《单元测试用例自动化生成》

1024041007 殷任政

目 录

PART 01

研　究　问　题

01 单元测试

软件测试是软件开发过程中必不可少的对软件功能进行验证和检查的活动，而单元测试针对软件的单个逻辑功能单元（通常是方法和函数）进行测试，是软件测试中最基本的测试环节。

02 自动化测试

手写测试用例往往耗时又费力，自动化测试有助于提升测试效率，缩短软件交付周期，同时提高测试的质量，节约长期成本。大型语言模型（LLM）技术的发展为自动化测试提供了新的策略和方向。

03 传统测试生成方法

基于搜索：使用遗传算法等搜索技术寻找最优测试输入，如Evosuite。其优点是测试覆盖率、自动化程度高，但计算开销大、生成测试的可读性弱。

基于符号：通过符号化执行代码路径自动生成测试输入，如KLEE。具有测试覆盖率高，测试生成逻辑完备等优点，但在复杂程序中会面临路径爆炸的问题。

PART 02

《Effective Unit Test Generation for Java Null Pointer Exceptions》

--Java 空指针异常的有效单元测试生成

研究背景

在 Java 程序中，空指针异常（NPE）是最为常见、致命的错误之一。NPE 广泛出现在软件开发的实践中，对空指针进行的不当操作往往致使程序崩溃，然而现在的传统单元测试生成工具 EvoSuite 和 Randoop 虽然有时能达到较高的代码覆盖率，但并不足以有效探查出代码中的 NPE 问题。针对该现状，作者在结合传统自动化测试工具 EvoSuite 的基础上提出了一种专精于 NPE 检测的测试用例生成方法 NpeTest。

研究方法

对于每个方法 m 的目标表达式 exp 和所在行号 loc 的集合 Texp，即（exp，loc）∈Texp。若满足以下条件之一，则将其收集到集合 Texp 中：(1) 以“E1 .E2”的形式表示的表达式 E1 。(2) 调用一个很有可能触发 NPE 的方法。(3) 当返回类型为引用类型时的返回变量。

```
1   public boolean addEdge(V src, V target, E e) {
2     if (src == null) return false;
3     getEdge(src).addEdge(e);
4     if (!src.equals(target)) {
5         getEdge(target).addEdge(e);
6     }
7     return true; }
```

```
1   public Edge<V, E> getEdge(V vertex) {
2     Edge<V, E> ec = vertexMap.get(vertex);
3     if (ec == null) {
4         ec = new Edge<>(edgeFactory, vertex);
5         vertexMap.put(vertex, ec);
6     }
7     return ec; }
```

以上图 中展示的两个方法为例，能得到目标集合如下所示：

Texp (addEdge) : { (getEdge (src) ， 3), (src ， 4), (getEdge (target) ， 5) },

Texp (getEdge) : { (vertexMap, 2), (vertexMap, 5), (ec, 7) }.

以Texp (getEdge)中的(ec，7）为例，可以构建出如下指向ec的路径:

path1 →(ec，7）：[ec = new Edge< >（edgeFactory，vertex）; return ec],

path2 →(ec，7）：[!（ec == null）; return ec]

$$
\begin{aligned}
\Theta(x := \mathtt{null}, isNull) &= isNull[x \mapsto true] \\
\Theta(x := y, isNull) &= isNull[x \mapsto isNull[y]] \\
\Theta(x := new\ C(), isNull) &= isNull[x \mapsto false] \\
\Theta(x := call(\mathtt{m}'), isNull) &= isNull[x \mapsto Ret_{null}(\mathtt{m}')] \\
\Theta(x == \mathtt{null}, isNull) &= isNull[x \mapsto true], \\
\Theta(\neg(x == \mathtt{null}), isNull) &= isNull[x \mapsto false],
\end{aligned}
$$

根据上图 isNull 的映射规则，在这两条路径的入口处，ec 的 isNull 属性都将被设为 false,因此这两条指向 return ec 的路径都会导致返回非空的对象，从而 getEdge 方法永远不会返回 null，即 Ret_{null} 为 false。

对所有方法进行可空路径识别后，可以根据可空路径数量占比计算每个方法 NPE 可能性的得分，随后用于变异测试用例的选择，调用得分更高的方法的测试用例更有可能被选中进行变异。

最后，基于 EvoSuite 的测试用例执行监控，收集产生NPE的目标表达式信息，将其从Texp中移除，使生成器能够更多地关注那些尚未充分探索的与 NPE 相关的代码。

小结

实验表明，NpeTest在NPE探测优方面于传统工具EvoSuite和 Randoop 的基准水平，能够有效探测一些难以触及的空指针异常。

然而，由于其在NPE方面的针对性，NpeTest总体代码覆盖率和非NPE缺陷的探测率较低。因此，在实际应用中，应结合传统测试生成工具，充分发挥NpeTest在NPE检测中的优越性。

PART 03

《Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation》

--了解大型语言模型在定向测试输入生成上的有效性

研究背景

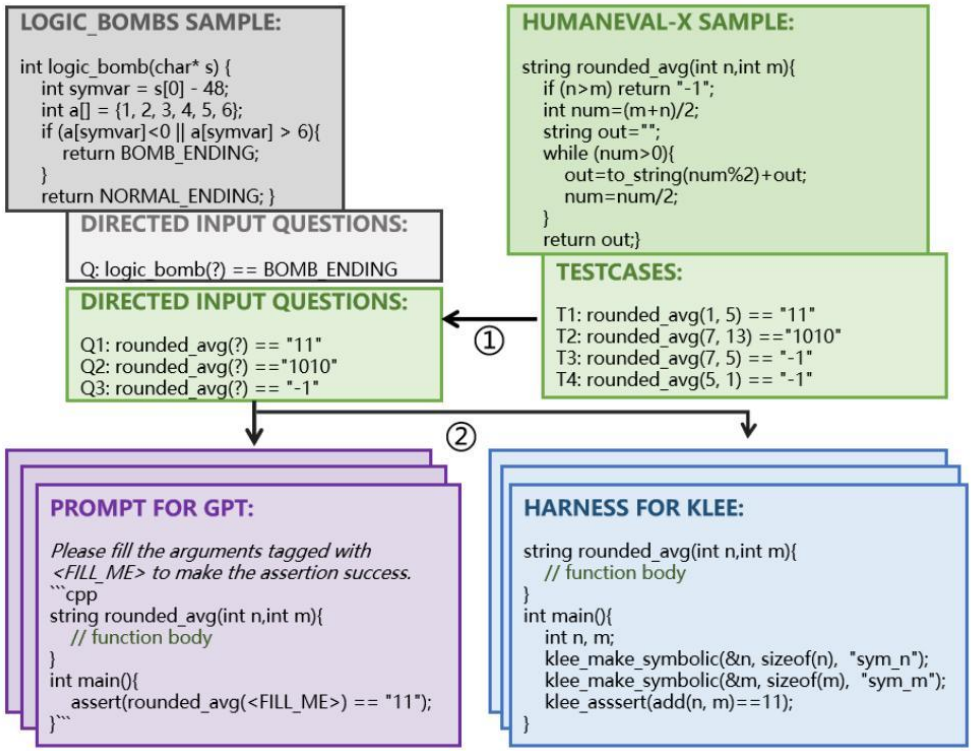
尽管传统约束工具在一定条件下确实有效，但在某些方面也有着局限性。例如随着代码复杂性增加， 会产生“路径爆炸”的问题。此外，这些工具不擅长处理与外部系统或通道交互的代码，如生成有效的 Linux 操作命令，传统约束工具无法对这些系统的约束条件进行建模。

LLM 的发展为自动化测试提供了新的方向。本文旨在将传统基于约束的工具与 LLM 在定向测试输入生成方面的有效性进行比较，以了解各自的优 缺点，并研究它们相互补充以提高性能的潜力。

研究方法

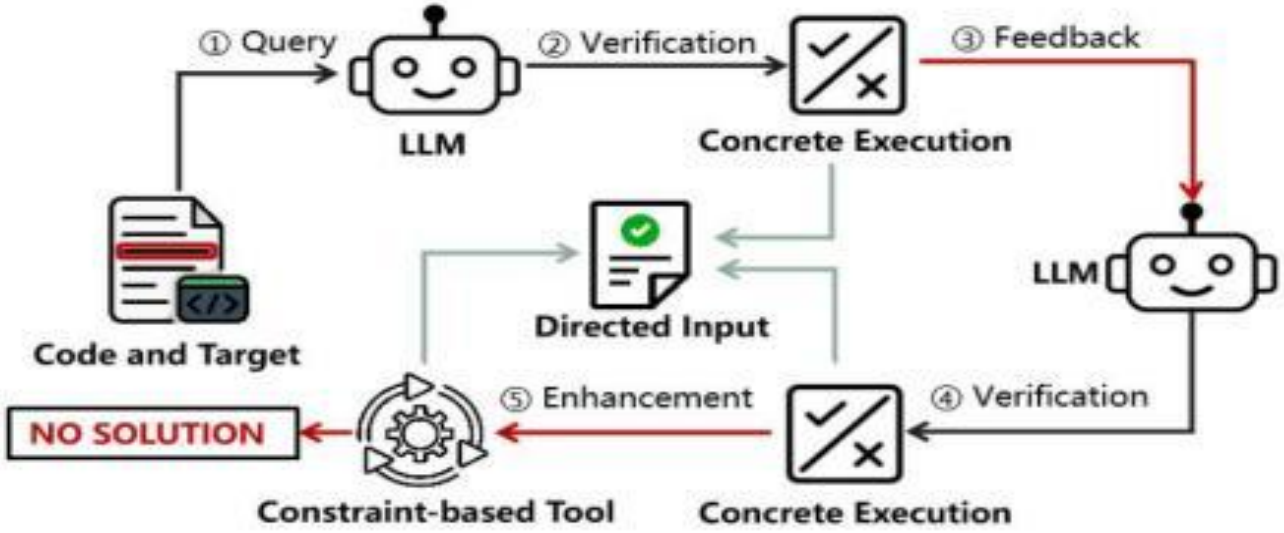
本文选择HumanEval-X作为原始数据集，包括 164 个手工编写的编程问题及其对应的单元测试，涵盖了语言理解、推理、算法、数学等各个领域。

具体地，在 HumanEval-X 的每个样本中使用测试用例来构建有向输入生成问题。首先，对于每个样本中 的每个测试用例，从相应的测试套件中提取出测试 oracle 并丢弃输入，以构建类似于 logic_bombs 数据集中样本的逆向求解问题。随后在 token 中用特殊的标记输入参数（如右图中的<FILL_ME>），生成相应的提示供LLM使用，或将它们符号化，构建一个框架供基于约束的工具解决该问题。该方法流程如右图所示。



通过上述方法获得测试样本后，使用通过率作为评估性能的标准。通过率表示工具在 K 次尝试内能够通过的数据集中样本的数量 M（即对每个样本进行 K 次尝试后成功生成能够返回目标输出的输入）与总样本数 N 的比值 M/N。

此外，本文考虑了 LLM 和传统工具在定向输入生成方面的独特优缺点，设计了一种结合 LLM 和传统工具优势的方法，称为 LLMSym，其主要工作流程如下图所示：在步骤①中，接收输入，并在步骤②中逐一通过具体执行验证候选方案。如果某个候选方案通过验证，则作为解决方案返回，过程结束。如果没有候选方案通过，则将反馈（即通过具体执行产生的 LLM 生成的输入的实际输出）返回给 LLM，在步骤③中进一步优化。然后，在步骤④中，再次通过具体执行验证优化后的解决方案。最后，如果 LLM 仍然无法生成解决方案，则将其视为 LLM 不适合解决的问题，例如精确数值计算等，从而转向传统工具。在步骤⑤中，从先前由 LLM 生成的输入中提取有用特征（如聚合数据结构大小和输入类型），为传统约束工具构建更定制化 的框架。



小结

本文研究了 LLM 在有向测试输入生成上的性能。从多个关键角度揭示差异，包括不同编程语言的影响、代码场景、输入特征以及代码的敏感性等。且通过在两个使用不同编程语言的数据集上评估 LLMSym，结果表明其性能相对于基线模型有所提升。

但是作者并没有将提示词设计、上下文等因素对LLM性能的影响纳入研究，LLM的选择也比较有限。LLMSym只是简单对LLM的生成进行验证，将无法处理的任务交由传统工具执行，并没有本质上性能的提升,且缺乏实际验证。可以在商业项目中引入该方法以进一步考察其性能。

PART 04

《On the Evaluation of Large Language Models in Unit Test Generation》

--单元测试生成中大型语言模型的评估

研究背景

LLM在单元测试生成方面的能力并未得到系统、全面的评估，本文主要就以下几点进行研究：

1、 提示词设计如何影响 LLM 在单元测试生成中的有效性。

2 、GPT-4 和开源 LLM相比EvoSuite在单元测试生成方面的表现。

3 、ICL 对基于 LLM 的单元测试生成有效性的影响。

研究方法

本文基于两种综合性能强大的开源模型 CodeLlama 和 DeepSeekCoder，选取了它们多个不同参数版本的模型进行比较。商用模型则选取了 GPT-4。

对于提示设计，作者选取了使用 LLM 处理代码任务时最常采用的两种代码描述风格：自然语言描述和代码语言描述。前者以自然语言形式描述任务和代码特征，而后者则通过在相应注释中引入任务和代码特征，使提示成为一段代码片段。

作者共收集了六个代码特征并将其分为了以下三类：

焦点方法：待测试的目标方法，从中提取的代码特征包括方法体及其参数。

焦点类：包含焦点方法的类，从中提取的代码特征包括其构造函数、类内定义的字段、类内的其他方法。

相关类：包含焦点方法参数的构造函数的类，从中提取的代码特征是这些构造函数。作者固定焦点方法的方法体，针对其余代码特征进行了消融实验，以研究它们对于性能各自的贡献。此外，本文将链式思维(CoT)和检索增强生成(RAG)这两种广泛使用的情境学习方法应用于实验并探讨了这些方法对单元测试生成效果的影响。

小结

本文对多种 LLM 进行了全面的单元测试生成效果的评估，就以下方面进行了研究：不同提示设计的影响、 开源 LLM 、商业闭源 LLM（GPT-4）和传统方法（EvoSuite）之间性能的比较 ，情境学习对基于 LLM 单元测试生成的影响，以及 LLM 生成的单元测试的缺陷检测能力。

不足之处在于作者进行了消融实验来研究不同代码特征的描述各自对LLM性能的影响，但并未考虑各种代码特征的组合和相互作用。且实验结论得出LLM因幻觉导致生成太多无效的测试用例，但并没有深入探讨能够减轻幻觉的办法。可行的解决方法是人工为LLM添加知识库，使其学习相关知识。

谢谢大家，恳请批评指正