exLong: Generating Exceptional Behavior Tests with Large Language Models

Jiyang Zhang
The University of Texas at Austin, USA
jiyang.zhang@utexas.edu

Yu Liu
The University of Texas at Austin, USA
yuki.liu@utexas.edu

Pengyu Nie
University of Waterloo, Canada
pynie@uwaterloo.ca

Junyi Jessy Li
The University of Texas at Austin, USA
jessy@austin.utexas.edu

Milos Gligoric
The University of Texas at Austin, USA
gligoric@utexas.edu

***Abstract*—Many popular programming languages, including C#, Java, and Python, support exceptions. Exceptions are thrown during program execution if an unwanted event happens, e.g., a method is invoked with an illegal argument value. Software developers write exceptional behavior tests (EBTs) to check that their code detects unwanted events and throws appropriate exceptions. Prior research studies have shown the importance of EBTs, but those studies also highlighted that developers put most of their efforts on "happy paths", e.g., paths without unwanted events. To help developers fill the gap, we present the first framework, dubbed EXLONG, that automatically generates EBTs. EXLONG is a large language model instruction fine-tuned from CodeLlama and embeds reasoning about traces that lead to throw statements, conditional expressions that guard throw statements, and non-exceptional behavior tests that execute similar traces. We compare EXLONG with the state-of-the-art models for test generation (CAT-LM) and one of the strongest foundation models (GPT-4o), as well as with analysis-based tools for test generation (Randoop and EvoSuite). Our results show that EXLONG outperforms existing models and tools. Furthermore, we contributed several pull requests to open-source projects and 23 EBTs generated by EXLONG were already accepted.**

***Index Terms*—test generation, large language models, program analysis, exceptional behavior tests**

I. INTRODUCTION

Many popular programming languages, including C#, Java, and Python, support exceptions [14], [18], [58]. Exceptions are thrown during program execution if an unwanted event happens, e.g., a method is invoked with an illegal argument value. To throw an exception, a developer writes a *throw statement* in their code. These throw statements are commonly guarded with conditional statements (e.g., `if`), as exceptions should be thrown only under exceptional circumstances. Figure 1a shows a code snippet, in Java, that throws an `IllegalStateException` (line 17) when the `next` character, parsed from argument `request`, is identified as a special atom (line 11) but is neither an opening (line 12) nor a closing parenthesis (line 14).

Software developers write *exceptional behavior tests* (EBTs) to check that their code properly detects unwanted events and throws desired exceptions. Figure 1b shows an example EBT. An EBT, similar to a *non-exceptional behavior test* (non-EBT), first performs necessary setup of the system under test, e.g., creates objects (lines 4-5), then invokes a method under

```
1  class SearchCommandParser extends CommandParser {
2    static final char CHR_SPACE = ' ';
3    static final char CHR_CR = '\r';
4
5    public SearchTerm searchTerm(ImapRequestLineReader
         request) throws ProtocolException {
6      ...
7      char next;
8      while ((next = request.nextChar()) != '\n' &&
9       next != CHR_CR) {
10       next = request.consumeAll(CHR_SPACE);
11       if (isAtomSpecial(next)) {
12         if (next == '(') {
13           ...
14         } else if (next == ')') {
15           ...
16         } else {            target throw statement
17           throw new IllegalStateException("Unsupported atom
                 special char <" + next + ">");
18         }
19       }
20       ...
21     }
22     return handleOperators();
23   }
24 }
```

(a) Method under test: `searchTerm`.

```
1  public class SearchCommandParserTest {
2    private SearchTerm parse(String line)
3       throws ProtocolException {
4      final byte[] bytes = (line.endsWith("\n")?line:(line
          + '\n')).getBytes();
5      ByteArrayInputStream ins = new ByteArrayInputStream(
          bytes);                                method under test
6      return new SearchCommandParser().searchTerm(new
          ImapRequestLineReader(ins, null));
7    }
8
9    @Test(expected = IllegalStateException.class)
10   public void testUnsupportedAtomSpecialChar()
11     throws ProtocolException {   exceptional behavior test
12     parse("*");
13   }
14 }
```

(b) Exceptional behavior test written using JUnit 4 that covers the highlighted statement above.

Fig. 1: An EBT ('testUnsupportedAtomSpecialChar') from `greenmail-mail-test/greenmail` and the target throw statement.

test (line 6), and finally checks the expected behavior (line 9). For an EBT, the expected behavior is that an exception was thrown and the type of the exception matches the one specified by a developer.

Prior research has studied EBTs in practice [2], [7], [13], [24], [28] and observed that most projects already have some EBTs, but that the number of EBTs is not as high as the number of non-EBTs. Simply put, developers focus on "happy paths" and have limited time to test exceptional behaviors. Furthermore, through interviews and surveys [7], [28], prior studies confirmed the importance of EBTs and developers' desire to improve the testing of exceptional behaviors.

Sadly, tool support for automatically generating EBTs is limited. Most existing analysis-based test generation tools (e.g., Randoop [36], [42] and EvoSuite [11]) and learning-based test generation tools (e.g., CAT-LM [40] and TeCo [32]) have no special settings for targeting EBTs and are primarily evaluated on non-EBTs. Random test generation tools can be guided by reinforcement learning to target exceptional behaviors [1], but the generation works only on the entire codebase, and not for a specific throw statement that a develop might select. Additionally, tests produced by analysis-based tools lack readability [5], [6], [37].

We present the first framework, dubbed EXLONG, an instruction fine-tuned large language model (LLM) that automatically generates EBTs. LLMs are shown to be effective in code generation, including test generation [22], [32], [40], [56], [63]. Such strong prior provides a good foundation yet is not enough. EBTs contribute to only a very small percentage in existing codebases, i.e., they are not well-represented in LLM training data. The special conditions that trigger an EBT during execution are not included in the training phase of standard code LLMs, thus they do not perform well on the task of generating EBTs.

Using CodeLlama [43] as its base, EXLONG is fine-tuned [45], [60], [62] with a novel task instruction and fine-tuning data, designed specifically to embed the reasoning about a context that includes: (a) traces that lead to target throw statements, (b) guard expressions (i.e., conditional expressions that guard those throw statements), and (c) non-EBTs that execute similar traces. This context is used as the input to generate an EBT that triggers the target throw statement. The EBT that we already showed in Figure 1b was generated by EXLONG.

We assess the power of EXLONG using two use cases. In the first use case, which we call *developer-oriented use case*, a developer selects a method under test and a target throw statement, as well as a destination test file. EXLONG takes these inputs and automatically generates an EBT that executes the target throw statement.

In this use case, we compare EXLONG with the state-of-the-art models for test generation (CAT-LM [40]) and strongest foundation models (GPT3.5 [34] and GPT-4o [35]). We use a number of standard metrics (BLEU [39], CodeBLEU [41], edit similarity [49], [64] and exact match), as well as metrics specific to code, including percentage of compilable tests, executable tests, and executable tests that cover the target throw statement. Our results show that EXLONG generates 83.8% and 9.9% more executable EBTs than CAT-LM and GPT3.5, respectively.

In the second use case, which we call *machine-oriented use case*, a developer uses EXLONG to automatically generate EBTs for the entire codebase with the goal to cover all existing throw statements (with one EBT per statement). EXLONG takes the entire codebase as input, finds throw statements that are in public methods already covered by at least one non-EBT and generates one EBT for each of the throw statements. This use case is similar to the traditional test generation setup targeted by analysis-based generation tools.

In this use case, we compare EXLONG with popular analysis-based test generation tools: Randoop [36], [42] and EvoSuite [11]. Although tools complement each other (i.e., each tool can generate EBTs for some target throw statements that other tools cannot), our findings show that EXLONG outperforms Randoop and EvoSuite.

Additionally, we built EXLONG on GPT-4o (a state-of-the-art language model) without fine-tuning and evaluated it in developer-oriented use case. Our results show that EXLONG–GPT-4o outperforms GPT-4o by up to 16.6%. This emphasizes that our technique is generalizable to the most advanced proprietary LLMs.

Finally, we selected a subset of EBTs generated by EXLONG and created pull requests for several open-source projects. By the time of this writing, 23 tests generated by EXLONG have already been accepted by developers of those projects.

The key contributions of this paper include:

- **Task**. We define a novel task for LLMs: generating exceptional behavior tests (EBTs).
- **Model**. We designed and implemented EXLONG, an instruction fine-tuned LLM built on CodeLlama, which reasons about traces to methods that contain throw statements, guard expressions, and non-EBTs that cover similar traces.
- **Use cases**. We recognized two use cases for EXLONG: developer- and machine-oriented use cases.
- **Evaluation**. We assess the power of EXLONG in both use cases. In developer-oriented use case, we compare EXLONG with existing models for code and test generation. In machine-oriented use case, we compare EXLONG with analysis-based testing tools: Randoop and EvoSuite. We find that EXLONG outperforms existing state-of-the-art models and tools.
- **Dataset**. We developed a novel dataset for the presented task and this dataset is publicly available.

EXLONG is available on GitHub at https://github.com/EngineeringSoftware/exLong.

II. USE CASES

At a high level, EXLONG is designed to help software developers write EBTs that cover the throw statements within the given repository. We propose two use cases for EXLONG: *developer-oriented use case* (Section II-A) and *machine-oriented use case* (Section II-B). Note that in this work we do *not* consider generating EBTs that cover throw statements in the dependency libraries of the given repository, e.g., `ArithmeticException` thrown from the `java.lang.Math`, as

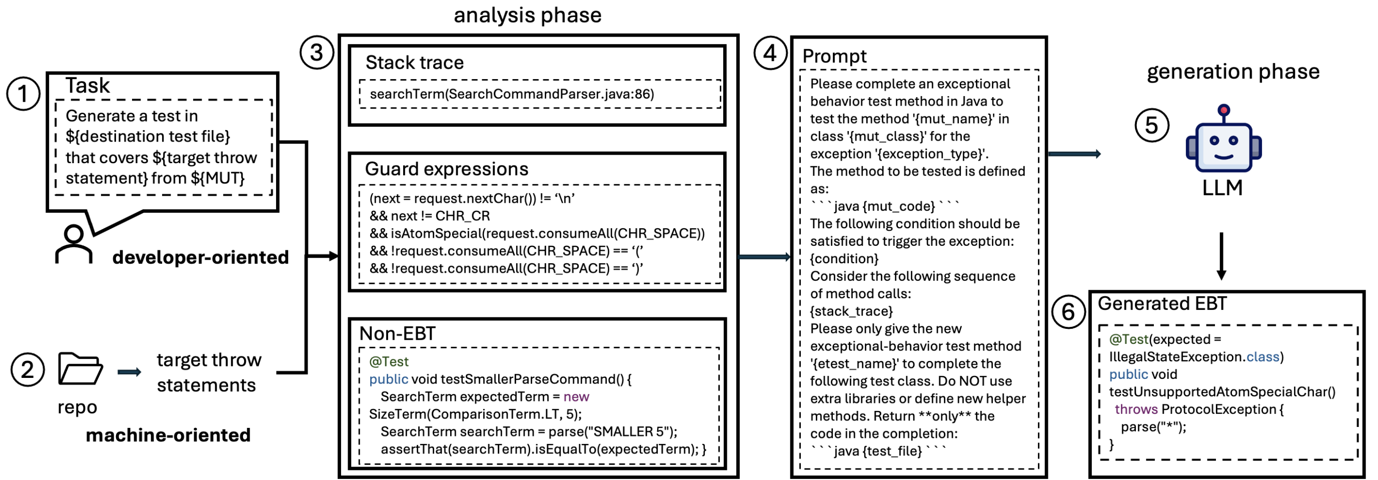


Fig. 2: Overview of EXLONG. Two use cases for EXLONG: (1) developer-oriented use case and (2) machine-oriented use case. In the developer-oriented use case, a developer specify the method under test, a target throw statement and a destination test file and ask EXLONG to generate an EBT that cover the target throw statement. In the machine-oriented use case, a developer gives an entire repository to EXLONG.

we assume that those throw statements are already covered by EBTs available in dependency libraries; developers can configure EXLONG to include such throw statements if necessary.

A. Developer-oriented use case

In the developer-oriented use case (box ① in Figure 2), a developer will specify the *method under test* (MUT), a *target throw statement*, and a *destination test file*, then ask EXLONG to generate an EBT that invokes MUT and triggers the target throw statement. When the developer specifies a target throw statement that is not within the MUT, EXLONG will first use a static program analysis technique to find all possible throw statements reachable through a sequence of method calls starting from the MUT, and then prompt the developer to select the target statement from the list.

B. Machine-oriented use case

In the machine-oriented use case (box ② in Figure 2), a developer gives an entire repository to EXLONG. EXLONG will first find all public methods and throw statements in those methods. Furthermore, EXLONG will heuristically select a destination test file. Then, EXLONG will create (public method, throw statement, destination test file) triples. Finally, it will generate one EBT to test each triple, without requiring any developer intervention. EXLONG uses static+dynamic program analyses to obtain necessary context such as traces leading to throw statements and guard expressions.

III. EXLONG

Figure 2 shows the workflow of EXLONG. Given an MUT ($\mathtt{m_{ut}}$), a target throw statement ($\mathtt{s}$), and a destination test file ($\mathtt{d}$), EXLONG reasons about the following context, collected using static and dynamic program analyses (③):

- stack trace ($\mathtt{r}$): the sequence of method invocations that start from the MUT and lead to the target throw statement.
- guard expression ($\mathtt{g}$): the logical formula representing the constraints on the symbolic variables that must be true to follow that particular trace.
- relevant non-EBTs ($\hat{\mathtt{T}}_{\mathtt{neb}}$): a subset of non-EBTs that invoke the given MUT or are in the given destination test file; their snippets and coding styles help with generating desired EBTs.

The above constitutes the prompt $\mathtt{p} = (\mathtt{m_{ut}}, \mathtt{s}, \mathtt{d}, \mathtt{r}, \mathtt{g}, \hat{\mathtt{T}}_{\mathtt{neb}})$ that encompasses the task inputs and the task context (④). During training, EXLONG is instruction fine-tuned over a base LLM to produce EBT ($\mathtt{t_{eb}}$) conditioned on the input p. During inference, EXLONG generates the EBT (⑥) given the prompt.

While the format of the LLM's input ($\mathtt{p}$) and output ($\mathtt{t_{eb}}$) is static, the steps for preparing the task inputs and reasoning about the task context differ for training and inference, and differ slightly between the developer- and machine-oriented use cases. We describe the methodology for training EXLONG in Section III-A and inference in Section III-B.

A. Training

To perform supervised fine-tuning (SFT) on EXLONG's foundation LLM, we collect a corpus ($\mathcal{C}$) of SFT data $\mathtt{c} = (\mathtt{p}, \mathtt{t_{eb}}) = (\mathtt{m_{ut}}, \mathtt{s}, \mathtt{d}, \mathtt{r}, \mathtt{g}, \hat{\mathtt{T}}_{\mathtt{neb}}, \mathtt{t_{eb}})$ from a set of repositories with developer-written EBTs and non-EBTs. Algorithm 1 shows the process of collecting the training corpus for EXLONG.

1) Identifying EBTs and non-EBTs: For each repository in the training set, we first parse the source code in the repository to identify test methods and categorize them into EBTs and non-EBTs. A test method is categorized as an EBT if it conforms to one of the four patterns that are widely used by developers [28]: Test(expected), assertThrow, expectedRule, tryFailCatch. All the other test methods are categorized as non-EBTs. The set of EBTs and non-EBTs are used as the

Algorithm 1 Collecting training corpus.

1: **inputs:** T_{eb}, T_{neb} - existing EBTs and non-EBTs
2: SUT - the system under test
3: **outputs:** $\mathcal{C}$ - the training corpus of $\{(m_{ut}, s, d, r, g, \hat{T}_{neb}, t_{eb})\}$
4: **procedure** COLLECTTRAININGCORPUS(T_{eb}, T_{neb}, SUT)
5: $\mathcal{C} \leftarrow \emptyset$
6: **for** $t_{eb} \in T_{eb}$ **do**
7: d ← GetFile(t_{eb})
8: r ← Execute(InstrumentPrintException(t_{eb}))
9: ▷ instrumenting and executing EBT to get stack trace
10: r ← ExcludeTestAndUtilMethods(r, d)
11: m_{ut} ← r[0].method ▷ MUT comes first in stack trace
12: s ← GetSourceCode(r[-1])
13: ▷ last stack trace item points to target throw statement
14: g ← ComputeGuardExp(r)
15: $\hat{T}_{neb} \leftarrow \emptyset$ ▷ initialize set of relevant non-EBTs
16: $\mathcal{C} \leftarrow \mathcal{C} \cup \{(m_{ut}, s, d, r, g, \hat{T}_{neb}, t_{eb})\}$
17: SUT ← InstrumentPrintMethod(SUT)
18: ▷ instrumentation for getting methods covered by non-EBTs
19: **for** $t_{neb} \in T_{neb}$ **do**
20: ${m_{ut}}'$ ← Execute(t_{neb})[0]
21: ▷ get MUT directly invoked by non-EBT
22: **for** $c \in \mathcal{C}$ **do**
23: **if** $c.m_{ut} == {m_{ut}}' \lor c.d ==$ GetFile(t_{neb}) **then**
24: $c.\hat{T}_{neb} \leftarrow c.\hat{T}_{neb} \cup \{t_{neb}\}$
25: **return** $\mathcal{C}$

Algorithm 2 Collect AST nodes along the stack trace.

1: **inputs:** r - the stack trace for a target throw statement
2: **outputs:** N - the collected nodes
3: **procedure** COLLECTNODES(r)
4: N ← ∅
5: **for** (m, lineno) ∈ Reversed(r) **do**
6: current ← GETSOURCECODE(m, lineno)
7: parent ← current
8: N ← N ∪ {current}
9: **while** current ≠ m **do**
10: **if** parent is ForStmt **then**
11: N ← N ∪ {parent.CompareExpression}
12: **if** parent is IfStmt **then**
13: **if** parent.ThenStmt == current **then**
14: N ← N ∪ {parent.ConditionExpression}
15: **if** parent.ElseStmt == current **then**
16: N ← N ∪ {¬ parent.ConditionExpression}
17: ▷ Other cases (while, switch, block and assignment statements) are in supplementary material.
18: current = parent
19: parent = parent.getParent()
20: **return** N

Algorithm 3 Compute guard expression based on stack trace.

1: **inputs:** r - the stack trace for a target throw statement
2: **outputs:** g - the guard expression for the target throw statement
3: **procedure** COMPUTEGUARDEXP(r)
4: N ← COLLECTNODES(r)
5: ▷ collect all condition (if, while, etc.) and assignment (assign, method call) nodes along the stack trace
6: $\mathcal{E} \leftarrow \emptyset$ ▷ the set of conditions in the guard expression
7: **for** n ∈ N **do**
8: **if** n is ConditionalExpr **then**
9: $\mathcal{E} \leftarrow \mathcal{E} \cup \{n\}$
10: **if** n is AssignStmt **then**
11: $\mathcal{E} \leftarrow$ MERGE($\mathcal{E}$, {n.lhs ↦ n.rhs})
12: **if** n is MethodDeclaration **then**
13: n′ ← n ▷ process with next node (method call)
14: **if** n is MethodCallExpr **then**
15: argmap ← ∅
16: **for** arg, argname ∈ n.getArgs(), n′.getParams() **do**
17: argmap ← argmap ∪{argname ↦ arg}
18: $\mathcal{E} \leftarrow$ MERGE($\mathcal{E}$, argmap)
19: **return** $\bigwedge \mathcal{E}$
20:
21: **procedure** MERGE($\mathcal{E}$, argmap)
22: $\mathcal{E}' \leftarrow \emptyset$
23: **for** $e \in \mathcal{E}$ **do**
24: **for** name ↦ expr ∈ argmap **do**
25: **if** e.contains(name) **then**
26: e ← e.replace(name, expr)
27: $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{e\}$
28: **return** $\mathcal{E}'$

inputs to the training corpus collection algorithm (line 1 in Algorithm 1).

2) Executing EBT and collecting stack trace: Each EBT will be expanded to one SFT example (c) in the training corpus ($\mathcal{C}$). Naturally, the file that contains the EBT is the destination test file (line 7 in Algorithm 1). To avoid data leakage problems, we remove all test methods in the test file (d) and only keep the test class structure and utility methods.

For training, stack trace is the sequence of method invocations from EBT (non inclusive) that lead to the target throw statement under test (inclusive). Line 8 in Algorithm 1 shows how to collect the stack trace: first, instrument the EBT by adding "`print(exception.getCause())`" to the code location after the exception is thrown and caught by the EBT; then, execute the instrumented EBT to get the printed stack trace. To avoid duplicate information, we exclude EBT itself and any utility methods in the destination test file from the stack trace (line 10 in Algorithm 1).

The first method invoked in the stack trace is the MUT by definition (line 11 in Algorithm 1). The last method invocation and line number in the stack trace point to the target throw statement (line 12 in Algorithm 1).

3) Computing the guard expression: Stack trace provides the sequence of method invocations that lead to the target throw statement, but knowing only the names of the methods is insufficient for generating EBTs. To aid the reasoning about the setup of the system under test, which lead to exceptional behaviors, we propose *guard expression*: a logical formula representing the constraints on the symbolic variables that must be true to follow the particular code trace. Specifically, we use conjunctions of expressions extracted from the invoked methods in the stack trace to form the guard expressions. For instance, the guard expression for the throw statement highlighted in Figure 1a is present in the Figure 2 (box ③).

The first step in the computation is to collect the list of guard-related AST nodes along the stack trace starting from the target throw statement to the MUT, as described in Algorithm 2. We traverse each method in the stack trace in

reversed order (line 5). Inside each method, we start from the AST node specified by the line number in the stack trace. We always include this node into the list of collected nodes (line 8) because the variable names in that statement may be used by the next step for replacing method call arguments. Starting from the statement, we traverse the AST by maintaining the pointer `current` that constantly moves from the child AST node to its parent node. AST nodes entailing condition expressions within the 'for' loop (lines 10 to 11), 'if' statement (lines 12 to 16), 'while' loop and 'switch' statement will be added to N. We also collect the assignment statements, method call expressions, and method declarations. Some of these are omitted from Algorithm 2 to keep it simple.

The second step of computing guard expressions is to process the collected nodes by propagating the symbolic variables through the stack trace, as described in Algorithm 3. The nodes are visited in the order being collected, and conditional expressions are directly added to the guard expression (line 9). Assignment statements, method declaration, and method call expressions will trigger a Merge operation (line 21). The goal of the Merge operation is to review the current guard expression and replace the symbolic variables (e.g., variables appearing in the target throw statement) with their corresponding values (e.g., constant values or MUT's arguments). Therefore, the guard expression reflects the MUT's arguments and public fields that are usable by EBT, rather than local variables that EBT may not have access to. For assignment statement (line 11), we replace the left hand side variable in $\mathcal{E}$ with the right hand side expression. For method declaration and method call expression (lines 15 to 18; these two nodes always appear in pairs in the collected nodes), we replace the method declaration's argument names in $\mathcal{E}$ with the actual arguments in the method call expression.

4) Connecting EBTs to relevant non-EBTs: We add the *relevant* non-EBTs to the prompt (lines 17 to 24 in Algorithm 1), to encourage the LLM to reason about the procedures to set up the object under test and the condition under which the exception will be triggered grounding the existing non-EBTs. Additionally, we believe the non-EBTs in the same repository will promote the consistency between the generated code and the existing code in terms of the both format and coding conventions. Given an MUT, we use two approaches to retrieve the relevant non-EBTs: non-EBTs that directly invoke the same MUT (used first when the context window of the LLM cannot fit all relevant non-EBTs) and non-EBTs that are already present in the destination test file (line 23 in Algorithm 1). If there is no relevant non-EBT retrieved, this part of the prompt is left empty.

5) Instruction fine-tuning: We use CodeLlama-Instruct-7B [43], an open-source foundation model designed for code generation and instruction following, as the foundation model of EXLONG. CodeLlama is pretrained on auto-regressive and infilling objectives, enabling tasks like code completion and document generation. We fine-tune CodeLlama on our collected training corpus of SFT data $\mathcal{C} = \{(\texttt{p}, \texttt{t}_{\texttt{eb}})\}$, with the novel instructions shown in Figure 2 (④). Given the instruction that includes the collected context, the model is expected to produce the EBT: $\texttt{LLM}(\texttt{p}) = \texttt{t}_{\texttt{eb}}$.

Algorithm 4 Prepare the pool stack traces from non-EBTs and assemble the prompt for evaluating EXLONG.

```
1: inputs: T_neb - existing non-EBTs
2:         SUT - the system under test
3: outputs: Q - stack traces to reach target throw statements
4: procedure COLLECTSTACKTRACESET(T_neb, SUT)
5:     SUT ← InstrumentPrintTrace(SUT)
6:     Q ← ∅
7:     for t_neb ∈ T_neb do
8:         for r ∈ Execute(t_neb) do
9:             r ← ExcludeTestAndUtilMethods(r, GetFile(t_neb))
10:            for s ∈ GetThrowStmts(r) do
11:                Q ← Q ∪ (r, t_neb, s)
12:    return Q
13:
14: global var: Q = COLLECTSTACKTRACESET(T_neb, SUT)
15: inputs: m_ut, s, d - task inputs selected by developers or inferred
16:         T_neb - existing non-EBTs
17: outputs: p - the prompt to give to the LLM
18: procedure ASSEMBLEPROMPT(m_ut, s, d)
19:    R ← ∅
20:    T̂_neb ← ∅
21:    for q ∈ Q do
22:        if m_ut ∈ q.r ∧ q.s == s then
23:            R ← R ∪ {q.r}
24:            if m_ut == q.r[0].method then
25:                T̂_neb ← T̂_neb ∪ {q.t_neb}
26:    if R ≠ ∅ then
27:        r ← RandomSelect(R)
28:        g = EXTRACTGUARDEXP(r)
29:        T̂_neb ← T̂_neb ∪ {t_neb|GetFile(t_neb)==d}
30:        return (m_ut, s, d, r, g, T̂_neb)
```

We fine-tune the CodeLlama model using the parameter-efficient Low-Rank Adaptation (LoRA) technique [20]. Rather than updating the entire set of parameters in the LLM, LoRA injects trainable low-rank matrices into each layer of the model. This approach dramatically reduces the number of trainable parameters and the amount of required computational resources.

B. Inference

The inference workflow of EXLONG is different from its training workflow in that we cannot rely on executing EBTs to collect the context (e.g., stack trace), as our goal is to generate those EBTs. Instead, EXLONG reasons about the context based on task inputs (MUT, target throw statement, and destination test file) and leveraging existing non-EBTs for the system under test. Algorithm 4 describes the key steps in preparing the inference prompt of EXLONG.

1) Collecting non-EBTs' stack traces to reach potential target throw statements: We first prepare a set of stack traces, from the execution of non-EBTs, that can reach potential target throw statements in the repository (lines 4 to 12). This only needs to be done once per repository. Specifically, we first instrument all methods that have throw statements to log the current stack trace (using

TABLE I: Statistics of the collected dataset. #MUTs is the number of unique method under test; #Exception Types is the number of unique exception types tested by the EBTs.

	#Projects	#Tests	#EBTs	#MUTs	#Exception Types
All	562	111,230	12,574	6,250	821
Train	501	100,030	11,182	5,508	725
Valid	29	5,298	550	279	66
Eval	32	5,902	842	-	-

`Thread.currentThread().getStackTrace();`) upon invoking those methods (line 5). Then, we execute all non-EBTs and collect the logged stack traces (lines 7 to 11). The execution of each non-EBT may generate multiple stack traces, as it may cover multiple methods with throw statements.

2) Selecting task inputs: Next, we select the task inputs (MUT, target throw statement, and destination test file), which can be specified by the developer or inferred by heuristics depending on the use case that EXLONG is targeting:

- Developer-oriented use case: developer specify the MUT and target throw statement to generate the EBT for, and the destination test file where the EBT should be placed.
- Machine-oriented use case: given a repository, EXLONG locates all throw statements and generates one EBT for each of them. For a target throw statement, the MUT is the method containing the throw statement, and the destination test file is selected based on (a) file name matching, and (b) test coverage analysis. Specifically, similar to prior work [40], given a code file named `FNM`, we search for test file named `FNMTest` or `TestFNM`. If there is no result based on file name matching, we run the existing non-EBTs to find any existing test class that cover the MUT or the class of MUT. If there is again no result based on test coverage analysis, EXLONG will not generate an EBT for this target throw statement (and it will move to the next one).

The selected task inputs are then used to assemble the prompt for EXLONG (line 15).

3) Assembling the prompt: We first need to find stack traces from the set of non-EBTs' stack traces that match the given MUT and target throw statement (line 22). If multiple matching stack traces are found, we randomly select one (line 27). Given the stack trace, we use the same algorithm in Section III-A3 to compute the corresponding guard expression (line 28). The relevant non-EBTs are selected using the similar criteria as Section III-A4, i.e., having the same MUT (line 25) or in the same destination test file (line 29). However, if no matching stack trace is found, EXLONG will not generate an EBT for the given inputs.

IV. DATASET

In this section, we describe details on collecting the dataset (Section IV-A), as well as the statistics of our dataset used for training and evaluation (Section IV-B).

TABLE II: Statistics of the evaluation dataset for developer-oriented use case. #MUT is the number of unique method under test; #Exception Types is the number of unique exception types tested by the EBTs; #Throw Statements is the number of unique throw statements covered by the EBTs.

	#EBTs	#MUT	#Exception Types	#Throw Statements
Developer-Oriented	434	267	41	278

TABLE III: Statistics of the evaluation dataset for machine-oriented use case. #Throw Statements is the number of target throw statements we extracted from the repository according to Section III-B2. #Exception Types is the number of unique exception types thrown by the target throw statements.

	#Throw Statements	#Exception Types
Machine-Oriented	649	81

A. Dataset Collection

Following prior work [32], we collect data from Java projects from CodeSearchNet [21], which are available on GitHub and satisfy the following: (1) use the Maven build system; (2) compile successfully; (3) do not have test failures; (4) have at least one EBT that follows one of the four patterns [28] (Section III-A1), and (5) have a license that permits the use of its data. Requirements 1-4 simplify the automation steps and ensure that we can run existing tests to collect dynamic data (e.g., stack traces), as well as run EBTs that we generate.

B. Dataset Statistics

The statistics for the collected dataset are presented in Table I. In total, we collected 111,230 tests from 562 projects, where 12,574 of these tests are EBTs. Collected EBTs cover a range of 821 unique exception types (e.g., `RuntimeException`, `IllegalArgumentException`).

The dataset is randomly split by projects into training (Train), validation (Valid), and evaluation (Eval) sets, where the training set is the SFT data used to instruction fine-tune EXLONG, the validation set is used for early-stopping the training process and guiding our design decision of EXLONG, and the evaluation set is used for evaluating the performance of EXLONG and baselines.

Table II presents the statistics of the evaluation data for developer-oriented use case. Note that this is a subset of the last row from Table I. Under developer-oriented use case, we benchmark EXLONG on the subset of 434 examples for which we are able to extract stack traces. In this paper, we focus on cases where accurate stack traces can be extracted by executing existing non-EBTs. When such non-EBTs are not available, namely the stack traces cannot be obtained, developers can first write or generate non-EBTs for the MUT with the help of other test generation tools, and then use EXLONG to generate EBTs.

Table III presents the statistics of the evaluation data for machine-oriented use case. Note that this is a subset of the last row from Table I. For machine-oriented use case, we evaluate on 649 examples as we filter the data for which we were not able to locate the destination test file with our designed heuristics (Section III-B2).

V. Evaluation Design

We assess the performance of exLong by answering the following research questions:

RQ1: How does exLong perform under the developer-oriented use case compared with the state-of-the-art models?

RQ2: How much do stack traces and guard expressions help exLong in generating EBTs?

RQ3: How much does the selection of non-EBTs help exLong in generating EBTs?

RQ4: How does exLong perform with different underlying LLM model?

RQ5: How does exLong perform under the machine-oriented use case compared with analysis-based test generation tools?

We next describe metrics used to compare models and tools (Section V-A) and then describe the baselines used in our comparison (Section V-B). We answer all research questions in Section VI.

A. Evaluation Metrics

1) Developer-oriented use case: For developer-oriented use case, we compare (using data shown in Table II) the generated EBTs against the developer-written EBTs by benchmarking on similarity-based and functional-correctness metrics.

Following prior work on learning-based test generation [32], [40], [56], we use the following similarity-based metrics to compare generated EBTs and ground-truth (i.e., developer-written ones):

Exact-match accuracy (xMatch): the percentage of the predictions that are exactly the same as the ground-truth.

BLEU [39]: the number of n-grams in the prediction that also appear in the ground-truth.

CodeBLEU [41]: adapted version of BLEU score for code. In addition to n-grams overlapping, it also computes the overlap of AST nodes, nodes in the data-flow graph between the prediction and ground-truth.

Edit similarity [49], [64]: calculates 1-Levenshtein distance which is the minimum number of character-level edits (insertions, deletions, or substitutions) required to change the prediction into the ground-truth.

The similarity metrics only capture the surface-level similarity between the prediction against an existing EBT; among them, xMatch is the most strict one as it requires perfect matches, while the others account for partial matches. However, such surface metrics do not adequately capture the functional validity of the generated EBT (e.g., whether the code can be compiled or executed), especially since the developer-written EBTs may not be the only correct implementation to cover a specific target throw statement. Thus, we additionally include the following functional-correctness metrics:

Compilable%: percentage of the generated EBTs that can be compiled. Being compilable is a basic functional requirement for the generated tests.

Matched-E%: percentage of EBTs that check the specified exception type. Namely, whether the exception class following '`@Test(expected =`' is the same as user specified one. This metric checks if the model hallucinates the exception type.

Runnable%: percentage of EBTs that check the specified exception type, and can be compiled and executed without any error. This metric, unlike others, requires the generated EBTs to be semantically valid.

ThrowCov%: out of all developer-specified target throw statements (Table II), the percentage of target throw statements with successfully generated EBTs, i.e., compilable, runnable, and checking the specified exception type. This is the strictest metric, ensuring that the generated EBTs are semantically valid and are targeting the throw statement specified by developers.

2) Machine-oriented use case: For machine-oriented use case, we benchmark tools ability to cover the throw statements within a given repository:

ThrowCov%: out of all target throw statements selected in repositories (Table III), the percentage of the target throw statements with successfully generated EBTs, i.e., compilable, runnable, and checking the correct exception type.

B. Baselines

1) Learning-based tools: We compare exLong with one of the strongest foundation models and one LLM that is specifically pretrained to generate tests.

GPT3.5: We instruct GPT3.5 [34] to write EBTs by first providing one random example from the training data. Namely, one prompt and the corresponding ground-truth EBT. The prompt we use to query GPT3.5 includes the MUT, the target exception type to test, the method containing the target throw statement, one relevant non-EBT, and the destination test file. We sample a single EBT from the output.

CAT-LM: CAT-LM [40] is an LLM pretrained on Java and Python repositories. It is pretrained with a novel objective that considers the mapping between source code and the corresponding test files. CAT-LM is pretrained to generate the remaining test methods given a code under test and the beginning of the test file. It has shown strong performance on several test generation tasks. To be consistent with its pretraining objective and intended use case, we prompt CAT-LM with the MUT followed by the destination test file, one randomly-selected relevant non-EBT, and the test annotation ('`@Test(expected =`'), encouraging the model to complete the EBT. Just like in other cases, we sample a single EBT.

2) Automatic test generation tools: In machine-oriented use case, we compare exLong with two widely-used analysis-based test generation tools.

Randoop: Randoop [36], [42] is a random test generation tool that creates tests by randomly generating inputs and recording the sequences of method calls. We run Randoop with a time

limit of 100 seconds per class to generate unit tests for each project (per the Randoop user manual [54]), we set `seed` to 42, `usethreads` to true, and other options to their default values.

EvoSuite: EvoSuite [11] is a search based test generation tool that randomly generates inputs and employs a genetic algorithm to evolve these inputs, aiming to maximize code coverage. We run EvoSuite for 120 seconds per class (as suggested in a recent SBST competition [47]). We also set the `seed` to 42. Unlike Randoop, which generates tests for the entire project, to generate more EBTs for the target throw statements within the time limit, we generate tests on a subset of classes when running EvoSuite. Starting from classes that contain throw statements, we use jdeps [53] to retrieve all classes that transitively depend on these initial classes, thereby creating a targeted subset for evaluation.

C. Hardware

We run EXLONG's program analyses part, Randoop, and EvoSuite on a machine with Intel Core i7-11700K @ 3.60GHz (8 cores, 16 threads) CPU, 64 GB RAM, Ubuntu 20.04, Java 8, and Maven 3.8.6. We perform EXLONG's LLM fine-tuning and generation, as well as CAT-LM on a server with 4 Nvidia A100 GPUs, 2 AMD Milan 7413 @ 2.65 GHz. We run fine-tuning and generation, for EXLONG and baselines, three times with different random seeds and report the average numbers across three runs.

VI. RESULTS

In this section, we present the evaluation results and answer each research question.

A. RQ1: Developer-Oriented Use Case

To answer RQ1, we compare the EBTs generated by EXLONG with developer-written tests. The results of EXLONG and baselines are shown in tables IV and V. Table IV presents the results when we inform LLMs the method name of the target EBT while in Table V we do not. (We describe the last row in these tables in a later subsection.)

EXLONG outperforms all the baselines on both similarity-based metrics (left side in tables) and functional-correctness metrics (right side in tables). EXLONG achieves higher performance than baselines for both generating executable EBTs (Runnable%) and EBTs that cover the target throw statements (ThrowCov%). This highlights that EXLONG can generate more EBTs that can be directly adopted by developers. In Table IV, we can see that EXLONG outperforms GPT3.5 by 9.9% and 22.8% on Runnable% and ThrowCov%, respectively. Similarly, we can see that EXLONG outperforms CAT-LM by 83.8% and 97.5% on Runnable% and ThrowCov%, respectively. This further underlines the benefit of the stack traces and guard expressions extracted via program analysis, and EXLONG's capability of reasoning about them.

To further understand the performance difference, we inspect the EBTs generated by GPT3.5 and EXLONG. Although GPT3.5 generates comparable number of compilable EBTs

```java
public static FileWriter createFileWriter(String className,
     LogFilePath logFilePath, CompressionCodec codec,
     SecorConfig config)
   throws Exception {
   return createFileReaderWriterFactory(className, config).
        BuildFileWriter(logFilePath, codec);
}
```

(a) The MUT to be tested.

```java
!FileReaderWriterFactory.class.isAssignableFrom(Class.
     forName(className))
```

(b) The guard expression for the target throw statement.

```java
@Test(expected = IllegalArgumentException.class)
public void testFileWriterConstructorMissing() throws
     Exception {
   ReflectionUtil.createFileWriter("MissingClass",
        mLogFilePath, null, mSecorConfig);
}
```

(c) Compilable but failing EBT generated by GPT3.5

```java
@Test(expected = IllegalArgumentException.class)
public void testFileWriterConstructorMissing() throws
     Exception {
   ReflectionUtil.createFileWriter("java.lang.String",
        mLogFilePath, null, mSecorConfig);
}
```

(d) Compilable and runnable EBT generated by EXLONG

Fig. 3: EBT (`testFileWriterConstructorMissing`) generated by GPT3.5 and EXLONG. The EBT generated by EXLONG covers the target throw statement satisfying the correct condition.

as EXLONG, it struggles to cover the correct target throw statements especially when they are not in the MUT (but could be reached through a sequence of method calls). For example, in Figure 3b, we show the guard expression extracted by EXLONG to trigger the `IllegalAccessException` with regard to the first argument (`className`) of the MUT (`createFileWriter`) in Figure 3a. The EBT generated by GPT3.5 can be compiled but fails to check the `IllegalAccessException` (Figure 3c). EXLONG uses the correct class "`java.lang.String`" that satisfies the condition and successfully covers the target throw statement (Figure 3d).

Comparing functional-correctness metrics in Table IV and Table V, performance of both GPT3.5 and CAT-LM declines significantly when the EBT method name is omitted. This result aligns with expectations, as the method name frequently implies the conditions under which the exception is supposed to be thrown, e.g., "`should_fail_if_time_provider_is_null`". In contrast, EXLONG demonstrates robustness regarding the inclusion or exclusion of the test method name, maintaining consistent performance on functional-correctness metrics. This further emphasizes the reasoning ability of EXLONG on the stack traces and guard expressions.

B. RQ2: Ablation Study of Stack Traces and Conditions

To evaluate the contribution of the components in EXLONG, we perform an ablation study. In Table VI, we show the results while including EBT's name in the prompt. We find that ablating each component deteriorates performance espe-

TABLE IV: Results on developer-oriented use case with ground-truth EBT's name in the prompt.

Models	BLEU	CodeBLEU	EditSim	xMatch	Compilable%	Matched-E%	Runnable%	ThrowCov%
GPT3.5-few-shot	56.61	64.28	82.30	14.98	75.12	**100.00**	61.29	48.39
CAT-LM	53.49	59.79	78.91	9.83	71.83	**100.00**	36.64	30.03
CodeLlama zero-shot	38.23	48.63	66.96	5.53	57.30	96.02	40.17	33.87
EXLONG	63.13	67.49	85.32	**19.05**	82.10	**100.00**	67.36	59.45
EXLONG-sample	**70.01**	**74.08**	**90.09**	15.28	**93.54**	**100.00**	**81.29**	**71.28**

TABLE V: Results on developer-oriented use case without ground-truth EBT's name in the prompt.

Models	BLEU	CodeBLEU	EditSim	xMatch	Compilable%	Matched-E%	Runnable%	ThrowCov%
GPT3.5-few-shot	38.67	49.33	69.32	0.00	82.83	**100.00**	57.58	32.32
CAT-LM	37.25	48.47	68.18	1.23	70.83	**100.00**	23.96	16.44
CodeLlama zero-shot	26.93	37.38	61.88	0.15	57.22	96.91	41.32	33.49
EXLONG	46.66	55.36	79.76	**2.07**	82.26	**100.00**	69.89	59.83
EXLONG-sample	**50.73**	**59.38**	**82.83**	1.92	**89.40**	**100.00**	**78.80**	**67.67**

cially across functional correctness metrics. Removing stack traces slightly hurts the performance of EXLONG in terms of functional correctness which is expected, because a guard expression is, in a way, the summary of a stack trace.

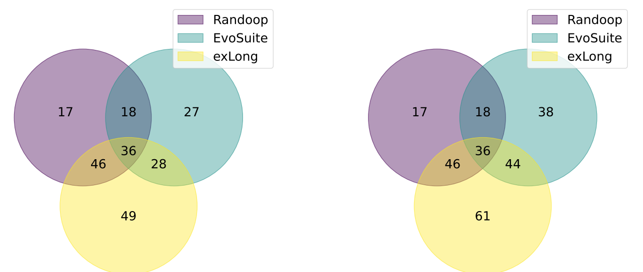
We observe a small drop in Compilable% but a rather larger drop in ThrowCov% if removing both stack trace and guard expression from the context. This underlines the importance of the condition information present by these two components on which EXLONG reasons about when generating EBTs that cover the target throw statements. Relevant non-EBTs mostly contribute to the Compilable% because the relevant non-EBT covers the same MUT which gives model a starting point to construct the EBTs. The ablation study underscores the importance of each component in EXLONG.

C. RQ3: Selection of non-EBTs

In addition to randomly choosing one relevant non-EBT to add to the prompt for EXLONG, we try running the inference of EXLONG for no more than 5 times, each time with a different relevant non-EBT, and reporting the best performance, dubbed EXLONG-sample in tables IV and V. We see that that performance differences can be substantial.

To study how the diversity of non-EBTs used in the prompt affect the EXLONG performance, our ablation study on the number of different relevant non-EBTs is shown in Table VII. We present results for 1) EXLONG generating only one EBT for each target throw statement; 2) EXLONG-sample but using the same non-EBT as 1) in multiple inference runs; 3) EXLONG-sample using different non-EBTs in multiple inference runs. For a fair comparison, we keep the number of generated EBTs for 2) and 3) same and we report the best-of-k metrics on a subset of 253 examples on which we extracted more than one relevant non-EBT.

We find that increasing the number of sampled EBTs improves the performance when compared to only generating one EBT, which can be attributed to the randomness. Moreover, sampling with a diverse set of non-EBTs further boost the performance across most of the metrics under the same sample size. This motivates a new research question on how to choose



(a) Covered throw statements on a subset of 27 projects.

(b) Covered throw statements on all 30 projects.

Fig. 4: Venn diagram that shows target throw statements coverage by EXLONG, Randoop, and EvoSuite.

better non-EBTs to the prompt for EBT generation, which we leave for future work.

D. RQ4: EXLONG Built with GPT-4o

Our technique, which assists LLMs in generating EBTs can be transferred to other LLMs with significant benefits. In addition to results with open-sourced LLMs in prior sections, we further present the results of EXLONG built on GPT-4o [35], currently one of the most powerful proprietary LLMs. Table VIII compares the performance of EXLONG–GPT-4o with baseline GPT-4o under the developer-oriented use case. The inputs to GPT-4o includes the MUT, the target exception type, the method containing the target throw statement, one relevant non-EBT, and the destination test file. We instruct EXLONG–GPT-4o with the prompt containing MUT, stack trace, guard expression, relevant non-EBTs and the destination test file. One example pair of input and ground-truth EBT is provided for both models. Results show that EXLONG–GPT-4o outperforms GPT-4o across both similarity metrics and functional-correctness metrics.

E. RQ5: Machine-Oriented Use Case

In Table IX, we present the throw statement coverage rate of EXLONG-sample and two analysis-based test generation tools. EvoSuite could generate tests for all projects, while Randoop

TABLE VI: Ablations on different context of EXLONG.

Models	BLEU	CodeBLEU	EditSim	xMatch	Compilable%	Matched-E%	Runnable%	ThrowCov%
EXLONG	**63.13**	**67.49**	85.32	**19.05**	**82.10**	**100.00**	**67.36**	**59.45**
No stack trace	62.61	67.36	**85.43**	17.74	81.41	**100.00**	67.05	58.14
No stack trace, guard expression	62.53	67.42	84.85	17.74	81.41	**100.00**	61.98	52.00
No stack trace, guard expression, non-EBT	54.84	60.62	80.70	12.06	61.52	**100.00**	46.70	38.25

TABLE VII: Comparison between using different non-EBTs to sample and using the same non-EBT but sampling multiple times. Note that we add the target EBT's name to the prompt and we only report results on examples that have more than one candidate non-EBT.

Models	BLEU	CodeBLEU	EditSim	xMatch	Compilable%	Matched-E%	Runnable%	ThrowCov%
EXLONG	61.44	66.21	85.34	13.44	83.40	**100.00**	66.14	56.79
EXLONG-sample w/ same non-EBT	63.64	68.28	86.70	13.18	87.35	**100.00**	72.46	61.53
EXLONG-sample w/ different non-EBT	**71.66**	**75.18**	**90.62**	**17.79**	**95.65**	**100.00**	**83.00**	**71.94**

TABLE VIII: Comparison between GPT-4o-few-shot and EXLONG–GPT-4o.

Models	BLEU	CodeBLEU	EditSim	xMatch	Compilable%	Matched-E%	Runnable%	ThrowCov%
GPT-4o-few-shot	60.07	65.56	84.00	16.82	81.87	99.73	71.52	55.53
EXLONG–GPT-4o	**60.48**	**66.77**	**84.77**	**17.74**	**82.49**	**100.00**	**75.35**	**64.75**

TABLE IX: Throw statements coverage rate for EXLONG, Randoop and EvoSuite.

Tools	ThrowCov%	
	Subset Projects	All Projects
EXLONG	**29.72**	**28.81**
EvoSuite	20.37	20.95
Randoop	21.87	18.95

could not generate tests for three projects. We inspected the issues and found that: (1) Randoop crashed on `OpenNMS/newts` because this project kept throwing runtime exceptions (all related to `com.codahale.metrics.ScheduledReporter`); (2) Randoop crashed on `pinterest/secor` because this project requires the configuration of Kafka; (3) Randoop could not load a class and crashed on `OpenHFT/Chronicle-Map`.

We report results on the subset of 27 projects where all the tools can be run successfully (Subset Projects) and results on all 30 projects (All Projects). Among the given target throw statements, EXLONG achieves higher throw statement coverage rate than analysis-based tools. Figure 4 illustrates the overlap and difference among the sets of target throw statements covered by EXLONG, EvoSuite, and Randoop. All three tools cover different sets of throw statements. EXLONG covers the most target throw statements that other two cannot.

VII. CASE STUDY

We performed a case study where we submitted the EBTs generated by EXLONG to the open-source projects (where the data was extracted from) to collect developers' feedback. Among the evaluation set for machine-oriented use case, EXLONG generated 187 EBTs across 30 projects that are runnable and cover the correct throw statements. We selected a subset of 9 projects that are actively maintained, i.e., they had at least one commit, accepted pull request (PR) or responded issue within the past six months (at the time of the paper submission). We found that the generated EBTs of 2 projects were the same as those added by developers on later commits (commits after the ones we used during evaluation), thus refrained from submitting PRs to them. In total, we submitted 7 PRs which include 35 EBTs (one PR per project). Among them, 4 PRs (23 EBTs) have been accepted, and 3 PRs (12 EBTs) are still pending. No PR was rejected. In one instance, a developer responded and merged our PR only 30 minutes after we create the PR. This was encouraging, and future tool development should integrate EXLONG into an IDE, such that EXLONG continuously provide EBTs for code that a developer is editing.

VIII. LIMITATIONS

We discuss several limitations and potential future work.

Programming language. In this work, we focused on supporting the Java programming language, which is among the most popular languages nowadays. We expect no substantial differences in our approach for similar programming languages, e.g., C#. Future work could evaluate and tune our model for dynamically typed languages, e.g., Python.

Project boundaries. In our evaluation, we generate EBTs for throw statements within a single project, and we ignore throw statements that are in libraries used by the project. We could not come up with a use case that targets throw statements in libraries, so we left it out of our work. If we were to target such a case, we would need to collect context (throw statement and conditions) from those libraries. One could take several directions, e.g., finding code of those libraries, building a model on bytecode level, or decompiling code and then extracting the context.

Destination test file. Not every MUT has a destination test file. We leave the problem on finding or generating destination test file for any given method under test as future work.

LLMs. We built EXLONG around CodeLlama [43], a recent open-source model. We believe that building on CodeLlama provides reproducibility guarantees that will help us and others to build on this work. Our contributions are the task, definition of a context for the task, tools for extracting the context, instruction fine-tuned model, and extensive evaluation. We expect that building EXLONG on other open-source LLMs would lead to similar results.

IX. Related Work

There has been significant work on test generation [10], [11], [36], [40], [48], [59] and code generation [23], [27], [30], [33], [38], [67], [68]. We cover related work on (1) LLM-based test generation, (2) generating tests for exceptional behavior, and (3) other test generation techniques.

LLM-based test generation. Transformer models have been used to generate tests [9], [22], [31], [32], [40], [46], [56], [59], [63] and test oracles [8], [57], [61]. CAT-LM [40] is a 2.7B model that is pretrained on a large dataset of Java and Python projects. It outperforms existing test generation tools StarCoder [23] and CodeGen 16B [33] in terms of the number of valid tests and test completion tool TeCo [31], [32]. So we compared our work with CAT-LM in this paper.

Conditions are useful for guiding the generation of tests and finding bugs [3], [4], [44]. SymPrompt [44] introduced path constraint prompting to guide LLMs to generate high-coverage tests without additional training. They collect constraints from each possible execution path in the target method and prompt the LLM to generate tests that cover those paths. We extract guard expressions by analyzing multiple methods along the stack trace starting from the throw statement to the target method for generating EBTs.

Existing test cases (including the setup and teardown methods) serve as a useful context to guide the generation [32], [40], [56]. Haji et al. [9] empirically studies the effectiveness of generating tests using GitHub Copilot and discovers that using existing test cases as context can increase the passing rate of generated tests by 37.73%. Our work uses existing non-EBTs as context and collects stack traces from those tests to guide the generation of EBTs.

Generating tests for exceptional behavior. Exception handling [28], [65], [66], [69] is an important aspect of software development. There are several techniques [1], [2], [13], [55] to generate EBTs. However, they either generate EBTs from specifications or use random-based or search-based strategies to generate EBTs. Our work is the first to use LLMs to generate EBTs. Also, prior work generates tests for the whole program, while our EXLONG allows users to specify which throw statements to cover.

Guo et al. [15] introduces boundary coverage distance (BCD) to evaluate the quality of test inputs, which can be used to guide the random generation of test inputs by minimizing BCD. Goffi [13] proposed throw statement coverage to measure the effectiveness of test inputs in triggering exceptions. We also use throw statement coverage in our evaluation.

Other test generation techniques. Other techniques incorporate random-based [36], [42], search-based [11], [17], [25], [26] and constraint-based [10], [12], [19] strategies to automatically generate tests. Tests can be derived from multiple sources, including the code under test, error messages [16], and specifications [29] like comments [50]–[52].

Randoop [36], [42] generates tests by randomly generating inputs and saving the sequences of method calls. EvoSuite [11] is a search based test generation tool that randomly generates inputs and uses a genetic algorithm to evolve the inputs to maximize code coverage. During the test generation process, both Randoop and EvoSuite create tests that cover normal as well as exceptional behaviors. However, since they generate inputs randomly, they do not guarantee to generate tests with high coverage or meaningful and readable inputs. Moreover, there is no assurance that these inputs will successfully trigger certain exceptional behaviors. EvoSuiteFIT [1] adapted EvoSuite's search algorithm with reinforcement learning to generate exceptional tests. Unfortunately, we cannot directly use EvoSuiteFIT (has error "Invalid or corrupt jarfile").

X. Conclusion

We presented the first work on generating tests for exceptional behavior (EBTs) using large language models. We introduced EXLONG that builds on top of CodeLlama to embed reasoning about traces that lead to throw statements, conditional expressions along those traces, and non-exceptional tests that cover similar traces. We evaluated EXLONG in two use cases: developer-oriented use case (i.e., generate EBT for a given method under test and a target throw statement) and machine-oriented use case (i.e., automatically generate tests for all throw statements available in a repository). Our results show that EXLONG outperforms existing test generation models and analysis-based test generation tools. We contributed a number of tests generated by EXLONG to open-source projects, and 23 EBTs are already accepted. We believe that EXLONG targets an important task, has good performance, and helps developers increase code quality assurance by automatically providing high quality EBTs.

Acknowledgments

We thank Nader Al Awar, Jayanth Srinivasa, Aditya Thimmaiah, Zijian Yi, Samuel Yuan, Zhiqiang Zang, Linghan Zhong, and the anonymous reviewers for their comments and feedback. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, CCF-2403036; as well as AST-2421782 and Simons Foundation MPS-AI-00010515 (NSF-Simons AI Institute for Cosmic Origins – CosmicAI). This work was in part supported by Cisco Research. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Cisco Research.

REFERENCES

[1] H. Almulla and G. Gay, "Learning how to search: Generating exception-triggering tests through adaptive fitness function selection," in *International Conference on Software Testing, Verification, and Validation*, 2020, pp. 63–73.

[2] R. D. Bernardo, R. Sales Jr., F. Castor, R. Coelho, N. Cacho, and S. Soares, "Agile testing of exceptional behavior," in *Brazilian Symposium on Software Engineering*, 2011, pp. 204–213.

[3] A. Blasi, A. Gorla, M. D. Ernst, and M. Pezzè, "Call me maybe: Using NLP to automatically generate unit test cases respecting temporal constraints," in *Automated Software Engineering*, 2022, pp. 1–11.

[4] I. Bouzenia and M. Pradel, "When to say what: Learning to find condition-message inconsistencies," in *International Conference on Software Engineering*, 2023, pp. 868–880.

[5] E. Daka, "Improving readability in automatic unit test generation," Ph.D. dissertation, University of Sheffield, 2018.

[6] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.

[7] F. Dalton, M. Ribeiro, G. Pinto, L. Fernandes, R. Gheyi, and B. Fonseca, "Is exceptional behavior testing an exception? An empirical assessment using Java automated tests," in *International Conference on Evaluation and Assessment in Software Engineering*, 2020, pp. 170–179.

[8] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in *International Conference on Software Engineering*, 2022, pp. 2130–2141.

[9] K. El Haji, C. Brandt, and A. Zaidman, "Using Github Copilot for test generation in Python: An empirical study," *International Workshop on Automation of Software Test*, pp. 45–55, 2024.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[11] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *International Symposium on the Foundations of Software Engineering*, 2011, pp. 416–419.

[12] P. Godefroid, "Test generation using symbolic execution," in *Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, 2012.

[13] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *International Symposium on Software Testing and Analysis*, 2016, pp. 213–224.

[14] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[15] X. Guo, H. Okamura, and T. Dohi, "Optimal test case generation for boundary value analysis," *Software Quality Journal*, pp. 1–24, 2024.

[16] X. Han, T. Yu, and D. Lo, "Perflearner: Learning from bug reports to understand and generate performance test frames," in *Automated Software Engineering*, 2018, pp. 17–28.

[17] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2009.

[18] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[19] J. Holmes, I. Ahmed, C. Brindescu, R. Gopinath, H. Zhang, and A. Groce, "Using relative lines of code to guide automated test generation for Python," *Transactions on Software Engineering and Methodology*, vol. 29, no. 4, pp. 1–38, 2020.

[20] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2022.

[21] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[22] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International Conference on Software Engineering*, 2023, pp. 919–931.

[23] R. Li, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, L. Jia, J. Chim, Q. Liu *et al.*, "StarCoder: May the source be with you!" *Transactions on Machine Learning Research*, 2023.

[24] L. P. Lima, L. S. Rocha, C. I. M. Bezerra, and M. Paixao, "Assessing exception handling testing practices in open-source libraries," *Empirical Software Engineering*, vol. 26, no. 5, 2021.

[25] Y. Liu, P. Nie, A. Guo, M. Gligoric, and O. Legunsen, "Extracting inline tests from unit tests," in *International Symposium on Software Testing and Analysis*, 2023, pp. 1–13.

[26] Y. Liu, A. Thimmaiah, O. Legunsen, and M. Gligoric, "ExLi: An inline-test generation tool for Java," in *International Symposium on Software Testing and Analysis*, 2024, pp. 1–5.

[27] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "LLaMA-Reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning," in *International Symposium on Software Reliability Engineering*, 2023, pp. 647–658.

[28] D. Marcilio and C. A. Furia, "How Java programmers test exceptional behavior," in *International Working Conference on Mining Software Repositories*, 2021, pp. 207–218.

[29] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *International Conference on Software Engineering*, 2019, pp. 188–199.

[30] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. Von Werra, and S. Longpre, "OctoPack: Instruction tuning code large language models," in *International Conference on Learning Representations*, 2023.

[31] P. Nie, "Machine learning for executable code in software testing and verification," Ph.D. dissertation, The University of Texas at Austin, 2023.

[32] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in *International Conference on Software Engineering*, 2023, pp. 2111–2123.

[33] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An open large language model for code with multi-turn program synthesis," in *International Conference on Learning Representations*, 2023.

[34] OpenAI, "GPT-3.5-turbo," https://platform.openai.com/docs/models/gpt-3-5-turbo, 2024.

[35] OpenAI, "GPT-4o," https://platform.openai.com/docs/models/gpt-4o, 2024.

[36] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[37] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: Detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, p. 170, 2022.

[38] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, "Learning to update natural language comments based on code changes," in *Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 1853–1868.

[39] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Annual Meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[40] N. Rao, K. Jain, U. Alon, C. L. Goues, and V. J. Hellendoorn, "CAT-LM: Training language models on aligned code and tests," in *Automated Software Engineering*, 2023, pp. 409–420.

[41] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBleu: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[42] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *Automated Software Engineering*, 2011, pp. 23–32.

[43] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code Llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[44] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage guided test generation in regression setting using LLM," in *International Symposium on the Foundations of Software Engineering*, 2024, pp. 951–971.

[45] V. Sanh, A. Webson, C. Raffel, S. H. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, T. L. Scao, A. Raja *et al.*, "Multitask prompted training enables zero-shot task generalization," *arXiv preprint arXiv:2110.08207*, 2021.

[46] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *Transactions on Software Engineering*, 2023.

[47] S. Schweikl, G. Fraser, and A. Arcuri, "EvoSuite at the SBST 2022 tool competition," in *International Workshop on Search-Based Software Testing*, 2022, pp. 33–34.

[48] M. L. Siddiq, J. C. Da Silva Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. Carvalho Lopes, "Using large language models to generate JUnit tests: An empirical study," in *International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 313–322.
[49] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *International Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.
[50] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments? */," in *Symposium on Operating Systems Principles*, 2007, pp. 145–158.
[51] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *International Conference on Software Engineering*, 2011, pp. 11–20.
[52] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *International Conference on Software Testing, Verification, and Validation*, 2012, pp. 260–269.
[53] J. Team, "Jdeps manual," https://docs.oracle.com/en/java/javase/11/tools/jdeps.html#GUID-A543FEBE-908A-49BF-996C-39499367ADB4, 2024.
[54] R. Team, "Randoop manual," https://randoop.github.io/randoop/manual/, 2024.
[55] N. Tracey, J. Clark, K. Mander, and J. McDermid, "Automated test-data generation for exception conditions," *Software: Practice and Experience*, vol. 30, no. 1, pp. 61–79, 2000.
[56] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.
[57] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *International Workshop on Automation of Software Test*, 2022, pp. 54–64.
[58] G. VanRossum and F. L. Drake, *The python language reference*. Python Software Foundation Amsterdam, The Netherlands, 2010, vol. 561.
[59] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *Transactions on Software Engineering*, 2024.
[60] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language models with self-generated instructions," in *Annual Meeting of the Association for Computational Linguistics*, 2023, pp. 13 484–13 508.
[61] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *International Conference on Software Engineering*, 2020, pp. 1398–1409.
[62] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.
[63] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? Evaluating and improving ChatGPT for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.
[64] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *Transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.
[65] H. Zhang, J. Luo, M. Hu, J. Yan, J. Zhang, and Z. Qiu, "Detecting exception handling bugs in C++ programs," in *International Conference on Software Engineering*, 2023, pp. 1084–1095.
[66] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," in *Automated Software Engineering*, 2021, pp. 29–41.
[67] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *International Symposium on the Foundations of Software Engineering*, 2023, pp. 695–707.
[68] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing," in *Automated Software Engineering*, 2022, pp. 1–12.
[69] H. Zhong, "Which exception shall we throw?" in *Automated Software Engineering*, 2022, pp. 1–12.