

4 EXPERIMENT DESIGN

Our evaluation is designed to answer the four main research questions in the experiments:

RQ1: How does the correctness displayed by the test cases generated by STRUT compare to the baselines?

RQ2: How does the coverage performance of the test cases generated by STRUT compare to the baselines?

RQ3: How effective is the feedback optimization step in improving STRUT's performance?

RQ4: How does the performance of STRUT on other LLMs?

4.1 Dataset

Our dataset consists of three categories: open source datasets, real-world datasets, and manually constructed datasets, as shown in Table 1.

The selection of open-source datasets follows specific standards. First, the dataset should encompass various application domains of C programs to verify the applicability of our method. Second, the projects within the dataset should have high star ratings on GitHub, serving as representative samples of their respective domains. To ensure this, we set a selection criterion that requires each project to have received more than 500 stars. Based on these criteria, we selected six high-quality projects from GitHub, including domains such as cryptographic algorithms, data structures algorithms, string manipulation, file handling, base64 encoding, and Operating Systems.

The real-world dataset consists of software programs sourced from safety-critical industries. These datasets include: CTU02 for system message services, C6X-16ST for bus data processing, and SPARCV8-SUT for telemetry data generation. These software programs generally operate under strict reliability and security requirements. The complex functions, high performance standards, and safety-critical nature of these applications present a challenging test scenario for our method. Evaluating our approach within these real-world software allows us to rigorously test the effectiveness and practicality of our method in actual engineering contexts, where robustness is essential.

To further diversify and enrich the comprehensiveness of the test data, we have invested significant effort in manually constructing a benchmark test set called UTBench. To better reflect the characteristics of industrial software, UTBench was constructed with a focus on several key factors:

Table 1. Dataset Statistics

Projects	Domain	Trained	Focal methods	Avg.Line	Avg.Branch	Line Distribution			
						0-10	10-20	20-50	>50
bstrlib[3]	C String Library	yes	139	20.37	11.31	11	41	57	29
TheAlgorithm_C[5]	C Data Structure Algorithm	yes	287	14.32	4.50	120	123	39	6
klib[2]	Generic C Library	yes	116	16.78	10.45	38	32	26	20
crypto-algorithms[4]	Crypto Algorithm	yes	56	21.74	4.00	7	19	23	7
xv6-riscv[1]	Operating System	yes	211	17.58	7.22	56	83	62	10
file[7]	File handling	yes	307	25.88	14.79	74	72	95	66
CTU02	System Message Service	no	236	45.24	14.51	17	23	89	106
C6X-16ST	Bus Data Processing	no	120	33.05	13.55	14	27	45	34
SPARCV8-SUT	Telemetry Package Generation	no	44	22.28	4.96	11	16	8	9
UTBench	Unit test Datasets	no	193	21.63	6.16	29	88	60	10
Total			1709	39256	16144	377	525	506	301

(1) real-world programming structures like sequential, selection, and loop structures, (2) boundary conditions such as array size limits and numerical range boundaries, and (3) robustness to handle common pitfalls in industrial software, including memory management, pointer operations, and data synchronization. These factors were carefully considered to ensure that the dataset accurately represents the challenges encountered in industrial C programming. UTBench contains 193 carefully designed focal methods, which cover a broad range of practical scenarios found in industrial contexts, ensuring comprehensive evaluation and compatibility with typical industrial software, such as embedded systems, system software, and libraries.

4.2 Configuration

In our experiments, we use OpenAI's GPT-4o as the large language model with the temperature set to 0.5. We chose GPT-4o as our experimental large language model because it is one of the most advanced LLMs. Meanwhile, to validate the effectiveness of our approach, we conducted experiments on other LLMs, such as Deepseek-v3. To evaluate the efficiency of our approach to generating test cases, we attempt to generate test cases once for each focal method and perform feedback optimization once. When focal methods have a large number of branches, symbolic execution may encounter the issue of path explosion. We set the upper limit for the unit test generation time at 120 seconds. If it exceeds this limit, we abandon the generation process. Like our method, all baselines default to stubbing the called functions to ensure the isolation of unit tests. We compile and run all method test cases in SunwiseAUnit and calculate the coverage. To alleviate the impact of LLM uncertainty on the experiments, all LLM-based experiments are conducted three times, and we used the average results from these three rounds as the final outcomes.

4.3 Baselines

To evaluate the effectiveness of STRUT, we compare it against three baseline tools: SunwiseAUnit, GPT-4o-C, and ChatUniTest-C.

- **SunwiseAUnit**[6]: It is a commercial symbolic execution-based unit testing tool designed for C/C++ programs. It generates high-coverage unit tests, achieving an average branch coverage of 80%. SunwiseAUnit is widely used across various industries, particularly for testing complex systems, owing to its strong performance and reliability.
- **GPT-4o-C**: GPT-4o[8] is one of the most advanced large language models to date, capable of mimicking human reasoning and generating code based on information learned from vast textual datasets. It is the large language model used in this study. To implement the baseline method, GPT-4o-C, we first use Algorithm 1 to extract the focal method and its context,

consistent with the approach outlined in this paper. We then instruct GPT-4o to generate test cases and stub functions, using the prompts shown in Figure 8. Unlike the approach proposed in this study, GPT-4o-C does not use structured seed cases, making it suitable for an ablation study to validate the key role of structured test cases in our method.

- **ChatUniTest-C**: ChatUniTest[39] is an automated unit test generation method based on the Generation-Validation-Repair framework, originally designed for Java programs and powered by GPT-3.5-turbo. It generates unit tests by analyzing a project, extracting key information, and constructing an adaptive context that includes the main method and its dependencies. We have replicated this method and implemented a version suitable for C programs, ChatUniTest-C, and employ the GPT-4o model to maintain consistency.

```
Please help me generate a test function for a specific C function.
I will provide the source code for the function, relevant function signatures and variable, required dependencies.
Create a test that include necessary head file, compiles without errors, and achieves maximum branch and line coverage.
To ensure isolation of unit tests, if there is a function call in the function under test, you need to write a stub function to simulate the called function. You can change the parameter value or return a value to simulate the called function.
Output the function under test and the test code to ensure that all code can be compiled successfully. No explanations is needed.
The source code of the function under test is:

{{ context }}
{{ focal method }}
```

Fig. 8. Prompts Used by GPT-4o-C baseline method

4.4 Evaluation Metrics

To answer the research questions, it is necessary to evaluate the quality of the test cases generated by the STRUT and compare them with those generated by baseline methods. The tests generated by the LLM cannot be guaranteed to be executable, and increasing the proportion of executable tests is a target for LLMs-based methods. Therefore, we compared the execution correctness of LLMs-based testing methods. Generating expected outputs is another important goal in unit testing. The LLM can generate expected outputs based on the focal method, assisting in identifying potential issues within the program. We statistic and calculate the proportion of test cases successfully executed and oracle-correct test cases for each method. Then, we compare the distribution of two types of errors: compilation errors and runtime errors. The metrics employed in our study are as follows:

- **CompileError** refers to the percentage of the test case that produces errors during the compilation.
- **RuntimeError** refers to the percentage of the test case that includes error or failure during the execution.
- **Pass** refers to the percentage of the test case that is syntactically accurate, compiles and runs without errors.
- **OracleCorrect** refer to the percentage of the test case is syntactically correct, compiles and runs without errors, invokes the target method, includes assertions, and enables the target code to pass the test.
- **Line Coverage** refers to the percentage of lines of code in the tested code that are executed during unit testing. Measures the extent to which test cases cover the lines of code.

Table 2. Pass and OracleCorrect Rate Comparison

Projects	Passed			OracleCorrect		
	STRUT	ChatUniTest-C	GPT-4o-C	STRUT	ChatUniTest-C	GPT-4o-C
bstrlib	**98.62%**	81.35%	59.99%	**41.60%**	17.56%	10.10%
TheAlgorithm_C	**93.54%**	83.71%	66.28%	31.87%	**38.48%**	30.02%
klib	**94.24%**	82.52%	62.53%	**60.81%**	38.75%	31.57%
crypto-algorithms	**96.30%**	93.18%	56.35%	**71.22%**	3.69%	7.67%
xv6-riscv	**95.94%**	61.70%	37.95%	**66.92%**	40.55%	27.28%
file	**97.84%**	68.03%	27.71%	**71.37%**	26.85%	10.33%
CTU02	**94.12%**	68.09%	65.32%	**25.97%**	19.04%	8.69%
C6X-16ST	**92.78%**	55.92%	52.89%	**52.37%**	23.59%	16.37%
SPARCV8-SUT	**99.19%**	91.90%	75.45%	**70.91%**	48.18%	36.16%
UTBench	**97.72%**	77.09%	67.34%	**53.19%**	37.59%	24.66%
Avg.	**96.01%**	73.37%	53.20%	**51.83%**	29.78%	18.93%

- **Branch Coverage** refers to the percentage of all branches in the tested code that are executed during unit testing. It measures the extent to which test cases cover all possible paths in the code.

5 EVALUATION

5.1 RQ1: Correctness of Test Cases

Given that SunwiseAUnit employs symbolic execution techniques, the test cases it generates are typically derived from the program's syntax and semantics, ensuring smooth compilation and execution. In contrast, large language models, due to issues such as hallucinations, may produce test cases that deviate from the program's intended behavior or contain logical errors. Consequently, we conducted experimental evaluations only on the LLM-based approach.

Table 2 compares the proportion of executable tests for LLMs-based methods. The data in the table show that our method, STRUT, achieves the best results, with a test pass rate of 96.01% and an oracle correctness rate of 51.83%. Compared to ChatUniTest-C, STRUT improves the test pass rate by 22.64% and oracle correctness rate by 22.05%. Compared to GPT-4o-C, the improvements are 42.81% and 32.90%, respectively. In Table 3, we present the distribution of compilation errors and runtime errors for each approach. The results show that STRUT has the lowest compilation error rate and runtime error rate, at 2.63% and 1.36%, respectively. Compared to ChatUniTest-C, these rates are reduced by 3.72% and 18.55%, and compared to GPT-4o-C, they are reduced by 32.36% and 10.45%.

These experimental results demonstrate the significant improvements achieved by STRUT in test case generation. By utilizing structured test cases, STRUT greatly enhances both the test pass rate and the oracle correctness rate, while minimizing the compilation error rate and runtime error rate. STRUT ensures that the generated test cases are more robust, leading to fewer errors and higher reliability in execution. This approach eliminates the need for program repairs, resulting in more stable and accurate test code.

Answer to RQ1:
STRUT achieves a compilation error rate of 2.63%, a runtime error rate of 1.36%, a final pass rate of 96.01% and oracle correct rate of 51.83%, significantly outperforming all the baselines. This result demonstrates the effectiveness and reliability of our proposed method, ensuring a high degree of executability for the generated test cases.

Table 3. Compile and Runtime Comparison

Projects	CompileError			RuntimeError		
	STRUT	ChatUniTest-C	GPT-4o-C	STRUT	ChatUniTest-C	GPT-4o-C
bstrlib	**1.00%**	2.55%	33.46%	**0.38%**	15.90%	10.55%
TheAlgorithm_C	5.39%	**2.47%**	21.91%	**1.06%**	13.90%	11.81%
klib	3.42%	**0.41%**	27.29%	**2.34%**	16.87%	10.18%
crypto-algorithms	3.70%	**0.57%**	41.27%	**0.00%**	6.25%	2.38%
xv6-riscv	**3.83%**	13.74%	52.23%	**0.23%**	24.57%	9.81%
file	**1.77%**	11.44%	66.15%	**0.39%**	20.45%	6.14%
CTU02	**2.38%**	5.05%	18.53%	**3.50%**	26.86%	16.15%
C6X-16ST	**3.04%**	7.42%	16.91%	**4.18%**	36.65%	30.20%
SPARCV8-SUT	**0.81%**	3.64%	19.64%	**0.00%**	4.46%	4.91%
UTBench	**1.28%**	5.63%	16.22%	**1.00%**	17.28%	16.44%
Avg.	**2.63%**	6.35%	34.99%	**1.36%**	19.91%	11.81%

5.2 RQ2: Coverage Performance of Test Cases

Table 4 presents a comparison of line and branch coverage for SunwiseAUnit, STRUT, ChatUniTest-C, and GPT-4o-C across 10 different C projects. The results show that STRUT achieves the highest overall line and branch coverage, with values of 77.67% and 63.60%, respectively. Compared to GPT-4o-C, STRUT improves line coverage by 37.34% and branch coverage by 33.02%. It also outperforms ChatUniTest, with improvements of 21.76% in line coverage and 22.20% in branch coverage. When compared to the symbolic execution based tool SunwiseAUnit, STRUT surpasses it in coverage for 7 out of the 10 projects, resulting in an overall increase of 5.92% in line coverage and 6.18% in branch coverage.

Furthermore, to investigate why STRUT significantly outperforms ChatUniTest-C and GPT-4o-C in terms of coverage, we conducted an in-depth analysis of the generated test cases. As highlighted in RQ1, LLM-generated test cases contain numerous compilation and runtime errors, which hinder their execution and thus limit coverage. However, by introducing structured test cases, we address this issue, significantly improving both coverage and the reliability of the generated tests. This structured approach enables the LLM to generate more comprehensive and accurate test cases. Additionally, the interaction between structured information and the LLM enabled a deeper understanding of the program, resulting in a greater number of test cases being generated and ultimately improving coverage even further.

To explore why STRUT performs worse than SunwiseAUnit in certain projects, we reviewed the source code and test cases of CTU02 and found that the project contains numerous focal methods with complex conditional branches. LLMs typically struggle to handle these conditionally complex branches, resulting in certain branches being uncovered and preventing high coverage. Additionally, we analyzed the reasons why STRUT significantly outperforms SunwiseAUnit in the UTBench. We found that the UTBench involves many complex data types, such as arrays and structures, and symbolic execution is less effective in handling these complex data types.

Answer to RQ2:
STRUT outperforms all the baselines in terms of coverage, achieving a line coverage of 77.67% and a branch coverage of 63.60%. When testing functions with complex data structures, STRUT demonstrates greater efficiency compared to the baselines. This indicates that STRUT is highly valuable for test case generation.

Table 4. Branch and line coverage comparison of STRUT with baselines

Projects	Line Coverage				Branch Coverage			
	SunwiseAUnit	STRUT	ChatUniTest-C	GPT-4o-C	SunwiseAUnit	STRUT	ChatUniTest-C	GPT-4o-C
bstrlib	43.88%	**76.65%**	60.55%	43.08%	26.61%	**65.56%**	50.32%	35.92%
TheAlgorithm_C	79.03%	**88.35%**	72.43%	53.69%	53.69%	**78.25%**	66.24%	49.97%
klib	73.38%	**83.72%**	67.97%	45.22%	57.85%	**68.35%**	48.87%	35.87%
crypto-algorithms	98.36%	**98.85%**	93.39%	69.38%	92.86%	**95.39%**	83.11%	48.68%
xv6-riscv	**83.58%**	81.94%	49.19%	24.98%	64.89%	**71.07%**	30.43%	15.28%
file	44.18%	**58.09%**	56.28%	26.60%	35.67%	**43.70%**	34.44%	18.47%
CTU02	**85.99%**	80.11%	50.12%	43.51%	**80.89%**	68.02%	36.93%	34.62%
C6X-16ST	77.36%	**82.13%**	30.23%	32.08%	67.84%	**72.24%**	24.61%	24.82%
SPARCV8-SUT	**92.68%**	90.15%	82.18%	61.21%	**89.47%**	75.73%	59.01%	39.91%
UTBench	74.64%	**85.19%**	66.03%	56.56%	62.93%	**78.11%**	62.19%	53.00%
Avg.	71.75%	**77.67%**	55.91%	40.33%	57.42%	**63.60%**	41.40%	30.58%

5.3 RQ3: Effectiveness of Feedback Optimization

To verify the effectiveness of the feedback optimization step in the performance of STRUT, we collected experimental data without using the feedback optimization step and compared the line coverage and branch coverage with the complete process, as shown in Figures 9 and 10. It can be clearly seen from these figures that after applying the optimization step, the line coverage and branch coverage of all projects are improved, with an overall improvement of 4.70% and 6.57% respectively, which fully demonstrates the effectiveness of the optimization step. We further analyzed the newly generated test cases and found that STRUT can effectively handle focal methods with multiple branches provided that the branch conditions are not complex. However, when there are complex branch conditions in the focal method, it is still difficult for STRUT to handle it. This observation underscores the rationale behind performing repairs on the test results only once. In addition, we note that even without feedback optimization, our method still achieves better coverage than symbolic execution.

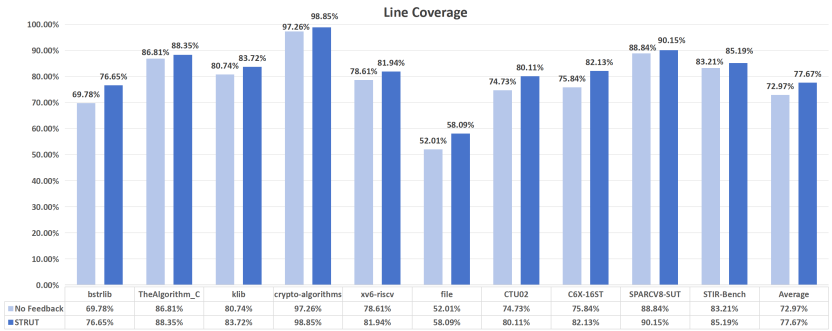

Fig. 9. Line Coverage of STRUT and STRUT Without Feedback

Answer to RQ3:
The feedback optimization step improved STRUT's line coverage by 4.70% and branch coverage by 6.57%. This further highlights the efficiency and practicality of our approach.

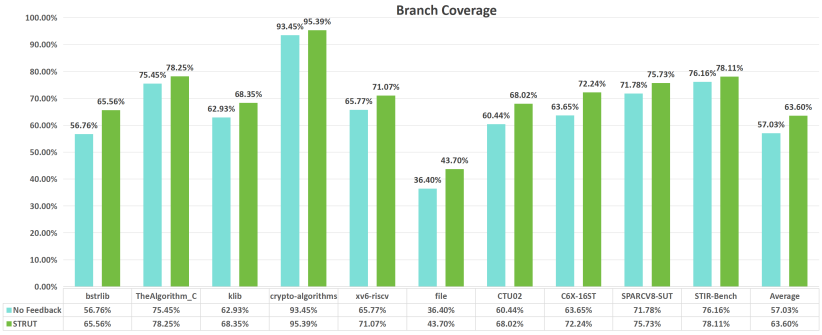

Fig. 10. Branch Coverage of STRUT and STRUT Without Feedback

Table 5. Performance of STRUT on different LLMs

Methods	Line Coverage	Branch Coverage	Comp. Error	Run. Error	Passed	OracleCorrect
GPT-4o	40.33%	30.58%	34.99%	11.81%	53.20%	18.93%
STRUT(GPT-4o)	77.67% (+37.34%)	63.60%(+33.02%)	2.63%(-32.36%)	1.36%(-10.54%)	96.01%(+42.81%)	51.83%(+32.90%)
DS-v3	22.44%	19.03%	52.56%	13.13%	34.30%	16.42%
STRUT(DS-v3)	63.60%(+41.16%)	48.97%(+29.94%)	7.66%(-44.90%)	1.18%(-11.95%)	91.14%(+56.84%)	43.94%(+27.52%)

5.4 RQ4: The performance of STRUT on other LLMs.

To validate the performance of STRUT on other LLMs, we conducted experiments using the open-source large model, DeepSeek-v3[16]. DeepSeek-V3 is a powerful Mixture of Experts (MoE) language model that employs Multi-Head Latent Attention (MLA) and the DeepSeekMoE architecture. It introduces a load-balancing strategy without auxiliary loss and sets a multi-token prediction training objective to achieve enhanced performance.

In Table 5, we compare the test performance of STRUT on GPT-4o, DeepSeek-v3, and the baseline methods. The table shows that the GPT-4o outperforms DeepSeek-v3 overall. At the same time, we can observe that STRUT significantly enhances coverage, reduces compilation and runtime errors, and improves both pass rate and accuracy on DeepSeek-v3. These results demonstrate that our method effectively boosts unit testing capabilities across different LLMs and achieve promising results.

In our evaluation, the average token cost for the prompts for each focal method is 720 tokens, and the output cost averages 1,330 tokens. By using structured test cases, our approach minimizes the overall cost when generating test cases with large language models. This efficiency is achieved through the systematic organization and targeted generation of test inputs, ensuring that the LLMs produce high-quality, relevant test cases without unnecessary computational expense. On average, GPT-4o costs $0.015 per unit test generation, while DeepSeek-v3 costs $0.0017 per test.

Answer to RQ4:
STRUT performs well across different LLMs and can significantly enhance the model's unit testing capabilities, indicating that our approach is not dependent on a specific model.

6 DISCUSSION

6.1 The Role of Seed Case Structure

The seed case of the focal method is key to STRUT's strong performance. Initially, we used a one-shot strategy with a general template to guide the LLM in generating structured test cases,

which worked well for simple programs. However, for complex data structures like nested structures and multi-dimensional arrays, the LLM struggled with accurate data representation and test logic generation. To address this, we extracted key information and designed refined seed cases to better convey complex structures. This improved the LLM's understanding, enhancing STRUT's accuracy and reliability.

In the GPT-4o-C experiment, we provided the LLM with the same context of the focal method and instructed it to generate test code instead of structured test cases. This setup effectively serves as an ablation study, where we isolate the impact of structured test case representation and feedback optimization in the STRUT method. By comparing the experimental results between STRUT without feedback optimization and GPT-4o-C, we observed a significant performance improvement of about 30 percentage points in test coverage when using STRUT.

This dramatic improvement underscores the critical role of structured seed cases in guiding LLMs to generate more effective unit tests. Without the structured representation, as seen in the GPT-4o-C experiment, the LLM's direct generation of test code lacks the systematic framework needed to ensure comprehensive coverage of the target code. The structured approach, by contrast, provides a more targeted and organized way to generate test cases, ensuring that the generated tests systematically explore a wider range of execution paths, including edge cases and corner cases that might otherwise be overlooked. Therefore, using structured seed cases efficiently guides the LLM in generating higher-quality test cases, marking a clear distinction between direct test code generation and the more systematic approach provided by STRUT.

6.2 Correlation between test case count and the coverage

In Table 6, we present the number of test cases generated for each dataset by different methods in the experiment. Overall, STRUT generated the most test cases, totaling 8,161, SunwiseAUnit generated 6,736 test cases, while ChatUniTest-C and GPT-4o-C generated 5,499 and 5,422 respectively. Compared to other LLM-based methods, our approach better guides the LLM in generating a broader range of test cases. The higher number of test cases generated by our method is a direct result of the structured approach we employ, which encourages the generation of a broader range of test scenarios, including edge cases and more comprehensive coverage of the target program's functionality. At the same time, our approach significantly enhances software test coverage while increasing the number of test cases. While the larger number of test cases generated may introduce overhead in terms of test execution and analysis, this tradeoff is mitigated by the substantial improvements in overall testing quality. The performance gains in terms of coverage and error reduction far outweigh the increase in the number of test cases, especially in environments where high reliability is paramount, such as in aerospace embedded systems.

In addition, it can be observed from the data in the table that the number of test cases generated by STRUT is 21% more than that of SunwiseAUnit. Still, the coverage is only 6% higher than that of SunwiseAUnit. Through an in-depth analysis of the test cases, it was found that some test cases in STRUT cover the same branches, while there are no such cases in SunwiseAUnit. This is because STRUT is generated based on the LLM. During the generation process, the LLM may generate some test cases with repeated branch coverage due to different understandings and processing methods of the program logic. SunwiseAUnit uses symbolic execution technology to generate test cases, and its test case generation focuses more on the precise coverage of program paths. Therefore, eliminating duplicate test cases is an important direction for our future work. We can consider designing more intelligent test case screening or merging mechanisms and better guiding the LLM to avoid generating test cases with repeated coverage during the generation process, thereby improving the quality and efficiency of test cases.

Table 6. Test Case Count Generated by Different Approaches

Projects	SunwiseAUnit	STRUT	ChatUniTest-C	GPT-4o-C
bstrlib	465	970	642	621
TheAlgorithm_C	607	1032	849	855
klib	433	544	326	327
crypto-algorithms	127	207	117	126
xv6-riscv	676	879	566	544
file	1786	1645	1166	1168
CTU02	1329	1105	627	576
C6X-16ST	594	646	355	390
SPARCV8-SUT	153	165	82	80
UTBench	566	967	770	793
Total	6736	8161	5499	5422

6.3 Threats to Validity

The threats to the validity of STRUT lies in the complexity of the testing scenarios it aims to address. C programs, particularly those involving multi-threading, external dependencies (such as databases or file systems), or intricate state management, present challenges for test generation. STRUT is designed to generate structured test cases that are intended to cover a wide range of possible execution paths and edge cases. However, in complex scenarios, particularly those involving non-trivial interactions between threads or dependencies on external resources, our method may struggle to fully capture the nuanced behavior of the program. This could lead to incomplete test coverage or failure to account for all potential runtime issues, limiting the comprehensiveness of the tests.

7 RELATED WORK

Unit testing is a software engineering activity in which individual units of code are tested in isolation[11]. Unit tests are a common practice in which developers write test cases together with regular code[15]. Using unit tests, developers can detect bugs in code during the early stages of the software development life cycle and prevent changes to the code from breaking existing functionalities, known as regression[42]. However, manually creating and maintaining unit test cases can be laborious and time-consuming[14, 24, 31]. This has prompted researchers to explore the automated generation of unit tests, with the primary goal of creating test suites that achieve high coverage of the software under test. Tools for automatically generating unit tests can be divided into two categories[36]: traditional program analysis-based and language models-based.

Within the traditional realm, researchers have explored software testing methods based on search-based software testing (SBST)[22], symbolic execution to explore possible paths and random testing. For SBST methods, EvoSuite[19] is the most widely used tool. EvoSuite randomly generates initial individuals and applies evolutionary algorithms to select, mutate, and combine test cases based on their fitness values, achieving high coverage of the software under test. However, it cannot generate meaningful test cases, leading to greater maintenance challenges. KLEE[13] is an automated test tool based on symbolic execution, capable of exploring different execution paths by analyzing a program's inputs and path conditions, thus identifying potential errors and vulnerabilities. However, KLEE struggles with complex data structures and is prone to state explosion problems. Stuart et al. [34] proposed a feedback-oriented path coverage search method, significantly improving test

coverage by refining the search strategy. CITRUS[23] is a new automated C++ unit-level testing tool to generate random method call sequences to produce a test suite achieving high test coverage. The ATheNA framework [18] is an innovative Search-Based Software Testing (SBST) framework that combines automatically and manually defined fitness functions to guide test case generation. Additionally, it introduces the concept of Instance Space Analysis (ISA). Randoop[29] utilizes the Random Testing algorithm, combined with Feedback-Directed Random Testing (FDRT), to improve the effectiveness of test generation. However, it can only generate unit tests with limited coverage, and the resulting tests are often difficult to read and maintain.

Language model-based methods, such as AthenaTest[37] and A3Test[9]. AthenaTest uses the BART [26] model as the underlying infrastructure. A3Test uses transformer architectures to learn from real-world data and generate unit tests that align with those written by developers. However, these models are unable to fix simple errors in the code, resulting in many incorrect test cases and low coverage. LLMs have demonstrated the potential for automatically generating test cases, and some efforts have explored using LLMs for unit test generation. Methods for generating unit tests based on LLMs aim to produce high-quality test cases by integrating LLMs with program analysis-based test generation, leveraging techniques such as fine-tuning [35] or prompt engineering [12, 25, 33]. ChatTester[41] investigated the capability of ChatGPT in generating unit tests, with experiments showing that LLMs outperform EvoSuite. ChatUniTest[39] introduced an adaptive context approach to address the token limitations of ChatGPT for generating Java unit tests. HITS[38] utilizes guided LLMs to first slice the program under test, then generates test cases for each slice individually, thereby improving coverage for complex functions by simplifying the focal method. Our work focuses on generating structured test cases for C programs rather than directly generating test code to enhance the execution correctness and coverage of test cases, aiming to improve the efficiency of LLM-generated unit tests and their practical application.

8 CONCLUSION

This paper introduces STRUT, a unit test generation approach that leverages structured test cases as a bridge between the complex test code and LLMs. Our approach achieved an impressive test execution pass rate and improved test coverage. We implemented STRUT and developed an industrial-grade unit test generation tool built upon a commercial tool named SunwiseAUnit. We conducted a comprehensive evaluation of STRUT. It achieved an impressive execution pass rate of 96.01%, along with 77.67% line coverage and 63.60% branch coverage, significantly outperforming GPT-4o and the commercial tool SunwiseAUnit. These results demonstrate STRUT's superior ability to produce high-quality unit test cases by harnessing the power of LLMs while overcoming their inherent flaws. We believe that the structured seed case guided approach can be extended to other programming languages.

REFERENCES

[1] 2006. "xv6-riscv". https://github.com/mit-pdos/xv6-riscv
[2] 2008. "Klib: a Generic Library in C". https://github.com/attractivechaos/klib
[3] 2014. "Better String Library". https://github.com/websnarf/bstrlib/tree/master.
[4] 2015. "crypto-algorithms". https://github.com/B-Con/crypto-algorithms
[5] 2016. "algorithm_c_data_structures". https://github.com/TheAlgorithms/C/tree/master/data_structures
[6] 2017. SunwiseAUnit. https://www.sunwiseinfo.com/sunwiseaunit
[7] 2018. "file". https://github.com/file/file
[8] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[9] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology* (2024), 107565.

[10] Luciano Baresi and Matteo Miraz. 2010. Testful: Automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 281–284.
[11] Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.
[12] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 54–61.
[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[14] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 107–118.
[15] 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.
[16] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437
[17] Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. 2022. Basic block coverage for search-based unit testing and crash reproduction. *Empirical Software Engineering* 27, 7 (2022), 192.
[18] Federico Formica, Tony Fan, and Claudio Menghi. 2023. Search-Based Software Testing Driven by Automatically Generated and Manually Defined Fitness Functions. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 40 (Dec. 2023), 37 pages. https://doi.org/10.1145/3624745
[19] 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.
[20] Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2201–2203.
[21] Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. 2024. TestART: Improving LLM-based Unit Test via Co-evolution of Automated Generation and Repair Iteration. *arXiv preprint arXiv:2408.03095* (2024).
[22] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–12.
[23] Robert Sebastian Herlim, Yunho Kim, and Moonzoo Kim. 2022. Citrus: Automated unit testing tool for real-world c++ programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 400–410.
[24] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.
[25] 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 *2023 IEEE/ACM 45th International Conference on*

Software Engineering (ICSE). IEEE, 919–931.
[26] M Lewis. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).
[27] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 212–223.
[28] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[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] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.
[31] P. Runeson. 2006. A survey of unit testing practices. *IEEE Software* 23, 4 (2006), 22–29. https://doi.org/10.1109/MS.2006.91
[32] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. 2014. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering* 41, 3 (2014), 294–313.
[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] Stuart Dereck Semujju, Han Huang, Fangqing Liu, Yi Xiang, and Zhifeng Hao. 2023. Search-Based Software Test Data Generation for Path Coverage Based on a Feedback-Directed Mechanism. *Complex System Modeling and Simulation* 3, 1 (2023), 12–31. https://doi.org/10.23919/CSMS.2022.0027
[35] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2023. Domain Adaptation for Deep Unit Test Case Generation. *arXiv e-prints* (2023), arXiv–2308.
[36] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering* (2024).
[37] 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).
[38] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. *arXiv preprint arXiv:2408.11324* (2024).
[39] 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).
[40] Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, et al. 2023. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588* (2023).
[41] 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).
[42] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.

Received 2024-10-31; accepted 2025-03-31