软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007268] [CSTR: 32375.14.jos.007268]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

大模型在软件缺陷检测与修复的应用发展综述*

香佳宏[1,2], 徐霄阳[1,2], 孔繁初[1,2], 彭 湃[3], 张 钊[3], 张煜群[1,2]

[1](南方科技大学 斯发基斯可信自主系统研究院, 广东 深圳 518055)

[2](南方科技大学 计算机科学与工程系, 广东 深圳 518055)

[3](深圳艾提亚科技有限公司, 广东 深圳 518067)

通信作者: 张煜群, E-mail: zhangyq@sustech.edu.cn

摘 要: 随着信息化的深入, 大量应用程序的开发和功能迭代不可避免引入软件缺陷, 并潜在地对程序可靠性和安全性造成了严重的威胁. 检测与修复软件漏洞, 已经成为开发者维护软件质量必要的任务, 同时也是沉重的负担. 对此, 软件工程的研究者在过去的数十年中提出大量相关技术, 帮助开发者解决缺陷相关问题. 然而这些技术都面对着一些严峻的挑战, 在工业实践落地上鲜有进展. 大模型, 如代码大模型 CodeX 和对话大模型 ChatGPT, 通过在海量数据集上进行训练, 能够捕捉代码中的复杂模式和结构, 处理大量上下文信息并灵活地适应各种任务, 以其优秀的性能吸引了大量研究人员的关注. 在诸多软件工程任务中, 基于大模型的技术展现出显著的优势, 有望解决不同领域过去所面对的关键挑战. 因此, 尝试对目前已经存在基于大模型相关成熟技术的 3 个缺陷检测领域: 深度学习库的缺陷检测、GUI 自动化测试、测试用例的自动生成, 与软件缺陷修复的成熟领域: 缺陷自动化修复, 进行分析和探究, 在阐述其发展脉络的同时对不同技术流派的特性和挑战进行深入的探讨. 最后, 基于对已有研究的分析, 总结这些领域和技术所面临的关键挑战及对未来研究的启示.

关键词: 大模型; 缺陷检测; 深度学习库缺陷检测; 测试用例自动生成; GUI 自动化测试; 缺陷自动修复

中图法分类号: TP311

中文引用格式: 香佳宏, 徐霄阳, 孔繁初, 彭湃, 张钊, 张煜群. 大模型在软件缺陷检测与修复的应用发展综述. 软件学报. http://www.jos.org.cn/1000-9825/7268.htm

英文引用格式: Xiang JH, Xu XY, Kong FC, Peng P, Zhang Z, Zhang YQ. Survey on Application and Development of Large Language Models in Software Defect Detection and Repair. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7268.htm

Survey on Application and Development of Large Language Models in Software Defect Detection and Repair

XIANG Jia-Hong[1,2], XU Xiao-Yang[1,2], KONG Fan-Chu[1,2], PENG Pai[3], ZHANG Zhao[3], ZHANG Yu-Qun[1,2]

[1](Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China)

[2](Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China)

[3](ITEA Technologies Co. Ltd., Shenzhen 518067, China)

Abstract: With the advancement of informationalization, the development of a variety of applications and iterative functions inevitably leads to software defects, posing significant threats to program reliability and security. Therefore, detecting and repairing software defects becomes essential yet onerous for developers in maintaining software quality. Accordingly, software engineering researchers have proposed numerous technologies over the past decades to help developers address defect-related issues. However, these technologies face serious challenges and make little progress in industrial implementation. Large language model (LLM), such as the code-based model CodeX and the prestigious ChatGPT, trained on massive datasets, can capture complex patterns and structures in code, process extensive contextual

* 基金项目: 国家自然科学基金 (62372220)
收稿时间: 2023-12-06; 修改时间: 2024-05-18, 2024-07-09; 采用时间: 2024-07-28; jos 在线出版时间: 2025-01-08

information, and flexibly adapt to various tasks. Their superior performance has attracted considerable attention from researchers. In many software engineering tasks, technologies based on LLM show significant advantages in addressing key challenges previously faced in different domains. Consequently, this study attempts to analyze and explore three defect detection domains where technologies based on LLM have been widely adopted: deep-learning library defect detection, GUI automated testing, and automated test case generation, along with one mature software defect repair domain: automated program repair (APR). This study delves into the progress of these domains and provides an in-depth discussion of their characteristics and challenges. Lastly, based on an analysis of existing research, this study summarizes the key challenges faced by these domains and technologies and offers insights for future research.

Key words: large language model (LLM); defect detection; deep-learning library defect testing; automated test case generation; automated GUI testing; automated program repair

随着信息化进程的深入, 软件应用成为社会发展和人类生活各个领域中不可或缺的重要部分. 与此同时, 软件缺陷的问题引起了研究人员的广泛关注. 软件缺陷会影响软件的正确性、可靠性和效率, 甚至可能对用户的生命财产造成重大损失. 在核电站、医疗设备等关键基础设施中, 软件缺陷可能导致灾难性的后果. 同时, 修复软件缺陷需要消耗大量的时间和人力资源[1,2], 为软件开发者带来了巨大的压力. 因此, 软件缺陷的检测和修复任务被认为是软件工程领域的一个重大挑战.

多年以来, 研究人员已经提出并实施了各种策略和方法来检测和修复软件缺陷, 包括静态代码分析[3,4]、动态测试[5,6]、符号执行[7,8]和形式化验证[9,10]等. 这些方法在一定程度上有助于挖掘软件缺陷, 自动修复代码错误, 或者生成测试用例. 然而, 这些方法存在一些局限性. 例如, 静态代码分析可能产生大量的误报; 动态测试可能覆盖不全; 符号执行和形式化验证则需面对状态爆炸问题. 而且, 开发者往往需要专家级的知识和经验来选择合适的策略并进行有效应用. 例如, 在缺陷自动修复任务中, 基于模板的自动修复技术[11,12]需要开发人员和研究人员针对特定类型的漏洞人工制定修复模板. 虽然这样的方式可以很有效地进行修复, 但是由于人工编写修复模板成本高, 可移植性差, 限制了这一方法的应用.

近年来, 预训练大语言模型, 如 GPT 系列[13,14]、BERT 系列[15,16]模型, 以其优秀的性能吸引了大量研究人员的关注. 这些模型在自然语言处理领域取得了令人瞩目的成果, 也被越来越多地应用到软件工程领域. 大模型有几个显著的特点使其在处理软件缺陷任务上表现突出. 首先, 它们能够捕捉代码中的复杂模式和结构; 其次, 它们可以处理大量的训练数据, 并从中学习到有用的知识和模式; 最后, 它们可以灵活地适应各种任务, 如代码理解、代码生成、缺陷检测、缺陷修复和 GUI 自动测试等. 研究表明, 大模型在测试用例自动生成、缺陷自动修复等软件工程任务上, 相较于传统方法, 取得了更好的效果.

我们发现大模型在软件缺陷上的相关工作集中发表在 2023 年软件工程顶级会议上, 例如 ICSE[17]、ISSTA[18]、ASE[19]、ESEC/FSE[20], 展现出一定的趋势性. 基于此, 我们对已经发表的基于大模型的文章在缺陷任务上进行了分类, 进一步发现这些文章集中在两个方向: 缺陷检测和缺陷修复. 缺陷检测包括深度学习库的缺陷检测、GUI 自动化测试两个系统缺陷检测领域以及测试用例的自动生成领域; 缺陷修复则主要集中在缺陷自动修复领域. 目前在这 4 个方向上发表在顶级会议和期刊的基于大模型的相关工作已有 15 篇, 包括深度学习库的缺陷检测 2 篇, GUI 自动化测试 3 篇, 测试用例的自动生成 4 篇, 以及缺陷自动修复 6 篇. 许多基于大模型的研究在这 4 个领域的性能表现显著超越了以往最佳基准线, 例如, 深度学习库缺陷检测领域的 FuzzGPT[21]在 PyTorch 上的代码覆盖率较最佳基准线提高了 60.70%, 并发现了 49 个未被发现的错误; 测试用例自动生成领域的 LIBRO[22]比最佳基线多复现了 91 个错误; 而 GUI 自动化测试领域的 GPTDroid[23]在活动覆盖率上相比最佳基线提高了 32%, 并以更快的速度发现了 48 个未被发现的错误. 在修复领域, 缺陷自动修复领域的 FitRepair[24]在 Defects4J 数据集上较最佳基准线多修复了 23 个缺陷. 鉴于大模型在这 4 个领域取得的显著成果, 我们尝试对这些研究进展进行深入分析和探讨. 具体来说, 本文采用以下流程完成对相关文献的获取.

(1) 本综述选用 ACM 电子文献数据库、IEEE Xplore 电子文献数据库、Springer Link 电子文献数据库、Google 学术搜索引擎及 DBLP Computer Science Bibliography 等进行原始搜索. 对于大模型相关文献的搜索, 本综述的搜索时间从 2020 年开始, 到 2023 年结束, 并采用与 Hou 等人[25], Pan 等人[26]对于大模型的划分方法. 具体而

言, 检索的关键字包括“LLM”“large language model”“pre-trained language model”“large model”“language model”等, 以及“ChatGPT”“CodeX”等模型名称. 同时, 在标题、摘要、关键词和索引中进行检索. 我们将搜索范围限定于中国计算机学会 (CCF) 推荐软件工程和人工智能领域的顶级国际学术会议和期刊列表, 有 ICSE、ISSTA、ASE、FSE/ESEC、TOSEM、TSE、OOPSLA 等, 从而得到发布在顶级会议和期刊的, 且与软件工程任务相关的文献共 311 篇.

(2) 对筛选出来的文献进行归纳总结、分类和人工审查, 剔除与软件缺陷检测与修复无关的文献, 如代码生成. 经过此步骤, 共获得 20 篇基于大语言模型的软件缺陷研究文章. 进一步对文献内容进行细分, 发现软件缺陷检测方面的文献主要集中于深度学习库缺陷检测 (2 篇)、GUI 自动化测试 (3 篇) 以及测试用例自动生成 (4 篇) 这 3 个领域, 缺陷修复方面的文献主要集中于软件缺陷自动修复 (6 篇). 部分基于大模型的系统测试工作[27−29]由于文献数量较少, 所处领域较为分散, 因此并未包含在本综述调研的基于大语言模型的软件缺陷研究之内.

(3) 随后, 本综述继续在深度学习库缺陷检测、GUI 自动化测试、测试用例自动生成和软件缺陷自动修复这 4 个领域搜索相关文献. 我们依然将搜索范围限定于中国计算机学会 (CCF) 推荐软件工程和人工智能领域的顶级国际学术会议和期刊列表. 对于深度学习库缺陷检测相关文献, 使用了“deep learning library testing”“deep learning library bug”“deep learning API testing”“deep learning library fuzzing”等关键词, 得到顶级会议和期刊的相关文献共 159 篇. 对于 GUI 自动化测试相关文献, 使用了“GUI test”“Mobile GUI Testing”“Android APP Testing”“user interface test generation”等关键词, 得到顶级会议和期刊的相关文献共 264 篇. 对于测试用例自动生成相关文献, 使用了“unit test case generation”“unit testing”“test oracle generation”等关键词, 得到顶级会议和期刊的相关文献共 1 824 篇. 对于软件缺陷自动修复相关文献, 使用了“automated program repair”“automated fix”“patch generation”“bug fixing”等关键词, 得到顶级会议和期刊的相关文献共 1 272 篇.

(4) 之后, 根据文章长度、与目标领域的相关性、在之后工作中的被引用次数等指标对这些文献进行人工筛选. 至此, 收集了深度学习库的缺陷检测 (19 篇)、GUI 自动化测试 (54 篇)、测试用例的自动生成 (62 篇) 以及缺陷自动修复 (67 篇) 这 4 个领域共计 202 篇论文, 其中基于大模型的工作 44 篇. 图 1 和图 2 展示自 2004–2023 年本文收集论文发表时间与数量的分布情况, 其中包含会议论文 162 篇, 期刊论文 40 篇. 我们进一步发现, 2021 年后基于大模型的工作迅速增加, 仅在 2023 年便有 51 篇论文. 与此同时, 考虑到近期在软件工程领域被广泛研究应用的大模型 CodeX 和 ChatGPT 于 2021 年和 2022 年方才提出, 部分研究者可能于近日完成其工作并发表在 arXiv 上. 为追踪学界最新的研究进展, 本综述收集了提交在 arXiv 上的 45 篇工作. 我们根据 Wohlin 提出的文献搜索方法[30], 根据已搜集到文献的参考文献继续搜集文献, 以避免遗漏相关文献.

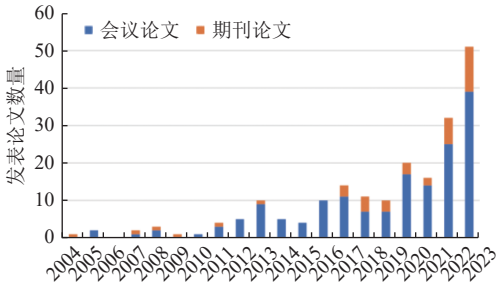


图 1 不同年份论文发表分布

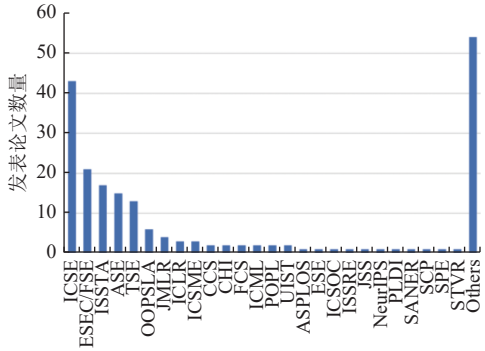


图 2 不同会议或期刊论文发表分布

事实上, 本文研究的 4 个软件工程领域历经多年, 积累了大量的工作和技术, 并且已经有综述文章对这些论文进行了翔实的调研, 例如 2021 年姜佳君等人[31]在缺陷自动修复领域的综述中收集了 94 篇高水平论文进行了详细的分析和总结. 而且本文涉及 4 个领域, 也无法将每一个领域的研究工作全部完整进行分析阐述, 因而我们将重

点关注基于大模型技术的最新进展. 此外, 关于大模型在整个软件工程领域的相关文献调研, 已有综述进行了充分的分析[25,26], 而本综述则专注于大模型在软件缺陷检测和修复领域中的应用并梳理其发展脉络. 鉴于此, 在保证本综述内容完整以及更好地反映出缺陷检测和修复 4 个领域相关研究发展脉络的前提下, 本文在介绍经典工作的同时会侧重于 2020 年后发表具有代表性的工作, 并重点聚焦于基于大模型的技术. 此外, 基于大模型的工作往往与过去的技术进行实验比较分析, 例如缺陷自动修复的技术 AlphaRepair[32]选取了大量自动修复技术进行比较. 因此, 本综述会重点选择多次被用作性能基线进行比较的技术进行分析介绍.

基于此, 本综述最终选取了 9 篇深度学习库缺陷检测文章、21 篇缺陷自动修复文章、28 篇测试用例自动生成文章、30 篇 GUI 自动测试文章, 共计 88 篇文章进行详细论述, 其中早于 2020 年的论文 34 篇, 2020–2022 年的论文 30 篇, 2023 年的论文 24 篇. 这些论文中大多数来自所涉及领域的高质量会议和期刊, 例如 ICSE 会议 (33 篇)、ESEC/FSE 会议 (12 篇)、ISSTA 会议 (8 篇)、ASE 会议 (6 篇)、OOPSLA 会议 (4 篇)、TSE 期刊 (2 篇). 综上所述, 本文的主要贡献如下.

(1) 对大模型在缺陷检测与修复方向的 4 个成熟技术领域: 深度学习库缺陷检测、GUI 自动化测试、测试用例的自动生成以及缺陷自动修复, 进行了详细的分析和探讨.

(2) 总结并分析了这 4 个领域传统技术流派的特性和挑战.

(3) 对大模型应用于这 4 个领域的机制特性进行分析阐述.

(4) 对基于大模型技术所面临的挑战进行了系统的梳理, 并总结了未来可能的研究方向.

本文第 1 节将对大模型的机制特性与常见应用于下游任务的方式进行概述. 第 2–4 节聚焦于大模型在缺陷检测领域的成熟技术与这些领域的发展脉络. 具体而言: 第 2 节在深度学习库的缺陷检测领域将首先分析介绍模型和 API 级别的工作, 进而分析最新基于大模型的技术进展. 第 3 节将介绍 GUI 自动化测试技术中传统技术的发展脉络和最新基于大模型技术的特性. 第 4 节是测试用例自动生成技术, 本文将首先回顾传统技术和挑战, 进而分析基于大模型工作的最新进展. 第 5 节将聚焦于大模型在缺陷修复方向上缺陷自动修复领域的应用, 首先分析软件缺陷自动修复机制, 并分别对传统技术、基于学习的技术和基于大模型的技术进行分析探讨. 在第 6 节, 本文将根据研究现状分析总结这些领域研究所面临的挑战及未来的机遇. 最后, 第 7 节对全文进行总结.

1 大模型概述

大型语言模型 (large language model, LLM, 简称为大模型) 通常指包含数百亿或更多参数的预训练语言模型 (pre-trained language model, PLM), 例如 GPT-3.5[33]、LLaMA[34]、PaLM[35]等. 这种模型在大量开源语料库中的文本和代码片段上进行预训练, 并且已经在各种与自然语言和代码相关的任务中展示出令人印象深刻的性能表现. 具体而言, 大模型基于流行的转换器 (Transformer) 架构[36], 该架构结合了编码器和解码器以处理文本生成任务, 其中多头注意力层堆叠在非常深的神经网络中. 编码器首先接收模型的输入, 然后生成编码向量. 解码器利用编码向量基于所有先前生成的标记自回归地生成下一个标记. 根据缩放定律 (scaling law), 模型的性能大致随着模型大小的增加而提高[37,38]. 随着研究者大幅扩展了模型的规模, 使用了更加复杂多样的预训练数据集和更加先进的训练方式, 大模型可以更好地根据上下文理解自然语言或者代码并生成更高质量的文本. 不仅如此, 相较于其他模型, 大模型的特殊性质是其具有涌现能力, 即较小规模的语言模型所不具备的处理复杂任务的通用能力. 在这种现象下, 研究人员尝试构建了越来越大的模型 (模型参数量多达 540B[35]), 并在代码补全和代码合成等代码相关任务上展示出令人印象深刻的效果.

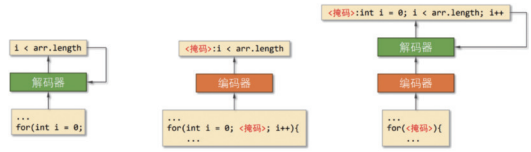
根据模型的架构和预训练目标, 如图 3 所示, 语言模型可以被划分为 3 种主要类型: 解码器模型、编码器模型 (掩蔽语言模型) 以及编码器-解码器模型. 这 3 类模型各自具有独特的架构和预训练目标, 因此在处理不同的任务和问题时, 它们的表现也会有所不同. 以下是对这 3 种模型更具体的描述.

(1) 解码器模型: 这类模型采用顺序预测的策略, 即它们生成一个标记 (例如一个词或者字符) 时, 只考虑该标记之前的上下文信息 (即前缀), 而忽略了之后的信息. GPT[13,14]是典型的解码器架构模型, 它的训练目标是根据给

定的所有前缀预测下一个标记. 这种遵从先后顺序的预测方式使这类模型特别适合于文本生成任务. 近期, 解码器模型如 CodeX[39]和 ChatGPT[40]在代码任务和文本生成任务上的优异表现, 吸引了诸多研究者的兴趣.

(2) 编码器模型: 这类模型只使用编码器组件来生成输入的编码表示. BERT[15]是典型的编码器架构模型. 它采用了一种被称为掩码语言模型 (masked language model, MLM) 的预训练机制. 在掩码语言模型中, 训练数据中的一部分标记 (例如 15%) 会被随机遮蔽, 然后模型的任务是利用该标记前后的上下文信息来预测这个被遮蔽的标记. 由于能同时考虑标记的前后上下文, 相较于单向模型, 编码器模型在理解和生成复杂句子结构方面具有一定优势.

(3) 编码器-解码器模型: 这类模型结合了上述两类模型的特点, 它们由一个编码器和一个解码器组成. 编码器用于理解输入的上下文, 而解码器则用于生成输出. T5[41]和 BART[42]是这类模型的典型代表. 这类模型通常采用一种被称为掩码范围预测 (masked span prediction, MSP) 的预训练机制. 掩码范围预测机制与掩码语言模型类似, 但该机制会使用一个范围标记替换一个连续的标记序列, 而不是遮蔽个别的标记, 而模型的任务是根据范围标记的上下文信息恢复被替换的标记序列. 编码器-解码器模型结合了上述两种模型的特点, 可以利用上下文信息进行文本生成任务.



(a) 仅解码器 (b) 仅编码器 (c) 编码器-解码器

图 3 大语言模型的 3 种架构

进而, 研究人员尝试将模型应用在下游任务中[43,44], 常见的方法是通过微调 (通过在下游任务上进一步训练来更新预训练模型的参数) 或提示 (将任务描述、示范或中间步骤以自然语言的形式表达) 增强其处理特定任务的能力. 如图 4 所示, 以大模型应用于软件缺陷自动修复任务为例: 图 4(a) 使用任务描述和修复样例对模型进行微调, 首先使模型理解和熟悉缺陷修复任务, 然后将待修复的缺陷代码输入至微调后的大模型, 即可使模型生成对应的补丁代码; 图 4(b) 将任务描述、修复样例和缺陷代码组成输入大模型的提示词, 从而使模型理解修复任务, 进而生成补丁代码. 具体而言, 微调是使用少量格式化实例对已完成预训练的模型进行受监督训练的过程. 当对模型进行微调时, 输入的指令包括任务描述和输入输出的实例. 微调能够增强各种规模的语言模型处理特定任务的能力, 经过微调的较小模型甚至可以比未经微调的较大模型表现得更好[45]. 早期将大模型应用于单元测试和软件修复等软工任务时, 微调技术被大量应用以增强模型的表现. 然而微调可能依赖数据集的质量和数量, 以及需要面对在微调数据集上过拟合的风险等问题. 在经过预训练或者微调后, 使用大模型的主要方法是为解决各种任务设计适当的提示 (prompt)[46,47]. 目前主要的提示策略有两种: 上下文学习和思维链提示[48], 具体如下.

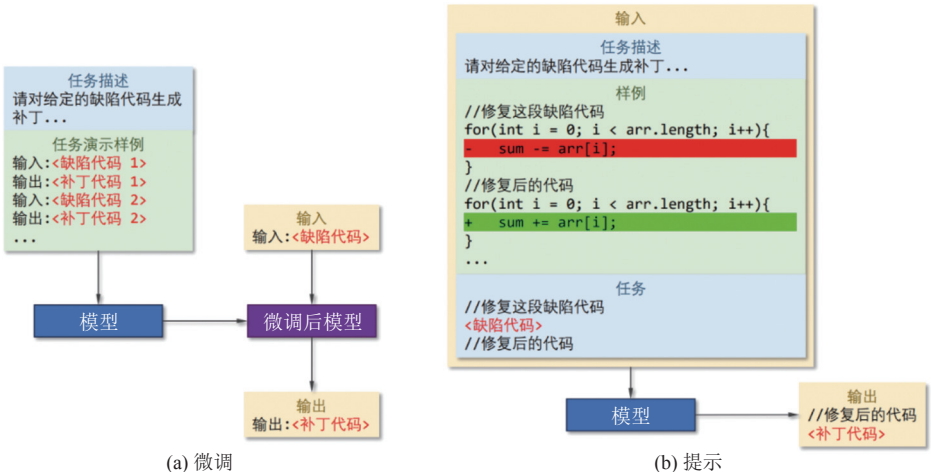
(1) 上下文学习是一种无需进一步训练就能直接将大模型用于下游任务的方法. 上下文学习使用一种由任务描述和几个用于示范的任务样例构成的自然语言提示. 首先, 以任务描述为开始, 从任务数据集中选择一些样例作为示范. 然后, 以特别设计的模板格式将它们按照特定的顺序组合成自然语言提示. 最后, 将待解决的任务添加到输入中以生成输出. 根据任务示范, 模型可以在没有更改参数权重的情况下理解和执行新任务.

(2) 思维链提示旨在提高大模型在复杂推理任务中的性能, 例如算术推理、常识推理等任务. 不同于上下文学习中仅使用输入输出对来构造提示, 思维链提示可以将中间推理步骤纳入提示中. 然而, 思维链提示主要解决推理问题, 对于不依赖推理的任务使用思维链提示可能效果不佳.

目前, 研究人员通过向模型提供任务的自然语言描述, 并尝试使用上下文学习方法提供相关任务示例, 已成功将提示策略应用于如代码补全和代码修复[49]等任务.

基于此, 我们将介绍近期大模型发展中有广泛影响和优异表现的模型, 并且这些模型已经被应用于各种软件工程下游任务上. 2021 年 8 月, OpenAI 推出了名为 CodeX[39]的代码大模型, 它是一款专门为编程任务设计的大语

言模型. CodeX 可以理解和生成代码, 处理多种编程语言, 并且适用于各种类型的编程任务, 包括代码补全、代码生成以及代码的错误修复等. CodeX 模型强大的能力引起了研究人员的注意, 已有大量的工作研究如何将 CodeX 模型应用在软工下游任务上. 在深度学习库的测试输入数据生成中, CodeX 被用于生成种子程序[50]. 在软件修复中, CodeX 在 Defects4J、QuixBugs 等数据集上均表现优异[51,52]. GitHub 还发布了基于 CodeX 的 GitHub Copilot, 它可以在编程时预测接下来要输入的完整代码片段, 并提供自然语言的描述[53,54].



(a) 微调 (b) 提示

图 4 模型应用于下游任务的方式

2022 年 11 月, OpenAI 公司推出了 ChatGPT[40], 一种基于生成预训练 Transformer 架构的大型语言模型. 开发人员在初期采取了监督学习的方式训练和调整模型, 随后则引入了一种以人类反馈为基础的强化学习方法 (reinforcement learning with human feedback, RLHF)[55]对模型进行进一步的更新和提升. ChatGPT 凭借其在处理各类任务上的卓越表现, 赢得了广泛的关注和肯定, 研究人员已经开始研究并挖掘 ChatGPT 在软件工程领域中的应用潜力, 如程序修复[56,57], 代码生成等任务. 在 2023 年 3 月, OpenAI 进一步推出最新的 GPT-4 模型, 该模型达到了目前最强的综合性能, 并支持多模态输入, 这使其在处理复杂任务以及模拟人类输入上具有相当的潜力, 比如 GUI 截图用于自动化测试[58].

2 深度学习库的缺陷检测技术

如第 1 节所述, 深度学习模型由于其强大的能力, 目前已经被广泛应用于现实生活的不同场景中, 承担着重要的作用, 例如人脸识别[59]、飞机碰撞警告系统[60]、阿尔茨海默病诊断[61]、自动驾驶汽车[62]等. 与传统软件系统相比, 构建模型基于的深度学习系统通常涉及更复杂的组件, 例如平台/硬件基础设施、深度学习库、模型、训练源程序以及训练和测试语料库, 使得深度学习系统的潜在漏洞难以被发现并且可能导致灾难性的后果. 例如, Uber 的自动驾驶车辆深度学习系统中的一个软件错误导致了一名行人的死亡[63], 特斯拉驾驶员在自主驾驶模式下遭遇车祸[64]. 出于对人身财产安全的考虑, 开发人员对深度学习库系统进行充分的测试进而保障其可靠性变得非常必要. 然而, 构建大语言模型的基础设施, 如深度学习框架 PyTorch 和 TensorFlow, 由于其复杂的架构和输入输出特征, 难以被传统软件工程的常规方法, 例如随机变异、约束求解等技术深入有效地测试. 为此, 本节首先简要介绍深度学习库测试技术的相关概念, 然后介绍不同技术的代表工作和其特性, 以及这些工作如何解决深度学习库测试的关键挑战. 在此基础上, 我们分析大模型的机制特性, 并介绍最新技术如何将其结合到深度学习库的测试任务中.

2.1 深度学习库相关概念简介

在本节中, 我们将以 PyTorch 为例对深度学习库测试技术的概念进行简要介绍. 如图 5 所示, 左侧为深度学习

模型的定义、训练以及推理过程, 右侧为深度学习库的从前端到后端再到硬件的概览. 在下文, 我们将首先介绍深度学习模型, 然后介绍模型的训练和推理过程, 最后介绍深度学习库的架构.

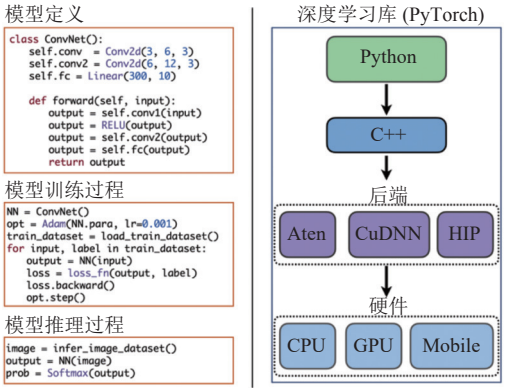


图 5 深度学习模型与深度学习库概览

深度学习模型是一种基于人工神经网络的机器学习模型, 旨在模拟人类的学习和推理过程. 如图 5 左侧模型定义相关代码所示, 该深度学习模型是一个图像分类模型, 由两个卷积层 (Conv2d) 和一个全连接线性层 (Linear) 组成. 在前向传播中, 第 1 个卷积层使用非线性激活函数 (ReLU) 产生中间输出, 然后将其传递给第 2 个卷积层, 最后调用全连接层在 10 个维度输出图片分类结果.

模型训练是通过迭代数据集、计算损失函数以及执行反向传播来更新模型权重的过程. 如图 5 左侧模型训练过程代码所示, 对于图像分类任务而言, 开发者通过数据集迭代训练, 定义损失函数来计算网络输出与其期望输出之间的差异, 并通过反向传播过程来更新模型中的权重, 从而获得训练好的模型. 与训练阶段不同, 模型推理过程是使用训练好的模型利用其学到的特征和模式来预测新数据的类别, 而无需更改模型参数权重.

深度学习库是深度学习模型训练和推理的基础设施. 如图 5 右侧所示, 深度学习库 (例如 PyTorch 和 TensorFlow) 通常为不同的硬件提供统一的抽象, 开发者在不同的环境下可以轻松配置并执行. 例如在 PyTorch 中, Aten 是一个用 C++实现的后端, 被用于执行多种张量操作, 并且为 CPU 和 GPU 等硬件进行了特殊优化. CuDNN 是另一个 PyTorch 的后端, 专为在 NVIDIA GPU 上执行深度学习任务而设计. 此外, PyTorch 还支持在移动设备上运行深度学习模型.

总的来说, 在实际中开发人员首先定义深度学习模型的层和参数等结构, 然后将模型在数据集上进行训练, 并希望模型在面对新的输入时可以根据先前学习到的知识给出正确的回答. 这些过程首先调用深度学习框架的 API (通常是 Python 语言实现), 然后 API 调用不同的深度学习库 (通常是由 C 或 C++语言实现), 最后在硬件 (例如 GPU) 上进行计算. 因此测试深度学习库要通过调用外层的深度学习框架 API 完成. 本节所介绍的深度学习库缺陷检测技术的主要测试目标是深度学习库的不同后端, 如 Aten 和 CuDNN. 这些技术利用差分测试方法, 通过发现不同后端输出结果的不一致, 以检测深度学习库的缺陷.

2.2 传统深度学习库测试技术

传统的深度学习库测试技术主要分为两类: 模型级别测试和 API 级别测试. 模型级别测试技术直接使用深度学习模型作为测试输入. API 级别测试技术针对特定的深度学习库 API 自动化或半自动化地生成测试输入, 以期找到深度学习库的异常行为. 下文我们详细介绍这两类技术流派的代表工作及关键挑战.

2.2.1 模型级别测试

CRADLE[65]是第 1 个将差分测试应用于深度学习库测试的技术, 其核心思想是在不同的后端上执行相同的深度学习模型, 以检测不同深度学习库输出结果的不一致, 从而发现深度学习库的缺陷. 在模型级别的测试中, 输出结果的不一致可能是由多个 API 运算累积的浮点精度损失引起的. 为此, CRADLE 设计了一套度量标准来区分差

分测试的错误和误报. 然而, 由于有限的 30 个测试输入模型, CRADLE 仅覆盖了 TensorFlow 中的 59 个深度学习库 API.

LEMON[66]技术在 CRADLE[65]基础上进一步提出应用模型级别的变异规则对深度学习库进行更深入的测试. 然而, 由于 LEMON 模型级别的变异规则受到深度学习库 API 参数以及输入输出形状的严格限制 (如 LEMON 论文中所述, 变异规则中的一个明确限制是插入的 API 的输出形状和输入形状必须相同), 在实际中仅有一小部分具有固定参数的 API 可以满足规则限制用于模型级别变异. 这在很大程度上影响了 LEMON 挖掘深度学习库缺陷的能力. Wei 等人[67]后续的研究表明, LEMON 的各种变异规则只能额外覆盖 5 个深度学习库 API. 2020 年, Guo 等人[68]提出了测试技术 AUDEE, 其思路和 LEMON 接近, 同样通过添加模型级别的变异规则尝试深入探索深度学习库的漏洞. 与 LEMON 变异规则不同的是, AUDEE 专注于变异层、权重张量和输入张量的参数, 但是这样的变异规则也受到严格的限制, 实际的效果较为有限.

为进一步解决深度学习库模型级别变异的约束问题, Muffin[69]尝试手动注释 API 约束. 此外, Muffin 使用预定义的代码结构来生成多样化的模型, 以保持测试输入模型的有效性. 为更准确地进行差分测试, Muffin 将模型训练阶段划分为 3 部分 (即前向计算, 损失计算和梯度计算), 并相应地设计一套在数据跟踪上的度量, 以衡量不同深度学习库的结果的一致性. 然而 Muffin 需要手动注释 API 约束, 导致其应用成本高昂, 并且该技术在使用大型深度学习模型作为输入时存在运行效率低, 随机误报漏洞等缺点, 限制了该技术的应用场景.

2.2.2 API 级别测试

与先前的深度学习库模型级别测试技术不同, API 级别测试技术并不使用模型作为测试输入, 而是将调用深度学习库 API 的片段代码作为测试输入, 对深度学习库单个 API 进行更加精细的测试. 但是 API 级别测试技术也同样面临深度学习库 API 的参数复杂和输入输出形状约束的挑战. 为解决这个问题, FreeFuzz[67]从文档、开发人员测试代码和 202 个深度学习模型的代码片段中动态跟踪 API 执行信息, 进而构建每个 API 的值空间, 再通过对 API 值空间的挖掘构建合法测试输入以进行模糊测试. 实验结果表明, FreeFuzz 可以覆盖 PyTorch 和 TensorFlow 中 1 158 个 API.

DocTer[70]与 FreeFuzz 类似, 同样将注意力集中在单个 API 的测试以及约束信息提取中. 与 FreeFuzz 不同的是, DocTer 更多地利用了深度学习库的文档信息, 通过人工构建提取规则, 从深度学习库 API 文档中提取 API 特定的输入约束, 并使用这些约束构建用于测试的输入. 然而, 在应用中 DocTer 需要对 30% 的深度学习库 API 进行手动注释, 高昂的人工成本限制了 DocTer 技术进一步的发展和应用.

在 FreeFuzz 的基础上, Deng 等人进一步发现大量深度学习库 API 共享相似的输入输出形状和约束. 因此, 通过合理调用 API 的输出作为其他 API 的输入, 即可测试其他关联的 API. 基于这样的思路, 研究者实现了可以自动化 API 输入关系推断的深度学习库模糊测试技术 DeepREL[71]. 实验结果表明, 在 PyTorch 和 TensorFlow 两个深度学习库上, DeepREL 可以覆盖 2 973 个 API, 达到了 API 级别测试的最优效果.

2.2.3 传统技术所面对的关键挑战

尽管目前的模型级别和 API 级别的深度学习库测试技术经过了多轮迭代优化和改进, 在变异规则和 API 约束推断上有显著的进步, 深度学习库 API 约束的复杂性仍然是这两类工具面对的主要挑战.

具体来说, 深度学习库 API 是基于动态语言 Python 实现暴露给用户的, 这使得直接获取 API 的输入和输出参数类型非常困难. 此外, 深度学习库 API 大量使用张量 (多维数组) 作为输入, 如果形状不匹配 (例如, 维度不匹配的矩阵乘法) 可能导致运行时错误. 传统的程序合成技术通常局限于程序语言中的部分功能和特性, 并且无法处理具有特定输入输出参数的深度学习库 API. 因此, 目前部分深度学习库测试技术使用预定义的语法规则, 专注于突变部分程序尝试挖掘漏洞. 然而, 这限制了测试输入中代码结构和输入类型的多样性. 例如, FreeFuzz 只能通过挖掘开源代码片段来收集目标 API 的有效参数空间 (例如, 输入张量的类型和形状) 对初始种子进行细微的突变以生成新的测试输入. 这样的方式导致现有的测试技术无法有效探索 API 输入空间以及挖掘潜在的漏洞.

此外, 虽然 API 级别的测试技术在覆盖率和发现漏洞的数量方面显著优于模型级别的测试. 但这些技术所面对的挑战在于, 它们通常仅利用简单突变的代码片段 (往往仅为单行代码) 作为测试输入, 因而限制了它们在检测

由复杂 API 序列触发的漏洞方面的能力. 虽然模型级别测试技术可以潜在地测试深度学习库 API 序列, 但是, 变异规则通常有严格的约束, 导致成本高昂且测试效果有限. 例如, LEMON 的层添加规则不能应用于具有不同输入和输出形状的层, 而 Muffin 需要手动注释 API 的输入和输出约束, 并使用额外的重塑操作来确保层间有效的连接. 因此, 模型级别和 API 级别的深度学习库测试技术只能覆盖有限的 API 范围, 无法生成 API 序列进而挖掘复杂的深度学习库漏洞.

因此, 目前模型级别和 API 级别的测试技术都面对着人工成本和修复性能平衡的关键挑战. 如果使用简单的, 完全自动化的技术, 那深入探索深度学习库的程序状态就非常困难. 例如 LEMON 和 AUDEE 技术尽管使用了经过细致调整的变异规则, 其效果仍十分有限. 而如果使用比较复杂, 可以深入探索深度学习库 API 状态的技术, 则需要耗费大量的人力成本. 例如, DocTer 技术需要人工构建规则提取文档特定信息以及手动注释一些 API 约束, Muffin 技术同样需要手动注释 API 约束保证变异的有效性. 综上, 如何进一步平衡人工成本和修复性能, 高效地生成符合深度学习库约束的测试, 是这两类技术的关键挑战.

2.3 基于大模型的深度学习库测试技术

大模型在自然语言和代码任务上的优越表现, 吸引了大量研究者的注意. 相较于模型级别和 API 级别测试依赖人工构建变异规则和手动注释 API 约束的方式, 大模型在处理深度学习库 API 约束和测试代码生成等方面有以下特性.

(1) 语义和语法理解: 大模型通过在大量代码库上进行预训练, 学习了编程语言的语义和语法规则 (例如, GitHub 上有超过 400 000 个 TensorFlow/PyTorch 项目). 这使其能够生成复杂且有效的代码片段, 从而深入挖掘深度学习库内部的复杂漏洞.

(2) 自然生成测试用例: 大模型的自回归生成性质使得它们可以基于给定的上下文 (例如, 一段代码的开始部分或者特定的 API 调用) 生成后续的代码. 该特性可以用于生成一系列复杂的 API 调用, 从而可以模拟真实应用场景下深度学习库的行为.

(3) 理解复杂类型和约束: 在训练期间, 大模型会在具有复杂 API 调用和类型约束的代码上进行训练并理解这些特征. 该特性使得即使在深度学习库 API 调用或数据类型非常复杂的情况下, 大模型仍然能够生成满足这些约束的测试输入, 对深度学习库进行有效的测试. 这个特性可以令大模型作为高效的深度学习库 API 的约束分析器, 自动理解分析深度学习库 API 的输入输出约束, 生成合法的测试输入.

(4) 灵活且自动化的代码突变: 大模型可以根据简单的提示以及自身所具备的随机性生成新的代码片段来替换已有的代码, 从而创建新的测试用例. 这个特性使得大模型本身可以作为强大的模糊测试变异器, 自动地对种子进行变异, 以完成对深度学习库的模糊测试.

因此, 研究者尝试应用大模型到深度学习库测试任务上. Deng 等人[50]率先提出了基于大模型的自动化深度学习库测试技术 TitanFuzz. 该技术首先在生成式的大语言模型上通过逐步输入提示以产生初始种子程序. 然后 TitanFuzz 使用大模型自动突变种子程序来产生新的测试程序以丰富测试程序的种子池. 最后, 该技术在不同的后端上使用差分测试执行生成的测试程序以检测漏洞. 实验结果表明, TitanFuzz 在发现漏洞数量, 漏洞多样性以及生成 API 序列多样性方面均优于传统技术, 并且在 PyTorch 和 TensorFlow 两个深度学习框架中发现了 30 个新的漏洞, 其中 27 个已经被开发者确认并修复.

在 TitanFuzz 基础上, Deng 等人[21]考虑到大模型学习的深度学习库代码已被世界各地的开发人员使用, 普通的测试程序几乎无法帮助覆盖额外的深度学习库行为/路径, 从而导致 TitanFuzz 模糊测试深度学习库的效果有限. 进而他们基于一个已知假设: 历史触发漏洞的程序可能包含对查找漏洞重要的边际情况或有价值的代码部分, 构建了 FuzzGPT. 该技术利用历史漏洞程序, 隐含地学习了深度学习库的约束条件 (包括语法/语义、深度学习计算约束以及新的异常约束), 并且能够完全自动化地进行测试. 通过收集目标深度学习库的漏洞报告以及触发漏洞的代码片段作为数据集, FuzzGPT 使用以下两个策略增强模型的测试能力: (1) 上下文学习: 研究者为大模型提供一些历史触发漏洞的程序, 以生成新的代码片段或自动完成部分代码; (2) 微调: 研究者通过在提取的历史触发漏洞

程序上进行训练来修改模型权重, 从而得到特别设计的大模型. 通过这些策略, 研究者得到了针对深度学习库漏洞检测定制的大语言模型, 这些大模型有能力生成与历史漏洞类似的代码. 实验结果表明, FuzzGPT 相较 TitanFuzz 分别在 PyTorch 和 TensorFlow 上提升了 60.70% 和 36.03% 的覆盖率, 并且在最新版本的 PyTorch 和 TensorFlow 上发现了 76 个漏洞.

2.4 小 结

深度学习库测试技术随着深度学习技术的广泛应用, 其重要性日益凸显. 自 2019 年 CRADLE 技术被提出后, 该领域的技术快速更新发展. 研究人员首先尝试直接将模型作为输入对深度学习库系统进行测试. 然而这些技术面对着测试输入模型数量有限而且耗时颇高、制定变异规则自动对模型进行突变成功率低等挑战. 因此, 研究者进而尝试在 API 级别上分析特定深度学习库 API 的输入约束, 并根据约束为 API 生成测试 (通常是单行测试). 但是在这个过程中, API 级别测试同样面临着深度学习库 API 约束复杂性的挑战. 为此研究人员尝试手动注释某些深度学习库 API 的约束, 以达到更好的测试效果, 然而这个方法往往伴随着大量的人力成本开销, 难以被推广应用. 伴随着大模型的火爆, 大模型的优越性能吸引了越来越多研究者的关注. 进而, 研究者尝试直接应用大模型到深度学习库测试任务中. 由于大模型具有强大的自然语言理解能力、代码生成能力以及训练中学习了大量深度学习代码, 研究者发现大模型可以自然地理解深度学习库 API 的约束条件, 进而构建测试代码, 完全无需人类专家手动构建规则和注释约束. 因而, 基于大模型的深度学习库测试技术达到了最优的测试效果.

有趣的是, TitanFuzz 和 FuzzGPT 在论文中都强调 LLM 隐含学习了深度学习库的使用和约束范式, 可以驱动深度学习库的测试技术, 并且进一步阐述这样的思路可以迁移到其他程序语言和应用上. 然而这 2 篇文章的实验仅包含了最流行的 PyTorch 和 TensorFlow 深度学习库, 并没有拓展到其他深度学习库上. TitanFuzz 和 FuzzGPT 的技术有效性建立在大模型学习了数十万个 GitHub 相关项目的基础上, 但是对于其他一些未有丰富训练语料的深度学习库, 大模型驱动的深度学习库测试能力还需要进一步探索. 目前大模型也在快速更新迭代, TitanFuzz 结合了 OpenAI 公司 CodeX 模型和开源模型 InCoder、FuzzGPT 使用了 CodeX 和 CodeGen 模型, 同样为 OpenAI 公司对话大模型 ChatGPT 进行了适配. 因此如何更好地利用更强大的开源模型 (例如 CodeLlama) 以及如何更好地结合闭源大模型 (例如 GPT-4) 获得更优的测试效果, 还是该领域下亟需探索的课题.

与深度学习库系统相关测试技术发展相类似, 随着移动应用的快速发展, 移动应用中的 GUI 自动化测试被越来越多的研究人员和开发者重视, 并积累了大量宝贵的技术. 因此第 3 节将聚焦于 GUI 自动化测试技术, 分析与总结大模型在测试 GUI 缺陷的最新技术的发展动向.

3 GUI 自动化测试技术

近年来, 移动应用 (APP) 得到了蓬勃发展, 在 Google Play 和 Apple APP Store 中有超过 300 万个应用供下载, 这些应用共积累了数十亿的下载次数[72,73]. 在移动应用中, 用户与应用的交互通常通过图形用户界面 (graphical user interface, GUI) 进行. 具体而言, 用户与 GUI 的交互包括点击、滚动或向 GUI 元素 (如按钮、图像或文本块) 输入文本等. 在现实中, 用户的体验直接与应用程序 GUI 的稳定性相关, 因此确保应用程序的 GUI 按预期工作至关重要. 然而, 在面对复杂的应用程序时, 人工测试 GUI 非常耗时且成本高昂, 难以被企业接受[74]. 诸多研究者尝试解决这一挑战, 提出了大量 GUI 自动化测试工具. 本节将首先简要介绍 GUI 自动化测试技术机制, 然后介绍传统和基于学习的技术以及这些技术所面对的关键挑战. 在此基础上, 分析大模型应用于这一任务的潜力并对目前基于大模型的 GUI 自动化测试相关工作进行介绍.

3.1 GUI 自动化测试技术机制简介

GUI 自动化测试涉及多个部分, 包括 GUI 测试输入生成、GUI 元素检测、GUI 测试框架、GUI 测试的记录与重放、GUI 测试脚本维护、GUI 测试报告分析以及 GUI 测试评估等[75]. 其中, GUI 测试输入生成是研究的重点, 因此本文将对这一部分进行重点介绍. 如图 6 所示, GUI 自动化测试机制可分为 3 个循环执行的步骤: (1) GUI

提取分析; (2) 测试输入生成; (3) 与应用程序互动. GUI 测试工具通过迭代地执行这些步骤以覆盖不同的 GUI 状态, 从而检测相关漏洞.

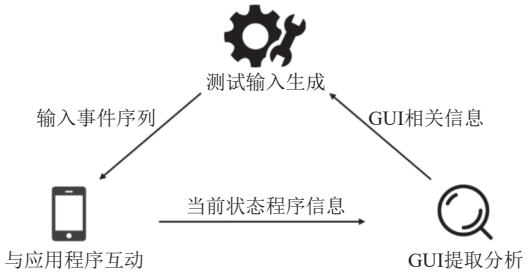


图 6　GUI 自动化测试总体流程

(1) GUI 提取分析: 这一步骤涉及对应用程序 GUI 信息的提取和处理. GUI 自动化测试工具需要从当前状态的应用程序中提取 GUI 信息, 分析应用程序的不同页面和其中的 GUI 元素. 这通常包括检测应用程序的 GUI 部件, 如按钮、文本框、菜单等, 并识别可触发的 GUI 事件. GUI 自动测试工具必须准确识别这些 GUI 部件和相应的可触发事件, 以便后续生成测试输入.

(2) 测试输入生成: 在 GUI 提取分析后, GUI 自动化测试工具将生成用于测试的输入事件序列, 其中包括 GUI 事件和系统级事件. GUI 事件是直接对应用程序的用户界面元素 (如按钮、文本框等) 进行交互的事件, 如按键输入、触摸屏输入、手势输入等. 系统级事件是指来自操作系统或硬件设备的事件, 如屏幕旋转、电池状态变化、来电等, 用于测试应用程序在不同系统条件下的响应情况. GUI 测试工具通过生成测试输入, 模拟真实用户在应用程序中的操作, 从而实现对应用程序的探索和测试. 测试输入事件生成历来是 GUI 自动化测试领域的研究重点, 传统的 GUI 自动化测试技术通常采用基于随机、基于模型和基于系统探索的方法来生成测试输入事件. 近年来, 研究者进一步尝试使用深度学习和强化学习等方法生成测试输入. 随着大模型技术被广泛研究, 研究者注意到大模型在自然语言理解与生成以及逻辑推理等方面的优异性能, 并尝试利用大模型引导测试工具生成测试输入事件, 以更智能地对应用程序 GUI 状态进行测试.

(3) 与应用程序互动: GUI 自动化测试工具生成的测试输入序列将被发送到真实或模拟的安卓设备上, 与目标应用程序 GUI 界面进行互动. 具体而言, 测试工具在不同 GUI 状态下执行输入事件序列并尝试触发应用程序的不同功能和界面, 以覆盖尽可能多的 GUI 活动. 在这个过程中, 测试工具将监视应用程序的响应, 并记录异常行为和错误.

在迭代的测试过程中, 测试工具还会统计已经探索的 GUI 状态和程序状态, 以计算代码和 GUI 活动覆盖率作为评估测试效果和测试工具性能的重要指标. 下面, 我们将分析传统和基于学习的 GUI 自动化测试技术中的经典工作和关键挑战.

3.2　传统和基于学习的 GUI 自动化测试技术

由于输入事件序列的质量对于 GUI 自动化测试工具的效果起到至关重要的作用, 研究者尝试应用不同的策略生成输入事件序列. 根据其策略的不同, 传统 GUI 自动化测试技术可分为 3 类: 基于随机的、基于模型的和基于系统探索的 GUI 自动化测试技术. 而近年来, 研究人员也开始应用深度学习和强化学习等技术进行 GUI 自动化测试, 我们将其归纳为基于学习的 GUI 自动化测试技术.

3.2.1　基于随机的 GUI 自动化测试技术

基于随机的 GUI 自动化测试技术根据启发式策略随机生成输入事件. 2007 年, Google 发布的安卓框架中便内嵌了最早的 GUI 测试工具 Monkey[76]. 该工具能够自动生成随机的 GUI 事件, 包括按键输入、触摸屏输入和手势输入等. 然而, Monkey 并不考虑应用程序自身的 GUI 结构, 因此它可能会生成大量冗余和无效的输入. 在实际应用中, 相较于程序 GUI 功能测试, 开发人员更多将 Monkey 应用在压力测试中.

Machiry 等人[77]在 2013 年提出了 Dynodroid. Dynodroid 被设计为观察者-选择器-执行器 (observer-selector-executor) 结构, 其中观察者负责分析当前页面可触发的 GUI 事件和系统级事件; 选择器根据不同随机策略, 基于观察者提供的信息选择并生成测试事件传递给执行器; 执行器将生成的测试事件在被测应用程序上进行执行. 相较于 Monkey, Dynodroid 的优势是: (1) 能生成系统级事件输入; (2) 可以结合测试人员生成的事件进行进一步的探索; (3) 经由观察者分析给出的事件输入排除了大量无效输入. 并且相较于纯随机探索, Dynodroid 会基于已探索事件的频率对策略中的概率参数进行修改, 使探索更加智能.

PUMA[78]是 Hao 等人在 2014 年提出的一个基于随机的 GUI 自动化测试框架. 该框架基于 Monkey 执行自动化的 GUI 应用状态探索, 支持对各种应用属性进行动态分析. 同时, 用户可以根据需求编写事件处理程序来指导应用状态的探索方式, 并可以在应用程序运行时动态修改环境. 这一设计使得 GUI 测试过程变得更加灵活, 能够满足不同应用的测试需求.

3.2.2 基于模型的 GUI 自动化测试技术

基于模型的 GUI 自动化测试技术在 GUI 自动化测试领域应用广泛. 此类技术以建模的方式对应用程序 GUI 行为进行抽象表示, 其建模方法涉及将应用程序的 GUI 状态变化与用户的交互操作 (也即输入事件) 相联系, 并通常采用有向图来呈现这些关系, 从而构建出一个反映应用程序行为的模型. 这样的模型不仅可以帮助 GUI 自动测试工具理解和预测用户界面的各种状态转换, 还为测试输入事件的生成提供了一定的指导.

2013 年, Orbit[79]通过静态分析程序源代码以提取 GUI 支持的操作, 然后使用动态遍历的方法在应用程序上执行, 最终得到以状态机表示的 GUI 模型. 作为早期基于模型的测试工具, Orbit 使用在深度优先搜索算法的基础上加以改进的图探索算法进行简单的 GUI 探索, 因此测试性能有限. 同年, Choi 等人提出 SwiftHand[80], 该技术应用机器学习方法在测试过程中构建并优化被测应用程序模型, 然后利用该模型指导生成用户输入以探索 GUI 程序状态. 同时, SwiftHand 通过减少被测应用重启次数以提高测试效率.

针对 GUI 测试中的状态爆炸问题, 2016 年 Mirzaei 等人[81]提出了 TrimDroid. TrimDroid 通过对应用程序进行静态分析得到 GUI 输入接口模型和 GUI 活动转换模型. 同时, 该工具结合控制流和数据流分析技术来识别应用程序 GUI 元素之间的依赖关系, 从而将 GUI 输入部件根据依赖关系划分成集合, 并将其作为组合测试的候选对象. 最后 TrimDroid 使用约束求解器计算出覆盖不同 GUI 部件组合的测试用例. 与一般的组合测试相比, TrimDroid 能够在保障达到一定代码覆盖率的同时减少生成冗余的测试事件序列, 有效缓解组合爆炸问题.

2017 年, Li 等人提出的 DroidBot[82]采取基于模型的探索策略, 首先将程序 GUI 状态和触发状态转换的输入事件构建为有向图, 然后该工具将探索应用状态的问题转化为图遍历问题. 此外, DroidBot 为用户提供一系列 API, 以支持用户自定义拓展相关策略, 并且该工具通过调用堆栈跟踪评估测试输入的有效性. 与先前的测试输入生成工具相比, DroidBot 更加轻量并适用于大多数安卓应用程序.

随机有限状态机是由程序的 GUI 状态和导致状态转换的输入事件以及相应的转换概率组成的有向图, 其中转换概率值反映了从一个状态到另一个状态的转换可能发生的概率. 由于应用程序都较为复杂, 状态繁多, 难以进行精确建模, 部分研究利用随机有限状态机的概率特性来近似地构建应用程序模型. 例如, 2017 年 Su 等人提出的 Stoat[83]技术便采用以随机有限状态机表示的随机模型对应用程序进行建模. 该工具通过静态分析和动态分析技术对应用程序进行逆向工程以得到随机模型, 并根据 Gibbs 采样[84]迭代地变异和优化模型, 进而引导测试生成. 与先前基于模型的方法相比, Stoat 通过注入系统级事件进一步增强了测试效果.

在 2019 年, Gu 等人提出的 APE[85]采用了基于决策树的方法来构建程序模型. 决策树通过节点和分支来表示各类决策规则和条件, 这使其能对不同的程序行为和属性进行抽象表示. 通过对决策树进行灵活的调整, APE 可以动态地更新模型, 对 GUI 信息的表示进行精确的增删, 进而改变模型的大小和精度. 这种方法不仅可以去除无关的 GUI 细节, 还能有效地反映被测应用程序的运行时状态, 从而提升测试的有效性. 在实验评估中, APE 的测试覆盖率和挖掘到的缺陷数量均高于 Stoat[83].

基于模型的 GUI 测试工具在应用于复杂应用程序时可能会陷入循环操作并耗尽资源. 针对这个问题, 2020 年字节跳动团队提出了 Fastbot[86]. 该工具通过在服务器端构建模型, 并采用多智能体协作机制, 以加速模型的构建.

Fastbot 采用多种启发式遍历算法以解决测试中的重复操作和资源枯竭等问题. 此外, Fastbot 将部分计算任务部署到云端, 在客户端仅保留监听 GUI 信息和动作注入功能, 以解决手机内存和计算资源大量消耗的问题. 在实验效果上, Fastbot 的多智能体云测试机制有效提高了 GUI 测试效率.

一些复杂的程序状态需要较长的测试输入事件序列来触发. 针对部分 GUI 测试技术在生成高质量的长测试输入事件序列上的挑战, Wang 等人于 2020 年提出了 ComboDroid[87]. 该工具将生成测试输入转化为用例生成和用例组合的反馈循环操作, 通过组合短测试输入事件序列生成高质量的长测试输入, 并从其生成的长测试用例中提取短用例作为组合材料, 以深入测试程序复杂 GUI 状态. ComboDroid 支持从 GUI 状态转换和数据依赖中提取应用程序的有效信息, 并利用轻量级静态分析来推断输入之间的依赖关系. 实验评估中, ComboDroid 的代码覆盖率优于当时的最佳基准线 APE[85].

2022 年, Liu 等人提出了 NaviDroid[88], 该工具结合自动化测试和手动测试, 以尝试解决自动化测试覆盖率较低和手动测试重复探索的问题. NaviDroid 通过静态分析和动态探索从应用程序中提取 GUI 状态转换图, 并使用动态规划算法指导路径探索, 以覆盖更多的程序状态并减少重复的探索步骤. 同时, 该工具采用上下文感知状态合并方法, 能有效地识别并整合相近的状态, 从而降低状态空间的冗余. 在手动测试过程中, 该工具通过视觉提示来引导测试者测试未探索的界面, 从而避免测试遗漏和重复探索等问题. 用户体验研究表明, NaviDroid 在协助手动 GUI 测试和挖掘缺陷方面具有优异的性能表现.

3.2.3 基于系统探索的 GUI 自动化测试技术

基于系统探索的技术使用符号执行和启发式搜索算法等技术来指导 GUI 测试工具探索特定程序状态, 侧重于解决特定应用程序行为只能通过特定测试输入触发的挑战. 2012 年, Anand 等人提出了第一个使用系统探索策略的测试框架 ACTEve[89]. 该框架使用符号执行技术, 通过检查程序执行条件来确定不同事件序列之间的包含关系, 从而避免重复访问等效程序状态, 并缓解路径爆炸问题. 同年, Amalfitano 等人提出的 AndroidRipper[90]基于已有的 Ripper 技术实现了对 GUI 的自动探索测试. 作为 AndroidRipper 的核心组件, Ripper 能自动识别当前 GUI 页面中可触发的事件, 并使用这些事件生成测试用例, 以触发应用程序状态的变化. 在 2013 年, Azim 等人提出的 A3E Depth-First[91]采用污点分析技术, 以深度优先方式探索应用的活动和 GUI 元素. 相较于 ACTEve, A3E Depth-First 注重对大规模应用程序进行 GUI 和传感器驱动的探索. 而 AndroidRipper 与 A3E Depth-First 相比, 则仅适用于安卓模拟器, 无法模拟传感器事件.

2014 年, Mahmood 等人提出了 EvoDroid[92]. 该技术使用进化算法, 将测试路径划分为段, 并根据这些段的信息执行进化和搜索, 以获取深入程序状态的测试用例. 这一方法有助于保留并改进搜索过程中的优秀测试用例的遗传信息. 此外, EvoDroid 还具备在云上并行执行测试用例的能力, 从而提高了测试的可扩展性和执行速度. 但需要注意的是, EvoDroid 在某些情况下会难以系统地推理输入条件, 导致性能显著下降.

2016 年, Mao 等人提出的 Sapienz[93]同样采用搜索和进化算法策略. 该技术采用帕累托多目标搜索方法组合最小化测试序列长度和最大化测试覆盖率等目标, 以实现对多目标的同时优化. Sapienz 使用进化算法中的基因编程技术来生成测试输入, 并采用多粒度级别的插桩和灵活的测试策略, 系统地探索应用程序. 相较于 Evodroid, Sapienz 同时考虑了覆盖率、测试序列长度、执行时间等多个目标, 提供更优的测试方案.

2020 年, Dong 等人提出了一种基于虚拟化技术的 GUI 测试工具 TimeMachine[94]. 该工具能够捕获系统状态快照保存重要的探索状态. 在探索陷入困境时, TimeMachine 将尝试恢复到先前记录的状态, 从而进一步触发新的程序行为, 实现系统性的状态空间探索. 与 EvoDroid 和 Sapienz 一样, TimeMachine 也采取了基于搜索的方法, 但它演化的对象是应用状态的种群, 而非输入序列的种群. 在实验评估中, TimeMachine 的代码覆盖率和发现缺陷数量均优于 Sapienz[93].

3.2.4 基于学习的 GUI 自动化测试技术

虽然传统 GUI 自动化测试技术已经得到了广泛的研究并在测试效果上不断取得进展, 但这些技术在面对复杂应用程序时仍然表现有限. 为此, 研究人员尝试应用深度学习和强化学习等机器学习方法进行 GUI 自动化测试. 这些方法通过模仿人类用户的行为, 生成与人类用户相似的动作和交互, 从而更有效地测试应用程序 GUI. 这些方

法的核心思想是: 测试算法执行的操作越接近人类用户的操作, 测试就越全面和有效.

Li 等人在 2019 年提出了一种基于深度学习的 GUI 自动化测试工具 Humanoid[95], 它采用深度神经网络模型, 从大量人类与应用程序 GUI 的交互数据中学习人类用户交互的偏好和方式, 进而使用该模型来指导输入事件的生成. Humanoid 在深度神经网络模型的帮助下能够以人类用户视角判断 GUI 事件的重要性, 从而优先生成能够触发重要状态的测试输入. 但在充足测试时间的情况下, Humanoid 是否使用学习模型对最终的测试覆盖率没有显著影响. 同时, Humanoid 不支持系统广播、传感器等系统级事件, 因而限制了该工具在测试覆盖率上的表现.

2020 年, Pan 等人提出了 Q-testing[96], 这是一种基于强化学习中 Q-learning 算法的 GUI 自动化测试工具. 该工具综合了随机探索策略和基于模型的探索策略的优点. Q-testing 采用 Q 表作为轻量级模型, 采用好奇驱动策略, 记录之前访问的部分状态以引导工具对探索较少的程序 GUI 状态进行测试. 此外, Q-testing 还引入了一个包含深度神经网络模型的状态比较模块, 用于根据功能场景的粒度划分 GUI 状态. 借助这个模块, Q-testing 能够比较不同 GUI 状态并计算相应奖励, 从而优先探索未曾到达的状态, 有效提高了代码覆盖率.

2022 年, Peng 等人[97]的研究认为 Humanoid[95]在具有更复杂 GUI 页面特征和交互逻辑的应用上的测试表现不佳, 这种情况的主要原因是 Humanoid 对 GUI 结构的表示有效性有限, 并且其使用的单模态模型限制了生成测试事件的多样性. 基于该见解, Peng 等人提出了基于深度学习的自动化测试工具 MUBot[97]. 该工具通过使用灰度图简化的 GUI 页面表示, 提高了对 GUI 元素的识别效果. 同时, MUBot 通过多模态深度学习算法[98]学习用户交互轨迹, 以生成多样化的测试输入. 相较于 Humanoid, MUBot 在模型规模和测试事件生成时间上均显著减少, 同时由于其生成的测试输入更加全面, 在面对复杂程序时该工具的测试覆盖率也有不俗的表现.

2021 年, Yazdani 等人提出了 Deep GUI[99]. 该工具采用深度学习技术, 尝试实现黑盒的 GUI 自动化测试. Deep GUI 运用深度学习分析应用程序的屏幕截图进而生成相应的热图, 后者可描绘屏幕上每个像素点被触摸的概率. 这使得 Deep GUI 能够根据这些热图生成智能化的测试输入, 高效地与应用程序的关键交互界面进行互动, 而无需深入了解应用程序的内部实现细节. 此外, 由于 Deep GUI 的数据收集、训练和推断过程独立于平台进行, 它不仅适用于安卓平台, 还具有跨平台的能力. Deep GUI 与 Humanoid 在采用应用程序屏幕截图作为输入方面相似, 但与依赖于程序分析以构建 GUI 状态转换图的 Humanoid 不同, Deep GUI 仅需要屏幕截图即可进行测试.

3.2.5 传统和基于学习的技术所面对的关键挑战

尽管传统和基于学习的 GUI 自动化测试技术被广泛研究, 相关技术经过不断的迭代在测试效果上有所进展, 但它们仍然面对一系列关键挑战. 其中, 在传统技术中有以下一些.

(1) 基于随机的测试技术虽然简单且易于实现, 但其主要挑战在于测试效率以及有效性难以保障. 随机生成的输入事件序列相对冗余, 大量测试输入事件难以有效地触发 GUI 状态变化, 这使得基于随机的测试技术往往测试效率低下, 测试覆盖率有限.

(2) 基于模型的测试技术需面对模型准确性的挑战. 该类技术难以对复杂应用程序进行准确建模, 因此无法精准捕获大型应用程序复杂状态的变化和行为. 此外, 对于基于模型的技术而言, 如何选择适当的模型粒度是一个重要的挑战. 模型粒度太细可能导致 GUI 状态重复甚至状态爆炸, 而模型粒度过粗可能导致测试工具无法准确获取 GUI 的状态信息, 从而产生错误的分析结果.

(3) 基于系统探索的技术通常使用符号执行技术和启发式搜索方法侧重于深入探索特定程序状态, 而在代码覆盖率等整体测试指标方面表现有限, 难以对程序进行较为全面的探索. 其中, 对于应用符号执行技术的 GUI 测试工具而言, 这类工具虽然能够深入探索特定程序状态, 但由于符号执行技术的路径爆炸问题, 其有效性容易受到应用程序代码复杂性的影响. 而启发式搜索算法虽然在处理大型应用程序和复杂场景时更加高效, 但它们通常依赖于预设的规则或经验, 无法覆盖潜在的测试路径, 导致这些工具可能遗漏重要的缺陷.

基于学习的 GUI 自动化测试工具生成的测试输入相较于传统工具更贴近人类与 GUI 进行交互的模式, 并在测试覆盖率的表现上有所提高, 但依然需要面对机器学习技术引入的关键挑战. 首先, 基于学习技术中模型的训练依赖高质量的训练数据, 而现实世界中来自真实用户的 GUI 交互数据数量有限, 难以获得. 其次, 基于学习的技术应用的模型在面对实际复杂应用时泛化能力有限, 难以达到较好的性能表现. 此外, 现实中移动应用程序非常复

杂, 涉及诸多关联数据和应用, 在执行相同操作的情况下, GUI 页面可能也并不相同 (例如, 在余额充足和不足的情况下, 点击购买按钮将会得到不同的反馈, 但余额未必会显示在当前的 GUI 页面上). 这使得基于学习的技术难以学习此类复杂情况并做出准确的预测[100].

除了前文所述的挑战之外, 传统和基于学习的 GUI 自动化测试技术在处理一些更为复杂的交互场景时表现不如人意. 这些场景包括测试工具与文本输入部件的交互、执行复合输入以及生成复杂的事件序列输入等. 在这些需要与 GUI 组件进行深度智能化交互的场景下, 现有技术往往难以达到理想的测试效果.

(1) 文本输入: 文本输入部件对于用户输入的文本通常有严格的格式规定和约束. 因此, 对于传统和基于学习的 GUI 自动化测试技术而言, 生成符合约束的文本一直是一项具有挑战性的任务. 面对这一挑战, 传统技术[82,83,85,86]尝试直接使用预定义的候选文本作为测试输入, 或分析并提取源代码和文本输入部件的约束并通过启发式规则生成测试输入[89]. 进一步, Liu 等人[101]利用 RNN 模型对文本输入相关部件进行预测, 但该工具需要大量人工编写的数据用于模型训练, 并且没有充分考虑上下文语义信息. 综上, 研究者尝试了诸多方法解决这一挑战, 但效果依然有限[102].

(2) 复合的事件输入: 部分 GUI 页面需要同时对多个部件进行输入 (即复合的事件输入) 才能访问后续 GUI 状态. 例如, 填写日期信息需要选择页面上“年”“月”“日”这 3 个部件. 传统或基于学习的 GUI 自动化测试技术通常只能迭代地生成单个输入事件, 并且难以分析输入事件之间的逻辑联系, 因此在处理复合的事件输入上效果不佳[100].

(3) 复杂事件序列输入: 某些 GUI 状态需要复杂的输入 (例如, 特定的 GUI 操作序列) 才能触发[103], 而这些操作往往和应用程序的功能逻辑紧密相关. 例如, 测试一个电子商务应用程序的购物功能, 可能需要浏览商品、选择商品、选择相关型号、选择支付方式、输入送货信息、确认订单共 6 个步骤的操作才能完成. 而传统和基于学习的 GUI 自动化测试技术难以分析测试输入事件间的联系, 在理解应用程序的功能逻辑上效果不佳, 因此难以对此类功能进行较为有效的测试.

总的来说, 现实中的移动应用程序功能复杂, 不同的应用场景可能包括数百甚至数千个不同的 GUI 页面, 并且每个 GUI 页面都具有独特的交互逻辑[103]. 传统和基于学习的技术虽然能生成多样的测试输入事件, 以实现一定的测试覆盖率, 但依然测试效果有限, 并且在理解应用程序的功能逻辑和 GUI 交互逻辑上存在一定挑战. 因此, GUI 自动化测试迫切需要一种更加智能的技术, 用以模拟真实用户的交互行为, 从而更智能地测试移动应用程序.

3.3 基于大模型的 GUI 自动化测试技术

如第 3.2.5 节所述, 传统和基于学习的 GUI 自动化测试技术存在一定挑战, 在实际应用中仍需测试人员手动构造测试脚本和文本输入. 并且, 在移动应用快速迭代的今天, GUI 页面的频繁变更导致自动化测试的维护成本较高. 此外, 由于传统和基于学习的技术优化目标大多以测试覆盖率为导向, 在真实场景下的 GUI 漏洞往往存在误报漏报等问题, 进而影响这些技术落地应用的效益. 与此同时, 研究者们注意到大模型在自然语言理解、生成与逻辑推理等方面均表现出优异的性能, 并体现出一定的智能. GPT 模型的强大的理解能力证明大模型可以理解人类的知识, 对 GUI 页面的状态进行自动化判断, 并可以模仿人类与 GUI 界面进行互动而无需测试人员手动构建交互脚本[104]. 具体而言, 鉴于前文关于传统和基于学习的技术所面对的关键挑战 (见第 3.2.5 节), 将大模型用于 GUI 自动化测试, 主要有以下几个方面的特性.

(1) 智能的文本输入生成: 许多文本输入部件有着严格的格式规定、语法规则等输入约束条件限制, 只有满足这些约束条件的文本输入才能触发后续的 GUI 状态. 大模型在预训练过程中充分学习了大量相关信息和知识, 拥有强大的自然语言生成能力, 并能够深入理解应用程序 GUI 的上下文语义信息. 因此, 大模型在理解 GUI 文本输入部件的隐含约束进而生成满足约束的文本输入上具有显著优势[102].

(2) 生成复合的事件输入: 部分复杂的 GUI 页面需要同时对多个 GUI 部件进行输入, 才能访问后续状态. 大模型的训练语料库中包含了大量的应用程序文档和测试报告等相关信息的自然语言描述, 这些描述有助于大模型理解 GUI 的复合操作, 从而使得基于大模型的 GUI 自动化测试工具有可能实现复合的输入事件生成.

(3) 复杂事件序列的生成: 部分应用程序的 GUI 缺陷需要复杂的事件序列才能触发. 目前, 大模型具有较强的

上下文记忆能力和学习能力, 能完成上下文关联的多段输出[104]. 同时, 在大模型训练的语料库中包含大量应用程序的使用教程和测试报告, 大模型在训练阶段已经学习了触发应用程序某个功能或重现某个缺陷的分步说明. 因此, 基于大模型的 GUI 自动化测试工具具有生成相关联的复杂事件序列的能力. 当向大模型提供了应用程序的结构和语义信息以及测试历史后, 大模型可以捕获 GUI 页面之间的历史依赖关系, 从而生成复杂的输入事件序列.

(4) 优先探索重要程序状态: 大模型的训练数据包含大量应用程序版本迭代和修复的历史信息, 这些信息通常将应用程序的重要功能、频繁迭代而没有充分测试的部件以及出错率高的功能突出. 因此, 基于大模型的 GUI 自动化测试技术隐含地了解这些重要测试信息, 并可以优先探索那些更为关键和更易发现缺陷的程序状态[100].

(5) 应用程序快速变更下的低成本测试: 目前移动应用的 GUI 页面随着版本迭代迅速更新, 大模型可以通过提示技术或微调技术对需求和产品手册进行学习, 进而快速了解应用程序的功能和 GUI 页面的变更, 而无需测试人员在每次版本变更都手动维护测试脚本. 此外, 目前基于大模型的 GUI 测试技术仅需与 GUI 页面翻译转换而成的文本进行交互, 自然地解决了 GUI 页面变化对整体测试流程的影响. 因此, 基于大模型的 GUI 测试技术在当今应用程序快速变更下可以以较低成本进行 GUI 自动化测试.

(6) 高可信度的 GUI 缺陷报告: 由于真实移动应用程序的复杂性, 传统 GUI 自动化测试常常出现误报和漏报等问题. 一个具体的情景是: 同样的测试用例, 在同样的测试执行环境下, 测试的结果有时正确, 有时错误. 这样的情况极大降低了 GUI 自动化测试的有效性. 而大模型在预训练阶段已经充分学习了应用程序的正常行为以及常见错误类型, 因此可以高效自动化地判别 GUI 缺陷是否准确, 提升了 GUI 缺陷报告的可信度.

鉴于大模型在 GUI 自动化测试任务上的特性, 研究人员尝试将大模型与 GUI 自动化测试相结合, 以克服过去传统和基于学习技术的关键挑战, 实现更加智能的 GUI 自动化测试.

针对传统 GUI 自动化测试工具在文本输入上表现较差的问题, Liu 等人提出了 QTypist[102]. 该工具使用经过提示微调 (prompt tuning) 后的 GPT-3 模型在文本输入部件中生成高质量的文本输入. 首先, 研究者对大量需要文本输入的应用程序页面进行了分析和调研, 并进行手动分类以确定需要输入文本内容的 GUI 部件类型. 然后, 对于一个文本输入部件及其 GUI 页面, QTypist 从视图层次文件中提取上下文信息, 经过处理后生成输入 GPT 模型的提示词. 最后, 根据 GPT 的反馈, QTypist 得到输入文本, 并填入相应的文本输入部件中, 从而实现与文本输入部件的交互. 研究者为进一步分析 QTypist 的真实使用体验, 让测试人员使用不同 GUI 自动测试工具并进行 5-Likert 尺度反馈调查. 调查结果表明, 测试人员强烈同意 QTypist 生成输入内容具有多样性和有效性, 并对其他自动 GUI 自动测试工具生成的文本输入表示强烈不满, 认为这些工具生成本文内容没有语义含义和实际意义. 此外, 在实验评估中, QTypist 的文本输入通过率为 87%, 相较过去工作有显著提升. 同时, 集成了 QTypist 的传统 GUI 自动化测试工具的覆盖率均有所提高, 并检测到更多的缺陷, 验证了 QTypist 在 GUI 自动化测试中的有效性.

不同于 QTypist[102]侧重于生成能够通过 GUI 文本输入部件的有效文本输入, InputBlaster[105]旨在利用大模型生成能导致程序崩溃的异常文本输入. InputBlaster 的核心思想是将异常文本输入生成问题转化为利用大模型生成文本输入测试生成器 (即一个程序函数, 其输出一系列用于对文本输入部件进行测试的异常文本) 的任务, 每个生成器都能根据相应的变异规则生成一系列异常文本输入. 具体来说, InputBlaster 首先从程序中提取 GUI 信息, 利用大模型生成有效文本输入, 以及该文本输入部件的输入约束规则. 进一步, InputBlaster 利用生成的有效文本输入以及约束规则构建新的提示词, 以要求大模型生成变异规则 (即如何从有效文本输入变异生成异常文本输入的自然语言描述) 和相应的测试生成器. InputBlaster 通过将测试生成器生成的测试文本输入到移动应用程序中, 检验这些输入能否触发应用崩溃. 应用程序的反馈随后被用来迭代优化大型语言模型, 从而生成新的变异规则和测试生成器, 实现持续的循环测试过程. 同时, 为使大模型考虑更加丰富的变异规则, InputBlaster 结合上下文学习模式, 通过为大模型提供触发异常文本输入错误的历史报告作为学习样本, 以增强模型的性能. InputBlaster 在实验中对 31 个热门 Android 应用程序的 36 个文本输入小部件进行了评估, 显示出 78% 的错误检测率, 比现有最佳基准高出 136%. 此外, 研究者将 InputBlaster 集成到自动化 GUI 测试工具中, 并在 Google Play 的实际应用程序中检测到 37 个之前未见的崩溃. 这些特点使 InputBlaster 在移动应用程序的异常文本输入测试领域表现突出, 有效提高了 GUI 自动化测试的深度和覆盖率.

GPTDroid[23,100]是一个基于大模型的 GUI 自动化测试工具. 该技术将 GUI 自动化测试转化为与大模型的问答任务, 通过将 GUI 信息输入大模型得到测试指引, 然后根据大模型输出的测试指引迭代地进行测试. 具体来说, GPTDroid 提取应用程序的整体信息 (如应用程序名, GUI 页面的活动名), 当前的 GUI 页面信息和部件信息, 并结合历史测试信息 (如功能, 活动, 部件的累计触发次数和最近执行的测试操作), 综合处理后形成输入 GPT-3 的提示词, 以询问 GPT-3 当前需要进行的 GUI 测试操作, 并根据 GPT-3 的回答提取操作指引, 最后生成测试输入事件. 为了使得大模型的输出易于识别, 并能被正确映射到实际被应用程序执行的输入事件上, GPTDroid 应用少样本学习策略, 通过为大模型提供输入输出模板的学习样例以指引大模型生成符合严格格式要求的操作指引. GPTDroid 在 Google Play 的 86 个应用程序上进行了评估, 在活动覆盖率和测试效率上相较过去工作均有显著提升. GPTDroid 还在 Google Play 上检测到 48 个新漏洞, 其中 25 个已被确认并修复. 研究者进一步分析并讨论了 GPTDroid 综合表现优异的原因. 他们发现 GPTDroid 可以: (1) 自动进行有效的文本输入: 通过利用大模型的上下文理解能力, GPTDroid 可以自动分析出 GUI 页面的关键点, 并尝试补全相关信息以通过 GUI 部件的输入校验; (2) 完成大模型引导的复合操作: 在研究者给出的例子中, 大模型自动对 GUI 页面的不同部件输入顺序进行分析, 并生成合理且有效的测试输入; (3) 追踪测试上下文信息: 在一个应用程序 GUI 漏洞检测案例中, GPTDroid 自动化地进行了 6 步操作, 包含点击、选择、长按、输入文本等, 最终触发了漏洞. 如果没有强大的上下文理解和操作能力, GPTDroid 无法如此深入地测试 GUI 程序状态; (4) 避免重复探索: 部分应用程序的设置部件为方便用户使用存在循环跳转的情况, 这使得 GUI 自动化测试工具很容易陷入无效的 GUI 状态循环. 而 GPTDroid 通过上下文记忆和理解能力可以高效地跳出 GUI 状态循环并优先探索重要 GUI 状态.

主流的针对 GUI 自动测试工具的评价指标通常是测试工具对 GUI 活动的覆盖率或对组件的覆盖率. 然而, 一项对安卓开发者的实证研究[106]表明, 比起具有高覆盖率的测试用例, 开发人员普遍更倾向于使用针对单个功能的测试用例. 不仅如此, 开发人员还希望自动生成的测试用例能够具有自然语言描述的任务目标, 以及预期的程序输出. 为了满足这些需求, Yoon 等人提出了 DroidAgent[107]. 针对被测应用程序的功能, DroidAgent 提出相应的测试任务 (比如使用即时通讯软件完成一次聊天), 并生成对该任务的自然语言描述. 然后, DroidAgent 与被测程序的 GUI 进行交互, 以期完成先前提出的任务. 如果测试成功, DroidAgent 将生成复现测试过程的测试脚本, 并将脚本和任务描述提供给开发者. 具体来说, DroidAgent 的组成部分包括 4 个基于大模型的任务执行模块, 即计划模块、动作模块、观察模块和反馈模块. 此外, 为了应对大模型上下文长度受限的问题, DroidAgent 还引入了 3 个辅助记忆模块, 即长期记忆模块、短期记忆模块和空间记忆模块. 计划模块从长期记忆模块中得到初始知识 (DroidAgent 所扮演的测试者的个人信息等) 和先前任务的总结, 然后根据这些信息生成新任务. 接收任务后, 动作模块根据短期记忆模块中的历史操作记录和被观测到的 GUI 组件选择接下来执行的操作, 然后观察模块将对 GUI 状态更新的总结写入短期记忆模块和空间记忆模块, 引导动作模块选择下一个操作. 当动作模块宣布任务结束时, 反馈模块将生成对任务的总结和反思, 并将这些信息输入到长期记忆模块. 通过使用任务驱动测试过程, 并对任务进行自动的总结和反思, 大模型可以充分利用先前测试过程反映的有效信息, 从而生成符合开发者意图且具有多样性的测试任务. 在对 15 个应用程序的测试中, DroidAgent 生成了 374 个测试任务, 其中 85% 的任务是相关且可行的, 59% 的任务由 DroidAgent 成功完成. 此外, 尽管并未直接将覆盖率作为测试目标, DroidAgent 达到了 60.7% 的活动覆盖率, 高于其他作为基准的 GUI 自动化测试工具.

大模型除了在移动应用的 GUI 自动化测试领域展现巨大潜力, 在 Web 应用的 GUI 自动化测试, 移动应用 GUI 自动化交互等 GUI 自动化测试相关领域也发挥显著作用. 传统的 Web 元素定位方法通常缺乏对语义和上下文的理解, VON Similo LLM[108]通过将过去的 Web 元素的定位工具 VON Similo 与大语言模型 (LLM) 进行结合, 利用大型语言模型 (如 GPT-4) 的语言理解和推理能力, 实现了更准确的 Web 元素定位. 该工具能分析 Web 元素的含义, 理解 Web 元素相邻的文本, 评估网页结构, 并在自动化测试过程中减少人工干预和脚本维护的需求. 但需要注意的是, 该工具需要大量调用大模型 (例如, GPT-4) 的 API, 会产生 API 交互请求延迟和成本问题. DroidBot-GPT[109]展现了大语言模型在移动应用 GUI 自动化交互领域的作用. 该工具使用大模型自动化地与安卓移动应用程序进行交互. 它借助基于模型的 GUI 自动化测试工具 DroidBot[82]提取并处理应用程序的 GUI 信息, 进一步生成

提示词以引导大模型输出操作指引, 并最终根据大模型提供的操作指引生成相应的安卓应用程序输入, 以实现移动应用 GUI 自动化交互任务. 在性能评估方面, 结果显示 DroidBot-GPT 成功完成了 39% 的应用程序交互任务, 平均完成进度约为 66%.

3.4 小 结

GUI 自动化测试技术旨在自动化地对应用程序进行 GUI 测试, 挖掘 GUI 缺陷并反馈错误报告, 其工作机制可分为提取并分析程序 GUI 信息、测试输入生成、与应用程序进行互动 3 个阶段. 其中测试输入生成历来是 GUI 自动化测试领域研究的重点. 根据生成输入事件序列的方式的不同, 传统的 GUI 自动化测试技术可分为 3 类: 基于随机的, 基于模型的和基于系统探索的 GUI 自动化测试技术. 基于随机的测试技术应用不同的策略随机生成测试输入事件, 但由于输入事件不够精准导致其存在冗余和无效的问题; 基于模型的技术对应用程序进行分析以构建模型, 进而以模型指导测试事件生成, 但这些技术需要面对模型不够准确的挑战. 基于系统探索的技术采用符号执行和启发式搜索方法, 侧重探索程序特定状态, 然而该技术难以保障测试的覆盖率. 由于传统技术面临一系列挑战, 研究人员尝试应用深度学习和强化学习等基于学习的方法进行 GUI 自动化测试. 这些方法通过模仿人类用户的行为, 从而更有效地进行 GUI 测试. 然而基于学习的方法对训练数据集质量和规模要求较高, 模型泛化能力有限, 因而在面对复杂程序时效果仍然有限. 与此同时, 研究人员发现, 在其他领域的任务上, 大模型在自然语言理解、生成与逻辑推理等方面均表现出优异的性能, 并体现出一定的智能. 受此启发, 部分研究者将大模型应用于 GUI 自动化测试, 并发现其在文本输入生成, 生成高质量复杂输入事件, 以及优先探索重要程序状态上具有显著优势. 例如, QTypist[102]在文本输入部件中使用经过提示微调后的 GPT-3 模型生成高质量的文本, 大幅提高了 GUI 自动化测试面对文本输入组件的通过率. 另一个工作是 GPTDroid[100], 该工具将 GUI 自动化测试转化为基于 GPT-3 模型的问答任务, 并在活动覆盖率和测试速度上相比传统方法有显著的提升. 这些研究成果展示了大模型在 GUI 自动化测试领域的巨大潜力. 在未来基于大模型的 GUI 自动化测试研究中, 如何对大模型进行微调以更好地适应 GUI 测试任务, 以及如何构建更有效的提示策略, 将是值得探索的方向.

基于此, 本文已经对深度学习库缺陷检测和 GUI 自动化测试领域进行了分析和探讨. 第 4 节我们将聚焦于被开发者广泛使用以保障其代码质量和有效性的方法: 测试用例生成技术.

4 测试用例自动生成技术

在当今的世界, 随着市场需求的快速变化, 大量技术公司采取敏捷开发的模式, 以最快的速度进行软件交付. 然而, 经过多次迭代更新后, 软件通常会愈发庞大和复杂, 其往往隐藏着未知的缺陷, 而即使是微小的缺陷也可能给公司带来重大的经济损失和声誉损害. 因此, 为保证软件的质量, 软件测试在软件开发生命周期中的作用变得愈发重要[110,111]. 软件测试的基础和关键部分是单元测试[112], 它涉及对特定代码功能进行测试. 在单元测试中, 测试用例首先设置一系列输入, 然后依次调用被测试的方法 (称为焦点方法), 最后使用测试断言来判断程序状态是否符合预期. 单元测试可以帮助开发者在软件开发的早期阶段发现缺陷和错误, 从而降低开发的成本. 此外, 单元测试还提高了软件应用的可维护性和可扩展性. 然而, 手动编写单元测试需要开发人员深入理解软件系统的各个组成部分的功能, 对开发者的技术水平和专业知识都有较高的要求. 这使开发者在面对繁重的开发任务时, 可能会忽视或简化单元测试的工作[113]. 在这种情况下, 通过使用测试用例自动生成工具, 开发者可以节约手动编写单元测试的时间, 在保证软件质量的同时降低开发成本, 提高开发效率. 在本节中, 我们将介绍传统和基于学习的测试用例自动生成技术, 以及这些技术的代表性工作和面临的关键挑战, 然后对基于大模型的测试用例自动生成技术进行介绍和分析.

4.1 传统和基于学习的测试用例自动生成技术

传统的测试用例自动生成技术可以被大致分为 3 类: 基于搜索的, 基于随机变异的和基于约束的测试用例自动生成技术. 在传统技术之后, 神经机器翻译 (neural machine translation, NMT) 和文本到文本转换器 (text-to-text transfer Transformer, T5) 等技术也被应用于测试用例生成, 这些技术被归类为基于学习的测试用例生成技术. 本节

将对这些传统和基于学习的技术的代表性工作进行介绍, 指出各技术路线的发展目标和面临的挑战. 需要注意的是, 在很多工作中, 为了增强工具性能或针对特定功能进行优化, 这些技术被结合使用以达到更优的效果.

4.1.1 传统的测试用例自动生成技术

基于搜索的测试用例自动生成技术将测试用例生成视为一个优化问题, 将覆盖率等评价指标作为优化目标, 利用启发式搜索等算法求出使目标达到较优水平的测试用例. EvoSuite[6]是具有代表性的基于搜索的测试用例自动生成技术, 该技术通过进化算法为给定的 Java 类自动生成测试用例. 具体来说, EvoSuite 采用基于搜索的方法, 集成了混合搜索和动态符号执行等技术, 并使用进化算法优化测试用例的覆盖率. 目前 EvoSuite 已经通过插件的形式集成到 Maven、IntelliJ 和 Eclipse 中, 并且被应用到一百多个开源软件和一些工业系统上, 发现了数千个潜在的漏洞. 然而, EvoSuite 生成的测试与人工编写的测试在可读性上有较大差距, 导致在实际应用中开发人员并不愿意采用 EvoSuite 自动生成的单元测试[114]. 在面向对象程序中, 方法通常对复杂的类对象输入具有隐含的合法性要求, 使得生成工具很难通过随机变异构造合法的类输入, 导致生成大量无效的测试用例. 为了应对这一问题, 在 EvoSuite 的基础上, Lin 等人开发了 EvoObj[115]. 通过结合静态分析技术, EvoObj 通过建立类对象构造图 (object construction graph) 来表示焦点方法中的控制流和数据流信息, 并基于类对象构造图生成初始的测试用例提供给 EvoSuite. 相比于仅使用 EvoSuite, EvoObj 在 SF100 数据集[116]及其他 3 个开源 Java 项目上具有更优异的表现.

基于随机变异的测试用例自动生成技术由随机测试发展而来. 随机测试的变异过程缺乏启发式方法的指导, 导致其很难生成有效的测试用例. 在这种情况下, 研究人员开发了反馈驱动的随机测试生成算法, 该算法利用执行已有测试用例时获得的反馈, 指导变异产生新的测试用例. Randoop[5]是一个经典的基于随机变异的测试用例自动生成技术. 在构建测试的过程中, Randoop 执行测试用例得到反馈, 并使用这些反馈信息指导搜索以产生新的、合法的对象状态序列. 在这个过程中, Randoop 随机选择要调用的方法或构造函数, 使用先前计算出的值作为输入迭代地构建测试. 因此, 该技术生成的测试通常由一系列创建和改变对象的方法调用, 以及对方法调用的结果的断言组成. 通过将测试生成和测试执行相结合, Randoop 成为一种高效的测试生成技术, 在很多广泛使用的库中发现了大量漏洞. 时至今日, Randoop 仍继续被工业界使用[117]. 然而, 先前的反馈驱动的随机测试生成算法只能顺序地测试焦点方法, 而无法为回调函数构造测试用例. 为解决这一问题, Selakovic 等人提出了 LambdaTester[118], 该工具通过分析预测焦点方法参数的位置及相关信息, 以更好地生成测试用例. 在 13 个 JavaScript 库中, LambdaTester 生成测试用例的覆盖率均高于忽略焦点方法嵌套的随机生成工具, 且由复杂嵌套构成的测试用例在 12 个库中发现了相关缺陷. 之后, Arteca 等人提出了 Nessie[119], 该工具使用测试用例树 (test case tree) 作为测试用例的中间表达, 进而生成具有复杂调用嵌套的测试用例. 实验数据表明, Nessie 在测试用例的有效性和覆盖率表现都优于 LambdaTester.

基于约束的测试用例自动生成技术根据程序的约束条件指导生成测试用例. 对于程序来说, 约束指一系列分支条件, 这些分支条件使程序沿一特定路径执行. 主流的基于约束的技术通常通过推断不变量 (即程序执行某一部分时, 部分变量恒满足的条件) 得到程序约束, 进而约束求解器可以直接根据程序约束生成测试. Daikon[120]是一个经典的基于约束的技术, 它首先运行初始的测试用例, 然后追踪程序运行时的数据, 并验证预设的不变量类型 (即不变量模板) 是否符合数据行为, 从而推断测试用例运行过程中的不变量. 测试工具通过分析这些不变量, 进一步分析出测试用例覆盖率不足的原因, 从而改进测试用例. 在许多相关工作中, Daikon 都被用作性能基准进行比较[116,121,122]. 在诸多基于约束的技术中, 研究者通过套用预先设置的不变量模板进行约束分析, 用户也可以根据自身需求加入自定义的不变量模板. 然而, 在复杂程序中, 随着需要追踪的变量组合的迅速增长, 以及变量行为的复杂化, 不变量推断的成本也呈指数级增长. 因此, 动态不变量推断系统通常只专注于一小部分简单的候选不变量. 基于此, Csallner 等人提出了一个经典的基于约束的测试用例自动生成技术 DySy[121], 该技术结合动态符号执行技术进行不变量推断, 从而更高效地捕捉不变量. 与过去技术不同, DySy 同时进行测试用例执行和动态符号执行, 进而采集程序的分支条件构造不变量, 有效地改进了先前基于约束的工具.

4.1.2 基于学习的测试用例自动生成技术

虽然传统测试用例自动生成技术可以在一定程度上生成达到有效覆盖率的测试用例, 然而, 相较于开发者手

工编写的测试, 自动生成的测试通常可读性较差并且难以理解[110], 尤其是其变量名和函数名令开发者困惑[114]. 此外, 自动生成的测试断言有效性较差, 例如仅包含常规的断言或是不相关断言[123], 因而开发人员在实践中大多不愿直接采纳[110]. 为此, 研究者进一步尝试应用深度学习以及神经机器翻译等方法, 通过更充分地学习数据集上开发人员编写的断言语句, 生成更加自然的断言.

Watson 等人[124]提出一种自动生成断言语句的方法 Atlas, 该工具利用神经机器翻译, 可以自动生成语义和语法正确的单元测试. 为训练神经机器翻译模型生成与开发者编写风格相近的断言语句, 研究者从 GitHub 中分析了超过 9000 个使用 JUnit 断言类的 Java 项目, 并从相关测试方法中提取了超过二百万个开发人员编写的断言语句作为训练数据. 在给定的测试用例和其对应的待测方法的情况下, Atlas 可以将这些输入“翻译”为适当的断言语句. 实验结果显示在 31.4% 的情况下, Atlas 可以自动生成与开发者手动编写完全一致的断言.

为了达到和 Atlas 相同的目标, 后续 Mastropaolo 等人[125,126]尝试使用迁移学习的方法, 通过对一种文本到文本转换器模型进行预训练和微调, 进而处理单元测试中的断言生成任务, 而 Yu 等人[127]使用信息检索技术进一步改进 Atlas 生成的断言语句. 最近 Nie 等人提出了一种深度学习方法 TECO[128]. 与 Atlas 相比, 该工具在利用语法层面信息的基础上, 进一步将代码语义等信息与深度学习相结合, 获得了更好的效果.

不同于基于循环神经网络 (recurrent neural network, RNN) 的 Atlas[124]和 Mastropaolo 等人[125,126]基于文本到文本转换器模型的技术, Tufano 等人[129]在富语义的英文语料库上对 BART 转换器模型进行预训练, 然后在大型代码语料库上对模型进行半监督训练, 最后在断言生成任务上进行微调. 实验显示, 在相同的测试数据集上, 该模型生成断言语句的 Top-1 准确率达到 62%, 相较于 Atlas 和 Mastropaolo 等人的模型分别进步了 80% 和 33%. 他们还尝试将断言生成用于增强传统工具生成的测试用例, 发现自动生成的断言语句能够增加由 EvoSuite 生成的测试用例的覆盖率.

Dinella 等人[130]发现由开发者编写的单元测试中的断言通常遵循一组常见模式. 基于此, 他们通过人工分析并提取这些模式的分类以及对应的约束, 从而在满足句法和类型正确性的条件下, 构建了一套断言生成技术 TOGA. 具体而言, TOGA 基于待测试的程序文档字符串以及带有遮蔽断言的测试用例前缀对代码模型进行微调, 然后通过模型决定给定方法是否需要断言来测试其异常行为. 之后, 该技术应用人工预定义的断言分类法生成断言, 并最后通过神经网络模型进行排序. 尽管 TOGA 在其实验评估中有不俗的表现, 并与 EvoSuite 集成创建了一个端到端的测试生成工具, 最近的研究[131]指出了评估方法的几个缺陷, 使得其报告结果的有效性受到怀疑.

除了仅生成断言外, Tufano 等人[132]提出了一种基于 BART 转换器模型的自动化测试生成工具 AthenaTest. 对于给定的测试用例, 该工具依赖启发式策略识别正在测试的焦点类和方法, 然后使用这些测试用例来微调模型, 通过将此任务表示为将焦点方法 (以及焦点类等) 映射到测试用例的翻译任务以产生单元测试. 在基于 Defects4J 缺陷数据库的实验中, AthenaTest 达到了与 EvoSuite 相当的测试覆盖率, 并且该工具生成的测试用例在代码风格上接近开发者编写的测试用例, 具有更好的代码可读性.

如第 1 节所述, 大语言模型是指具有更大参数规模和预训练数据集的预训练语言模型. 上述两个 Tufano 等人完成的工作中[129,132]都使用了转换器架构的预训练模型 BART, 并使用大量自然语言文本和代码片段对模型进行预训练, 其工作原理已十分接近大语言模型. 随着算力的提升和开源语料库规模的扩展, 基于大模型的技术呼之欲出.

4.1.3 传统和基于学习的技术所面对的关键挑战

尽管传统和基于学习的测试用例自动生成技术被不断改进, 这些技术仍然面临着一系列关键挑战.

(1) 基于搜索的技术由于难以将生成测试的质量和有效性可读性等方面纳入其搜索函数, 因而容易生成低质量的测试代码. 研究显示[123,133], 基于搜索的技术很容易生成难以理解的断言, 并且同时测试一个焦点类中过多的焦点方法, 或生成包含大量重复代码的测试用例, 使测试代码难以被开发者理解和维护.

(2) 基于随机变异的技术受限于庞大的突变空间, 生成的测试无法很好覆盖边际条件, 因而很难检测到需要特定输入数据触发的缺陷. 一项研究显示[110], 在对需要特定的输入数据引发故障的方法生成测试用例时, 基于随机变异的技术检测到的缺陷数量远低于基于搜索的技术.

(3) 基于约束的技术难以生成精确的测试断言. 该类技术根据焦点方法的前置和后置条件, 推导出由不变量表达的程序规范, 从而自动生成断言语句以检查程序行为. 然而, 上下文信息中往往也隐藏着很多约束条件, 仅从焦点方法直接提取约束生成高质量的断言是非常有挑战性的.

除上述技术各自面临的挑战外, 传统技术还普遍面临如下挑战: (1) 焦点代码上下文相关信息难以有效利用; (2) 生成测试用例多样性有限; (3) 生成的断言难以捕获缺陷并可能有大量误报. 以上这些挑战都在很大程度上阻碍了传统技术在工业界的大规模应用.

基于学习的工作应用机器学习和强化学习等技术于测试自动生成领域并达到了较为优异的效果, 其生成的测试用例在可读性和有效性上都有明显的提升. 然而, 该类技术依然需要面对训练数据质量和数量上的挑战. 现有基于学习的测试用例生成技术通常使用开源项目中的测试代码作为数据集对模型进行训练. 具体而言, 研究人员根据注释或文档等相关信息提取出测试代码, 并进一步根据启发式规则进行过滤 (如断言数量[124]或重复数据[125]). 然而, 一些开源项目包含了修复缺陷的版本, 在这些版本中, 原本用于检测缺陷的测试用例已经失效[131], 进而导致这些失效的测试用例成为训练数据集中的噪声, 影响模型生成测试用例和断言的性能. 此外, 在训练过程中, 基于学习的方法也容易在有限的数据集上过拟合, 导致在真实场景下表现不佳.

因此, 基于学习的技术仍然面临高度依赖训练数据的质量和数量上的挑战. 总的来说, 由于训练数据集规模不足和过拟合等原因, 基于学习的技术并面对生成的测试用例缺乏多样性和有效性较低的关键挑战. 综上, 目前测试用例自动生成任务仍迫切需要一种简单易用并且智能的技术, 以生成符合软件功能和开发者意图的测试用例.

4.2 基于大模型的测试用例自动生成技术

目前基于学习的测试用例自动生成技术重点在于生成断言, 并且训练数据主要集中于开发者编写的单元测试. 部分基于学习的测试用例自动生成技术使用模型驱动生成测试用例, 然而其采用的模型在上下文理解和代码能力方面相较 CodeX 和 ChatGPT 等大模型有较大差距. 而基于神经机器翻译的技术则面临训练语料质量和数量不足, 以及训练过拟合等问题, 在应用范围上有局限性. 因此, 研究者进一步尝试将上下文学习能力和代码生成能力更强的大模型应用到测试用例自动生成领域.

使用基于搜索的技术进行测试生成时, 初始随机生成的测试输入可能并没有很好地适配待测程序, 导致难以通过针对初始测试用例进行变异生成有效的测试用例. 为解决这一问题, Lemieux 等人提出了 CodaMosa[134], 该工具结合了基于搜索的软件测试 (search-based software testing, SBST) 和大型语言模型, 在基于搜索的工具陷入覆盖率停滞时, CodaMosa 应用 CodeX 模型生成覆盖程序更多行为的自动测试用例以增强基于搜索的工具, 帮助开发人员发现并改进程序中的错误. 实验评估表明, 相对于仅采用搜索或大模型的测试用例生成技术, CodaMosa 实现了显著的覆盖率提升, 有效地提高了测试用例生成的效率和效果. 然而, CodaMosa 输入 CodeX 模型的提示只包括函数实现和生成测试用例的指令, 并没有更多探讨不同设置下的效果. 此外, 其生成的测试用例采用了 Mosa 格式, 作者承认这可能会导致可读性下降, 并且这些测试用例并不包含断言.

在 CodaMosa 基础上, Schäfer 等人[135]更加精细地探索了 CodeX 模型的不同设置条件对于测试用例生成质量的影响, 并提出工具 TESTPILOT. 该工具利用函数签名、文档注释、测试用例示例和源代码等进行提示设计, 然后生成并执行测试用例. 如果测试用例执行失败, TESTPILOT 使用包含失败测试和其报错信息的特殊提示再次驱动 CodeX 模型重新生成新的测试用例. 在实验评估中, TESTPILOT 在 25 个 npm 软件包上达到了目前最优异的语句覆盖率效果.

如上文所介绍, 开发人员编写测试的目的之一是防止漏洞或者程序错误再次发生. Kang 等人[22]通过分析使用 JUnit 的 300 个开源项目样本, 发现有 28% 的测试是为复现错误报告中的漏洞所添加的. 这表明根据错误报告生成复现错误的测试是一种被低估但影响重大的自动生成测试的方式. 因此, 他们提出了一种名为 LIBRO 的工具, 该工具将缺陷报告作为对大语言模型的提示, 从而生成可复现错误的测试案例, 帮助开发者有效诊断并修复错误. 研究者在两个数据集上进行了实证分析, 发现 LIBRO 可以根据缺陷报告为 251 个缺陷 (占所有研究缺陷的 33.5%) 生成至少一个重现缺陷的测试. 此外, LIBRO 能够以 71.4% 的准确率判断其重现缺陷的尝试是否成功. 相

较于其他方法, 在生成可复现错误的测试用例方面, LIBRO 取得了显著提升.

Yuan 等人[47]对目前最先进的对话型大语言模型 ChatGPT 在生成测试方面进行了充分的实证研究. 实验结果表明: ChatGPT 生成测试用例的覆盖率和可读性均与开发人员编写的有效测试用例相近, 这些测试用例有时甚至更受开发人员欢迎. 然而 ChatGPT 自动生成的测试用例的成功率不高, 仅有 24.8% 能够通过执行, 其余面临编译错误或执行失败等问题. 在此基础上, 他们提出 CHATTESTER 工具, 该工具包括一个初始测试用例生成器和一个迭代测试用例修复器, 通过 ChatGPT 与自身迭代对话, 提高其生成测试的有效性. 实验结果显示, 相较于直接使用 ChatGPT, CHATTESTER 在编译率和执行通过率方面分别提高了 34.3% 和 18.7%. 然而该技术也受限于生成式语言模型本身的局限性, 例如 ChatGPT 对目标项目代码的深层信息缺乏了解, 以及对待测的焦点方法理解不足.

和 Yuan 等人[47]的发现类似, Xie 等人[136]也发现直接应用 ChatGPT 生成的测试用例仅有 20% 的成功率, 并进一步发现这样的结果主要是因为 ChatGPT 受限于有限的上下文令牌数量, 例如 GPT-3.5 仅支持 4096 个令牌, 以及由于缺少适当的编译器和测试执行器, ChatGPT 无法验证生成的测试. 在这些发现基础上, 他们提出 ChatUniTest 工具, 该工具由 ChatGPT 驱动, 通过自适应上下文生成机制控制 ChatGPT 的上下文空间, 并进一步引入了验证和修复组件以提高生成测试的成功率. 实验显示, ChatUniTest 生成了 30% 的正确测试, 并在多个项目中相较主流工具有显著优势. 然而 ChatGPT 驱动的测试自动生成技术还共同面临一些挑战, 如处理大型代码时的令牌限制, 以及缺乏适当的编译器和测试执行器来验证生成的单元测试等.

Chen 等人[137]提出了 CodeT, 该工具尝试将测试用例自动生成技术应用到基于模型的代码生成任务中, 为生成的代码提供测试并验证其有效性. 具体而言, CodeT 首先使用大模型生成待测代码, 然后采用零样本提示策略, 使用同一模型为该任务生成测试用例. 在对测试用例的分析中, Chen 等人发现, 由 CodeX 代码大模型生成的测试用例的准确度和覆盖率均优于其他大模型. 而且, 大多数大模型的测试用例覆盖率都在 94% 以上. 结果显示, 通过使用大模型生成的测试用例进行自动验证, 大模型在代码生成任务中的性能得到了大幅提高.

此外, 在利用大模型进行代码生成的场景中, 开发者与大模型使用自然语言交互可能存在偏差, 导致模型生成的代码有缺陷或难以理解. 为解决这一问题, Lahiri 等人[138]提出了 TiCoder, 该框架采用测试驱动的用户意图形式化 (test-driven user-intent formalization, TDUIF) 的工作流程. 在该工作流程中, 模型将用户意图中表述不清晰的部分转换为测试用例, 并让用户决定该测试用例是否体现了所需的代码功能, 从而让模型生成更符合用户意图的代码. 在 MBPP 数据集的测试中, TiCoder 的效果较为优异.

4.3 小 结

本节简要介绍了测试用例生成的目的和机制, 并分析了不同技术的代表工作和特性以及对应的关键挑战. 测试用例自动生成的技术多而庞杂, 不同的技术经常被研究人员混合应用, 从而提升工具性能. 然而, 传统和基于学习的工具始终面临覆盖率低、断言无效、测试用例多样性差等挑战. 考虑到大模型拥有强大的解决复杂问题的能力, 研究人员进一步尝试将大模型应用于测试用例生成任务中. 具体而言, 大模型在被用于测试用例生成时, 它们不仅单纯地生成测试用例, 还可以利用更多上下文及代码相关信息并且可以与其他技术相结合. 此外, 基于大模型的技术还可以采用更复杂的生成策略, 以达成更好的测试效果. 部分研究者还尝试利用大模型生成测试用例以检验自动生成的代码或更清晰地表达用户需求.

许多工作已经表明, 大模型在自然语言与形式化表达的互译等方面具有相当的灵活性, 通过实施复杂的任务流程或提供丰富的程序相关信息, 其在测试用例生成等任务上可以达到更好的效果或满足传统工具无法实现的功能. 在这种情况下, 面对实践中无穷多的开发需求和应用场景, 大模型在测试用例生成乃至复合的软件工程任务中潜在的应用场景仍有待研究人员发掘. 比如, 在某些对时间敏感的应用场景 (如竞赛代码、音视频处理、游戏项目等) 中, 程序的时空效率应被有效测试. 时空开销问题几乎不被当前的单元测试技术视为缺陷, 但又会影响用户体验, 而由于大语言模型具有理解代码功能和算法过程的潜力 (比如, 通过简单提示, GPT-4 给出了与 AlphaDev[139] 相同的排序优化算法[140]), 它们可能可以被用于针对程序运行效率进行测试用例生成.

本文已经探讨了大模型在深度学习库的缺陷检测、GUI 自动化测试及测试用例自动生成技术中的应用及进

展. 这些章节介绍了大模型在理解复杂代码功能、生成有效的测试方案方面的能力. 同时, 部分研究者也尝试使用大模型的智能解决当下程序自动修复的挑战. 因此, 第 5 节我们将探讨软件缺陷自动修复领域, 这是大模型在软件缺陷领域中的又一应用.

5 软件缺陷自动修复技术

目前, 随着现代信息技术的迅速发展, 软件程序早已被深度应用到生活的各个领域, 例如监控金融交易[141], 控制交通系统[142]以及协助医疗工具[143]. 然而, 随着软件规模的不断扩大, 软件缺陷的数量也在逐步增加, 并且实际的软件程序往往会在存在已知和未知缺陷的情况下发布[144], 导致大量隐藏缺陷的存在. 这些缺陷会破坏程序的正常执行, 使程序在一定程度上不能满足其既有的功能需求. 更为严重的是, 这些缺陷可能引发巨大的经济损失, 甚至对人们的生命安全造成威胁. 然而寻找和修复缺陷是一项困难、耗时且人力密集的工作. 调查研究显示, 在程序的开发过程中, 开发者大约要花费一半的时间在修复缺陷上. 在美国, 软件维护的年度总成本高达 700 亿美元[1]. 开发者修复程序中的缺陷既耗时又易出错, 甚至有可能在修复过程中引入新的缺陷[2], 使得软件缺陷的修复变得更加困难. 因此, 及时有效地修复程序中的缺陷显得尤为重要. 为了减轻开发者的负担, 提高修复效率, 软件缺陷自动修复 (automated program repair, APR) 技术应运而生. 不过与复杂的深度学习库系统架构不同, 现实中的软件通常遵循传统基础的开发范式和架构, 因此, 过去研究者使用的传统技术在自动修复任务上已经初有成效. 然而在现实中, 软件程序随着程序规模的增大, 往往变得越来越复杂, 传统的技术也面对越来越严峻的挑战. 进而研究者尝试使用机器学习等技术进一步提升修复能力. 本节将对该领域技术历史发展轨迹和面临的挑战进行阐述和分析. 具体而言, 本节将首先介绍缺陷自动修复机制, 然后介绍传统和基于学习的缺陷自动修复技术和其代表工作, 进而本文尝试分析这些技术所面对的关键挑战. 在此基础上, 本文尝试分析大模型驱动软件缺陷自动修复技术的特性并介绍相关工作, 最后进行展望和总结.

5.1 软件缺陷自动修复技术机制简介

软件缺陷自动修复系统通常可以分为 3 个模块: 缺陷定位 (fault localization)、补丁生成 (patch generation) 以及补丁验证 (patch validation). 如图 7 所示, 当给定一个缺陷程序, 修复模块 (1) 首先通过缺陷定位技术确定程序中可能的缺陷位置, 然后模块 (2) 应用补丁生成技术尝试修复缺陷代码, 模块 (3) 最后对生成的补丁进行自动化或者人工的验证. 在这个系统中, 软件缺陷自动修复技术往往聚焦于模块 (2)(3), 尝试针对漏洞生成有效补丁, 然后使用排序算法或者启发式规则对生成的补丁进行过滤和排序, 最后验证补丁的有效性. 目前的工作中, 修复中止条件通常是设定一个固定的时间, 在这个时间内, 自动修复工具反复迭代生成补丁尝试修复漏洞, 或是达到修复目的中止修复流程. 接下来我们分别介绍缺陷数据库以及自动修复系统中的 3 个模块.

图 7 软件缺陷自动修复技术机制

● 缺陷数据库: 在实证研究中, 由于缺乏被广泛接受且易于使用的真实项目构成的缺陷数据库, 实证研究的可重复性在软件测试领域一直是一个挑战. 手动植入的缺陷往往与开发人员无意中引入的缺陷存在差异, 并不适合评估测试技术. 为解决这样的困难, 2014 年 Just 等人[145]提出基于 Java 语言的用于软件测试研究的 Defects4J 缺陷数据库. 该数据库集成了大量真实的程序缺陷以支持相关研究, 目前已经被广泛应用于评估自动修复工具的修复能力上. 此外, 为衡量自动修复工具在不同程序语言和不同场景的修复效果, 研究人员还构建了另外一些缺陷数据

库, 例如一个包含 40 个经典编程问题的多语言缺陷数据库 QuixBugs[146], 以及包含从 9 个开源项目中收集的 185 个错误基于 C 语言的缺陷数据库 ManyBugs[147].

● 缺陷定位: 缺陷定位是修复流程中的首要步骤, 并且缺陷定位的有效性及准确性极大影响缺陷自动修复的效果. 因此, 缺陷定位技术引起了研究者的广泛关注, 并主要分为 3 个技术流派: 基于切片[148,149]、基于频谱[150,151]以及基于统计的[152,153]定位技术. 其中, 基于频谱的缺陷定位技术由于能够在语句级别定位错误, 在自动修复系统中得到广泛的应用. 该技术依赖于排名指标 (例如 Trantula[154]、Ochiai[155]) 来计算每个语句的可疑性. 由于在多项实证研究中已经证明了它们的有效性, GZoltar[156]和 Ochiai 被广泛整合到自动修复系统中[157−160]. 然而正如 Liu 等人[161]论文中所阐述: 这种缺陷定位技术的配置仍然存在定位错误位置准确性和有效性上的局限. 因此, 研究人员试图使用新技术来增强错误定位技术, 如谓词切换[162,163]和测试用例净化[164,165]. 值得一提的是, 近期已有一篇基于大模型的缺陷定位工作发表[166]. 该工作通过在已有大模型基础上微调一系列双向适配器层以实现缺陷定位, 并在性能评估中取得了高于最佳基线工作的效果. 由于缺陷定位是一个独立的研究领域, 其中相关工作繁多, 而且和软件缺陷自动修复技术解耦, 在本节中我们只在此做简要介绍, 更多工作进展请参考相关论文及综述[167−169].

● 补丁生成: 补丁生成模块是自动修复系统的核心. 目前传统的补丁生成技术可以分为 3 类: 基于启发式搜索的、基于约束的以及基于模板的技术. 其中, 基于启发式搜索的技术主要依赖人类专家应用遗传算法为搜索修复补丁制定的启发式规则, 基于约束的技术采用约束求解器进一步提高补丁生成效率, 基于模板的技术依赖于人类专家对缺陷特征进行分类并专门编写对应的修复模板. 后续研究者尝试应用机器学习和深度学习等技术到缺陷修复任务上, 提出了基于学习的缺陷自动修复技术. 在该技术中, 研究者使用神经机器翻译 (neural machine translation, NMT) 技术进一步提升了缺陷修复的效果. 然而, 基于学习的技术也面对诸多挑战. 诸多研究的实验结果显示, 在给定正确缺陷位置的前提下, 已有的自动修复方法仅能修复小部分程序缺陷[170]. 因此目前的缺陷自动修复系统面临的核心挑战是如何高效正确地生成补丁. 而大模型在这一方面展现出优异的性能: 在近期 Xia 等人[171]的研究中, 使用预训练大语言模型 CodeX 生成修复补丁达到了领域中最好的修复效果, 并且耗时上也远远优于传统技术. 故大量研究者尝试进一步开发利用模型的修复能力. 各个技术的代表工作会在第 5.2 节和第 5.3 节中介绍.

● 补丁验证: 当拥有易于验证修复的缺陷数据库后 (例如, Defects4J), 自动程序修复领域广泛应用生成和验证 (G&V) 模型, 该模型首先利用故障定位技术缩小搜索空间, 然后生成候选修复补丁, 并通过测试套件对其进行编译和验证. 然而, 由于测试的覆盖范围不全面, 即使补丁通过了所有测试 (称为可信补丁), 该补丁在其他输入下可能仍然存在失败的情况, 因此需要进一步的人工检查以确定最终正确的补丁. 然而, 由于需要反复执行测试, 该方式的修复效率较低, 通常需要几十分钟到几个小时来修复一个程序缺陷, 从而限制了其在线使用和实时反馈的能力. 虽然一些研究已经尝试通过不同的方法提升修复效率, 如避免重复补丁的验证过程、通过补丁排序筛选正确的修复补丁、以及通过补丁隔离和字节码修复来避免代码反复编译的开销[172], 但自动修复方法仍难以满足开发者对高效进行缺陷修复的需求, 修复效率仍是自动修复方法面临的核心挑战.

综上, 本文主要针对缺陷自动修复中的补丁生成技术在第 5.2 节和第 5.3 节进行详细的分析和探讨. 对于另外两个模块以及缺陷数据库的相关内容, 受限于篇幅, 本文不展开论述. 更多信息可以参考相关的综述论文[1,167−169,173].

5.2 传统和基于学习的缺陷自动修复技术

根据补丁生成的方式, 传统软件缺陷修复技术可以分为 3 类: 基于启发式搜索的、基于语义约束的和基于模板的缺陷自动修复技术. 进而, 研究人员尝试将深度学习和强化学习等技术应用于缺陷自动修复任务中, 我们将其归纳为基于学习的缺陷自动修复技术. 在本小节中, 我们将首先简要介绍各个传统技术的代表工作, 然后进一步分析不同技术所面对的关键挑战.

5.2.1 基于启发式搜索的缺陷自动修复技术

基于启发式搜索的缺陷自动修复技术应用遗传算法, 并利用人工定义的启发式规则, 通过变异生成通过所有测试用例且不存在特定缺陷的程序补丁从而自动修复软件缺陷. 其中 GenProg[174]是一个经典的基于启发式搜索的技术. 为解决遗传变异算法补丁搜索空间过大的问题, GenProg 仅在抽象语法树 (abstract syntax tree, AST) 级别

对缺陷代码进行变异, 并使用相关测试用例来评估新的变异程序的适应度, 其中具有高适应度的个体将被选中进行持续的进化, 直到生成保留程序功能且没有漏洞的程序补丁. 实验成果表明, GenProg 有效地解决了 16 个 C 程序中的错误, 涵盖了 8 种缺陷类型, 如死循环、段错误、缓冲区溢出等. 这验证了 GenProg 在实际运用中的效能, 进而激发了大批科研人员对自动修复领域的研究兴趣.

Wen 等人[175]发现, 可疑代码的上下文信息能够隐含地表示修复漏洞的可能性. 基于这一发现, 他们提出了一种基于上下文的缺陷自动修复技术 CapGen. 该技术利用开源项目中大量真实漏洞补丁的上下文信息, 解决基于启发式搜索的自动程序修复技术中的搜索空间爆炸问题. 在 Defects4J 缺陷数据集上, CapGen 为 21 个漏洞成功生成补丁, 并且准确率达到 84%. 但是由于 GapGen 对搜索空间的高度约束, 导致可修复的漏洞数量和范围比较局限.

在过去的自动修复工具中, 针对 Java 语言的自动修复工具往往从源码级别对程序进行修复, 需要反复编译程序导致时间开销过高, 因此于 2019 年 Ghanbari 等人提出 PraPR[172]技术尝试解决这一挑战. PraPR 在 Java 虚拟机 (JVM) 字节码级别进行操作, 采用了一系列简单的字节码突变规则. 为了进一步提升修复效率, PraPR 产生的所有补丁都可以无需编译直接进行验证. 相比于过去需要编译和加载每个候选补丁的技术, 在效率上具有显著的优势. 实验评估表明, PraPR 可以明显提高修复漏洞的数量和效率. 然而由于 PraPR 在 Java 字节码上进行修复, 部分程序语义信息在修改编辑过程中丢失, 导致其漏洞修复能力受限.

5.2.2 基于约束的缺陷自动修复技术

基于约束的缺陷自动修复技术通过约束求解和程序合成生成修复代码. 在 2013 年, Nguyen 等人提出 SemFix 工具[176], 这个技术包含 3 个部分: (1) 应用 Tarantula 缺陷定位方法[177]计算出语句缺陷概率, 即尝试定位缺陷位置; (2) 语句级约束推理, 即使用符号执行技术自动发现错误语句的约束; (3) 程序合成, 即使用程序合成思想合成符合约束的语句. 在 Defects4J 数据集的实验结果中, SemFix 成功修复了 13 个漏洞.

DeMarco 等人提出缺陷自动修复工具 Nopol[7]. 该工具主要针对两类错误模型: 错误的 If 条件和程序前置条件缺失 (即某个方法或函数在执行之前需要满足的某些条件缺失), 这两类错误都属于程序条件错误的范畴, 且在实际应用中非常常见. Nopol 将包含至少一个失败测试案例的程序和其相关测试作为输入, 通过逐个分析失败测试案例执行的语句, 尝试找到可能存在修复方案的语句. 在找到可能的修复位置后, Nopol 通过代码插桩从相关测试执行中收集信息, 然后将这些运行时信息转化为可满足性模块理论 (SMT) 问题, 如果存在解决方案, SMT 求解器便生成源代码补丁.

JAID[8]是一种基于程序约束的自动修复技术, 旨在解决基于有限测试集的自动化程序修复容易过拟合的问题. JAID 的操作机制类似于先前基于约束的修复技术, 但并不依赖于用户编写的约束条件, 而是通过分析目标缺陷状态建立大量约束, 从而进行修复补丁的生成和验证. 其中, JAID 主要依赖无副作用的函数来描述状态, 并采用缺陷定位和启发式的排序方法来识别可能与故障行为有关的程序状态, 从而减少需要生成和验证的补丁的数量. 实验结果表明, 在 Defects4J 缺陷数据集中, JAID 生成了 31 个可信补丁以及 25 个与程序员编写的修复方案相近的正确修复.

5.2.3 基于模板的缺陷自动修复技术

基于模板的缺陷自动修复工具使用开发者或者研究人员根据经验预定义的一些修复模板或者修复策略以修复特定类型的缺陷. 2013 年 Kim 等人提出了基于模板的缺陷自动修复工具 PAR[11], 该工具通过分析开源项目中大量漏洞的修复代码, 从中提取出几种常见的修复模式, 并为自动修复工具针对 6 类漏洞创建了 10 个修复模板. 在实验中, PAR 相较 GenProg 有更好的修复效果. 此外, 在面向开发者的调查中表明, PAR 生成的漏洞修复代码更具可读性, 也更容易被开发者接受.

部分基于模板的缺陷修复技术需要迭代地编译与测试修复补丁以获得验证信息. 在这个过程中, 编译和测试时间开销非常高. SketchFix[12]技术聚焦于模板修复技术中的效率问题. 该技术通过利用运行时信息在测试验证过程中大幅削减修复候选方案的数量, 以进行更有效的 G&V 程序修复. 在 Defects4J 缺陷数据集的评估结果中, SketchFix 在默认设置下平均 23 分钟内能正确修复 19 个漏洞, 相较于过去的基于模板的修复技术提高了修复的效率.

近期提出的 TBar[170]修复技术对过去基于模板的缺陷自动修复工具进行了整合, 总共集成了 35 种修复模板,

并通过大量的实验对各种修复模板的有效性进行了全面评估. 在实验中, TBar 达到了基于模板的修复技术中最优的修复性能: Defects4J 中 74 个错误在具有完美缺陷定位信息的情况下可以得到修复. 在后续的自动修复工作中, TBar 多次被用作模板技术修复性能基线进行比较.

5.2.4 基于学习的缺陷自动修复技术

为进一步拓展自动修复工具的修复范围和能力, 研究人员尝试应用机器学习和深度学习技术生成补丁. 其中广泛使用的神经机器翻译技术 (NMT) 旨在通过神经网络学习缺陷代码到修复代码的转变模式尝试将有漏洞的目标代码翻译为修复代码. 其中神经机器翻译模型通常由编码器和解码器组成, 编码器捕捉有漏洞的代码元素和其上下文信息, 解码器接受编码输入并生成修复补丁. 2020 年 Lutellier 等人应用神经机器翻译技术 (NMT) 构建了缺陷自动修复工具——CoCoNuT[178]. 具体而言, CoCoNuT 采用了一种上下文敏感的 NMT 架构, 它使用两个独立的编码器分别处理错误代码和上下文信息. 这种设计不仅减小了输入序列的长度, 提高了处理效率, 同时也能更有效地捕捉上下文中的有用信息, 使模型可以学习并生成更复杂、更精确的修复代码. 另外, 为了应对代码修复的多样性问题, CoCoNuT 还结合了不同复杂度的模型来捕捉错误代码和修复代码之间的多种关系. 实验结果表明, CoCoNuT 在 6 个缺陷数据集上成功修复了 509 个缺陷, 其中 309 个缺陷是过去的自动修复工具无法修复的. 在当时, 该工具出众的效果展现出 NMT 模型强大的能力, 吸引了众多研究者的研究兴趣.

2021 年 Jiang 等人[179]提出了卷积网络实现的基于编码器和解码器的自动修复模型 CURE. CURE 主要包含 3 种策略: (1) 应用预训练的代码模型学习开发人员编写的源代码, 从而生成高可读性的补丁; (2) 实现编码感知的搜索策略, 以提高修复补丁的有效性; (3) 利用过滤技术在推理阶段减小修复补丁搜索空间, 加速修复过程. 通过这 3 种策略, CURE 不仅提升了修复的效率, 还可以通过更有效的搜索策略高效地查找正确的修复方案. 同年, Zhu 等人提出了 Recoder[180]. 此方法以现有的编码器-解码器架构为基础, 旨在解决目前基于深度学习的自动修复方法在修复效率和正确性上的不足. Recoder 的特点在于引入了两个新颖的解码技术, 用于在抽象语法树级别生成修复. 这样的机制可以使 Recoder 生成更多语法正确的补丁, 以更少的解码步骤来修复缺陷.

2022 年 Ye 等人提出了基于 Transformer 架构的神经修复模型 RewardRepair[181]. 该模型将基于标记的语法训练目标与基于程序执行的语义训练目标相结合. 在训练期间, 该模型还将修复补丁的执行信息应用于损失函数计算中, 从而使模型学会生成可以编译且正确的补丁. 2023 年 Jiang 等人提出了基于深度学习的缺陷自动修复技术 KNOD[182], 该技术主要包括两个创新: (1) KNOD 使用图转换器和解码器来生成用于修复的抽象语法树, 使模型能自然地捕获抽象语法树中的结构, 有助于模型学习抽象语法树的语法和语义; (2) KNOD 在解码器中融合了一个领域知识蒸馏组件, 使用领域知识指导模型学习代码的语法和语义. 实验结果显示, KNOD 的修复性能超过先前其他技术.

5.2.5 传统和基于学习的技术所面对的关键挑战

基于启发式搜索, 模板和约束的传统缺陷修复技术共同面对这样的挑战: (1) 搜索空间有限; (2) 无法生成多编辑修复补丁; (3) 对程序上下文和依赖缺乏了解. 这些挑战导致这类工具可拓展性不强, 往往需要开发者和研究人员对特定的漏洞类型进行大量的工作 (如编写模板或者添加约束条件). 下面我们进一步分析不同技术面对的主要挑战.

基于启发式搜索的缺陷自动修复技术具有较强的通用性, 可以应用于不同规模的程序和不同种类的程序缺陷. 该技术的核心是在搜索空间内探索有效补丁并进行验证. 因此, 合适的搜索空间和搜索策略是找到有效补丁的关键. 众多研究者在过去的数十年中对两者进行了详细的探究, 进而发现: 启发式搜索的空间过大会导致探索效率较低, 并且会生成大量的待验证补丁, 降低自动修复系统效率; 搜索空间过小会导致无法找出有效补丁. 最新的评估结果表明, 该类技术修复效果有限, 并且非常依赖人工设定的启发式搜索策略. 因此, 如何精细地设置和调整启发式搜索策略并提高修复性能是该技术所面对的主要挑战.

基于约束的缺陷自动修复技术应用约束求解尝试推断程序的正确规约, 进一步指导补丁生成. 这种技术的优点是可以充分利用已知的约束优化求解算法, 进而降低生成无效补丁的概率. 但是基于约束的缺陷自动修复技术引入了符号执行以及约束求解等技术, 不可避免地受限于这些技术自身的局限性. 在比较复杂和大规模的代码上, 符号执行和约束求解技术由于路径爆炸等问题求解效率过低, 难以实际应用. 此外, 基于约束求解的技术还容易过拟合并生成可读性差的代码片段, 降低补丁的可信程度并提高代码的维护难度.

基于模板的缺陷自动修复技术通过从历史已经修复的缺陷中提取修复模板. 该类技术的优势是: 开发人员人工对缺陷特征进行分类并手动实现对应的漏洞模板, 因而针对特定类型的漏洞通常可以达到令人满意的修复效果, 在速度和修复准确率上都有优异的表现. 然而, 基于模板的缺陷自动修复工具主要面临缺陷特征分类困难和缺陷模板开发成本高昂等挑战. 研究表明[183], 在实际的程序修复中, 仅有 20% 的缺陷存在重复, 大部分缺陷无法通过人工分类缺陷特征以提供补丁模板. 此外, 添加新的缺陷修复模板需要研究者或者专家花费大量时间提取特征手动编写, 成本高昂. 所以, 基于模板的缺陷自动修复工具只能针对特定缺陷, 而无法应用到其他漏洞上.

为解决上述传统技术所面对的挑战, 近年来许多基于学习的缺陷修复技术被提出. 实验结果证明, 相较于基于启发式搜索, 模板和约束的缺陷修复技术, 基于学习的技术更具有泛用性也更有效. 然而现有的基于学习的技术仍然面临以下问题:

(1) 训练数据的质量: 现有的基于学习的缺陷修复工具需要使用历史缺陷修复数据集, 即由缺陷代码和对应补丁构成的数据集, 对模型进行训练或微调. 研究者通常使用一些启发式策略来提取这些数据, 例如使用一些关键词 (如 bug、fix、patch、solve 等) 来过滤提交记录[178,180,184,185]. 然而, 这些错误修复提交记录可能包含了与缺陷修复无关的编辑, 如代码重构或新功能实现[186], 导致训练数据集中存在各种不相关的提交记录和代码更改. 因而缺陷修复数据集的质量一直是该类技术面对的主要挑战.

(2) 训练数据的数量: 与大量的开源代码片段相比, 与漏洞相对应的修复代码数量相对较少. 为了减少数据集中补丁代码包含其他不相关补丁代码的影响, 基于学习的自动修复工具通常将其数据集中的补丁代码的长度限制为仅有几行[178−180,187], 从而进一步限制了训练数据的数量. 在这些有限的历史修复上进行训练, 基于学习的工具修复能力可能受到限制, 导致无法修复需要多处编辑的复杂缺陷.

(3) 上下文表示: 缺陷的上下文信息对于基于学习的技术中的模型理解补丁代码修复场景是至关重要的. 当前的基于学习的缺陷修复工具首先将上下文以纯文本[178,179]或结构化表示[180,187]的方式传递给编码器, 然后将编码后的上下文与有缺陷的代码片段输入传递给解码器. 然而这样的方式是不自然的, 因为模型很难分析程序缺陷和上下文编码的位置关系. 因此, 这种技术可能会忽略缺陷程序代码片段和其上下文之间的复杂关系, 影响模型的修复性能.

因此在基于学习的技术中, 仍面对: (1) 修复容易过拟合; (2) 多编辑修复有效性低; (3) 对程序依赖缺乏了解等挑战. 并且其修复性能极大依赖于训练数据的质量和数量, 而缺陷修复数据集的搜集和清洗又是困难且耗时的工作. 因此如何更好地收集和整理缺陷修复数据集以及如何更好地对缺陷和修复代码进行编码和学习, 仍是基于学习的技术亟待探究的问题.

5.3 基于大模型的软件缺陷自动修复技术

相较于基于学习的技术在收集缺陷数据集, 提取代码表征和上下文信息处理方面的关键挑战, 大模型在大量代码片段上进行无监督学习, 并且拥有很强的代码能力和上下文感知理解能力. 在其他任务上, 大模型展现了优越的性能, 吸引了诸多研究者进一步探究应用大模型到软件缺陷自动修复任务中的效果和方法. 具体而言, 相较于传统技术的关键挑战, 大模型在缺陷修复任务上有以下特性.

(1) 代码理解和生成能力: 预训练的大语言模型, 如 CodeX, 采用了更先进的训练技术和更大的模型规模, 在诸多代码任务上有优异的表现. 面对程序修复任务, 这些模型能够更好地理解代码上下文, 生成更准确和自然的代码, 从而使缺陷修复更加准确和高效.

(2) 补丁搜索空间: 大模型具有强大的代码生成能力, 它们可以在缺陷代码的修复过程中探索更大的搜索空间, 生成更多有效的修复补丁. 这意味着在需要多行编辑修复的缺陷中, 相比仅能进行单行修复的传统工具, 大模型驱动的自动修复技术能找到更好的修复方案.

(3) 修复信息分析: 大模型在训练阶段就已经学习了大量的代码和自然语言数据, 从中学习到了常见代码的合理逻辑和范式. 因此大模型可以使用学习到的代码逻辑和范式, 对缺陷代码进行分析进而尝试找出潜在的缺陷原因并修复.

(4) 上下文理解能力: 在诸多下游任务上, 大模型在处理上下文信息方面表现优异, 能够在代码上下文间的复

杂关系中理解复杂的代码结构和语义. 这使得模型能够考虑到缺陷的上下文信息, 进行更合理的修复.

(5) 自动化和泛用性: 大模型可以对缺陷代码进行自动修复, 不需要人工介入, 大大提高了修复效率. 同时, 由于模型在训练阶段就已经学习了各种编程语言和领域的知识, 因此它们的泛用性更强, 可以应用于更多的修复任务和场景, 而不是仅局限于某一种特定的程序语言.

因此大模型驱动的缺陷自动修复技术成为过去技术面对关键挑战的一个自然的解决方案, 2022 年来研究者便尝试在自动修复系统中应用预训练大语言模型. Xia 等人[32]提出直接应用模型根据漏洞上下文来预测正确代码, 无需对漏洞代码进行分析的见解, 开发了基于代码模型 CodeBERT 直接进行自动修复的工具 AlphaRepair. 实验结果表明, AlphaRepair 可以在 Defects4J 1.2 缺陷数据集上达到优异的修复效果, 修复缺陷数量上优于过去所有传统和基于学习的缺陷自动修复技术, 并且在 Defects4J 2.0 中, AlphaRepair 达到更加优异的修复表现. 这样的结果表明, 大模型驱动的自动修复工具的修复能力和泛化能力优于传统技术, 并且相较基于学习的缺陷自动修复技术, 基于大模型的缺陷修复工具可以有效避免现有基于学习的技术容易过拟合等问题, 进而修复更多漏洞.

在此之后, 随着 OpenAI 公司推出的 CodeX 预训练代码大模型的性能得到工业界和学术界的广泛认可, 研究者开始对应用大模型 (如 CodeX、ChatGPT 和其他开源模型) 到自动缺陷修复任务上进行大量实证研究. 这些工作主要聚焦于: (1) 大模型驱动的自动修复技术修复效果; (2) 如何更好地驱动大模型进行修复 (例如, 微调大模型或者优化提示信息).

在 2023 年, Xia 等人[171]对于大模型驱动的自动修复工具的效果进行了充分的实证研究, 对包含 CodeX 在内的 9 个代码大模型在 5 种不同的数据集上进行比较. 研究发现: (1) 直接应用最新的大模型的修复效果已经大幅度优于所有现有的自动修复工具. 例如, CodeX 模型可以比现有最佳的自动修复技术多修复 32 个漏洞; (2) 模型的规模效应在缺陷自动修复任务上依旧成立: 越大的模型能够修复越多的漏洞; (3) 代码大模型生成的修复补丁比传统自动修复工具生成的补丁更加自然, 并且这样的特性可以被用于补丁排序和正确性检查. 此外该研究通过实验进一步阐明结合大模型和传统技术可以有更好的修复效果, 为未来大模型驱动的自动修复技术提供了新的思路和方向.

由于 CodeX 代码大模型的巨大的影响力和广泛应用, Fan 等人[188]构建了一个基于 LeetCode 的数据集并发现 CodeX 模型生成的代码和程序员编写的代码具有相同的漏洞特征. 他们进一步发现, 由于现有的基于模板和基于学习的自动程序修复技术面对以下挑战: (1) 有限的搜索空间; (2) 无法生成多次编辑的修复; (3) 缺乏对程序依赖性的认识, 导致这些技术仅能修复 CodeX 模型自动生成的代码中的一小部分缺陷. 然而有趣的是给定适当引导的情况下 (例如修复位置), CodeX 模型表现优于基于模板和基于学习的自动程序修复技术. 最后作者尝试将 CodeX 模型和基于模板的技术结合, 发现可以生成更多的有效修复.

Jiang 等人[189]在单行缺陷修复任务上, 分析了代码大模型微调前后的效果, 并选择了 4 个基于学习的技术进行比较. 具体而言, 该工作选择了 PLBART、CodeT5、CodeGen 和 InCoder 等代码大模型以及 4 个最先进的基于深度学习的自动修复工具 CURE、RewardRepair、Recoder、KNOD 在 Defects4J 等数据集上进行比较, 得到了以下发现: (1) 修复任务中表现最好的代码大模型在没有改变任何参数的情况下, 修复漏洞数量比最先进的基于学习的自动程序修复技术多出 72%; (2) 代码大模型针对自动修复任务微调后能够显著提高修复能力. 该实证研究充分表明无需任何其他设置, 直接应用代码大模型的缺陷自动修复工具在修复数量上显著优于基于学习的缺陷修复工具, 并且相较于基于学习的缺陷自动修复技术, 代码大模型只需要在缺陷数据集上简单地微调便可以进一步提升修复能力.

不同于 Jiang 等人[189]的实证研究仅聚焦于单行修复, Huang 等人[45]对于基于代码大模型的微调在多行修复等不同实验设置中进行了详细的研究. 研究者选择了 5 个具有代表性的预训练代码大模型在 3 个不同的程序语言上进行了大量实验. 实验结果表明, 微调后的代码大模型可以显著优于先前的最先进的缺陷自动修复工具. 相较于传统技术只能在单行代码进行修复, 代码大模型可以在单块和多块漏洞修复方面表现出类似的性能. 然而, 对于过于复杂的漏洞, 修复准确率随修复位置的增加而急剧下降. 有趣的是, 不同于 Xia 等人[171]关于大模型规模效应的发现, 在该研究中, 作者发现较小的模型 (UniXcoder) 在修复能力方面可以与较大的模型 (CodeT5) 匹敌甚至超越, 表明对小模型的微调仍需进一步研究挖掘.

根据这些实证研究的发现, 研究者开始尝试提出更好的修复策略以进一步提升大模型在缺陷自动修复上的效果.

在 AlphaRepair 之后, Xia 等人提出了一种通过系统地结合多种微调和提示策略自动进行缺陷修复的工具 FitRepair[24]. 在 CodeT5 模型基础上, FitRepair 使用知识增强微调和修复导向微调生成了两种领域特定的微调模型, 并应用相关标识符提示策略, 通过信息检索和静态分析获取错误行的相关标识符列表以优化提示信息. 在修复过程中, FitRepair 将 4 种模型变体 (包括基础模型、两种微调模型和带有提示的基础模型) 进行结合以便更好地发挥大模型的能力. 在 Defects4J 1.2 和 2.0 缺陷数据集上的评估中, 相较于先前的工作, FitRepair 取得了更好的修复效果.

在之前的诸多研究中, 研究者直接利用大型预训练语言模型对漏洞进行自动修复, 然而这样的方式会导致多次采样生成重复的补丁等问题. 基于此发现, Xia 等人[57]提出基于 ChatGPT 的对话式自动修复工具 ChatRepair. 在修复过程中, ChatRepair 使用最新的对话大模型 ChatGPT 交替进行补丁生成和验证: 首先让大语言模型生成补丁, 然后针对测试集进行验证, 提供反馈并用新的反馈信息提示大模型生成新的补丁. 对于未能通过所有测试的补丁, ChatRepair 将其与相应的测试失败信息结合起来, 构造一个新的提示继续进行迭代. 实验结果表明: ChatRepair 在 Defects4J 1.2 和 2.0 数据集上分别修复了 114 和 48 个漏洞, 达到了目前最佳的修复效果. 并且根据作者统计, 单个漏洞的修复成本平均为 0.42 美元.

5.4 小　结

本节首先简要介绍了自动修复系统的机制, 并聚焦于不同补丁生成技术进行探究. 具体而言, 传统技术中, 基于启发式搜索和约束求解的技术在早期验证了自动修复系统的有效性. 然而, 基于启发式搜索的技术需要面临搜索空间过大及规则依赖人类专家精细设置等挑战, 基于约束求解的技术存在路径爆炸和代码不可读等挑战. 后续研究者尝试通过人工分类缺陷特征定制修复模板, 然而现实中软件缺陷的复杂性导致大部分缺陷特征无法被准确分类, 限制了基于模板技术的发展. 基于此, 研究人员尝试使用深度学习和神经网络相关技术解决之前工作的挑战并获得了更好的修复效果, 但是基于学习的技术所依赖的缺陷修复数据集在数量质量以及收集清洗方式上的困难是该类技术长期面对的难题. 在此之后随着大模型的推出, 研究人员尝试将大模型应用到软件缺陷自动修复任务上并获得了优异的效果. 目前主要的工作集中在将大模型应用到缺陷修复的实证研究中. 进一步, 许多研究者尝试提出定制化的工具和框架以更好地应用模型进行软件缺陷修复.

尽管诸多研究者通过大规模的实证研究已经验证大模型驱动的自动修复工具的有效性, 并尝试微调大模型及调整提示词以达到更好的修复效果. 但是目前研究者尚未对大模型的输入进行更加精细化的构建, 而是将大模型当作一种高效的补丁搜索工具使用. 事实上, 研究者青睐 Defects4J 数据集的原因是该数据集是在真实复杂的 Java 项目上构建而成, 其包含的漏洞拥有超过人工构建缺陷数据集的复杂度. 因此, 该数据集也可以更好地展示大模型在修复复杂漏洞, 尤其是需要多编辑的漏洞时的修复能力. 在模型尝试修复这类复杂缺陷时, 修复的上下文至关重要. 目前 Xia 等人[24]尝试使用缺陷项目对模型微调, 帮助大模型理解缺陷上下文信息, 获得了不俗的修复效果. 进一步, 他们尝试将对话式模型 ChatGPT 引入到自动修复任务中, 在模型对话迭代中理解修复上下文, 同样获得了优异的缺陷修复性能表现. 因此, 如何进一步发掘和利用上下文信息增强模型的修复能力, 还有待研究人员进行更多的探索. 此外, Xia 等人[171]、Jiang 等人[189]和 Huang 等人[45]的大规模实证研究已经表明不同模型的修复性能有显著的区别, 不同模型在微调前后有显著的区别, 以及不同模型在使用不同的输入构造方式时效果也有截然的不同. 那么基于以上几个方面, 研究者仍然可以进行更加深入的探索和研究, 尝试挖掘出不同模型的不同特性, 并且可以进一步有针对性地对模型进行微调和构造提示输入.

6 挑战与机遇

大模型的出现推动了软件工程任务的发展, 但是考虑到现实世界软件应用的复杂性和大模型本身的随机性等因素, 大模型在真实场景的应用仍存在一系列挑战. 下面我们将对本文所涉及的 4 个领域进行分析, 并在此基础上探讨基于大模型的技术在模型快速迭代更新下的挑战与机遇以及在工业实践与落地的未来发展方向.

6.1 深度学习库缺陷检测

使用大模型进行深度学习库缺陷检测最核心的逻辑是大模型在预训练过程中隐含地学习到深度学习库相关

代码的结构和约束, 因此可以高效地自动生成符合深度学习库输入约束的测试代码片段, 并且大模型输出结果的随机性使其成为自然的模糊测试工具. 目前的自动测试框架仅通过简单改变提供给模型的提示词 (prompt) 就可以达到对测试种子变异的目的. 其背后的精妙思想是将大模型作为一个天然的模糊测试器和约束求解器, 完成对深度学习库的测试. 然而, 世上并无万丈高楼平地起, 大模型目前学习的深度学习库代码绝大多数基于 PyTorch 和 TensorFlow 这两个最主流的深度学习库框架, 对于未有如此多训练语料的深度学习库框架 (例如, Caffe[190], MXNet[191]), 其测试能力如何仍值得探究. 此外, 如 Deng 等人[21]在文章中所提到的, 大模型的训练语料大多都是被广泛使用的深度学习库代码片段, 而这些代码被开发人员大量使用并验证, 存在严重缺陷的概率较低. 因而可能导致基于大模型的深度学习库测试效果快速收敛. 那么我们可以认为: 大模型在被广泛使用的深度学习库代码上进行大量训练并学习到了其输入输出关系和约束, 然而这些代码恰恰是深度学习库系统中得到最多人工测试和验证的部分. 而对于那些未被广泛使用的, 可能存在严重缺陷的深度学习库代码和 API, 大模型反而并没有学习到其调用逻辑和约束, 可能无法有效地进行测试. 因此, 我们认为, 将大模型对深度学习库代码的知识从“已知”引申到“未知”, 是未来基于大模型的缺陷挖掘亟待探索的方向. 而大模型展现出的智能和现在模型能力的快速发展, 也为该探索方向提供了一定的信心和保障.

6.2 GUI 自动化测试

尽管 GUI 自动化测试能够节省人工测试所带来的巨大人力成本开销, 但其在测试覆盖率上尚存在限制, 特别是在面对复杂的大型应用程序时, 传统 GUI 自动测试技术难以实现较为全面的探索和测试[103,192]. 与 GUI 自动化测试相比, 手动测试能够发现更多样化和复杂的错误. 因此许多研究表明, 软件开发人员仍更倾向于手动测试应用程序 GUI 页面[106,193,194]. 而大模型强大的能力有望改变过去 GUI 自动化测试落地应用的情况. 具体而言, 目前基于大模型的 GUI 自动化测试技术展现出强大的能力, 在代码覆盖率, 测试效率等各方面较过去技术均有显著提高[100], 并在 GUI 漏洞报告的可信程度以及自动化测试的维护成本上展现出显著的优势. 在过去测试人员需要编写大量的测试脚本将 GUI 自动化测试工具与复杂的应用程序进行对接. 然而现实中移动应用程序往往随着业务需求变更快速迭代更新. 并且, 在应用程序版本的迭代过程中, 应用程序相关功能的 GUI 页面也往往会有显著的变动. 因此对于使用传统技术的 GUI 相关测试人员而言, 在 GUI 页面快速变更的情况下, 持续维护 GUI 自动化测试体系的成本和难度可能超过了手动测试. 因而测试人员往往不愿意使用自动化的 GUI 测试工具和框架. 而基于大模型的 GUI 自动测试技术可以直接使用 GUI 页面的图片作为 GUI 状态信息输入, 无需更改和调试相关接口, 极大缓解了快速版本迭代过程中测试人员的压力. 此外, 虽然传统和基于学习的 GUI 自动测试技术达到了一定的覆盖率并可以挖掘一些 GUI 漏洞, 但是在实际中, 测试人员往往苦恼于 GUI 自动工具的误报和漏报现象. 具体而言, GUI 自动测试工具给出的漏洞缺陷报告在相同测试执行环境下结果不同, 严重降低了 GUI 自动测试的可信度. 而基于大模型的技术可以利用模型的智能自动地对 GUI 页面的状态进行分析, 给出合理的建议, 从而可以很大程度上降低测试人员对 GUI 漏洞验证的人工成本. 综上, 大模型为过去 GUI 自动化测试落地的诸多难题提供了自然的解决方案.

然而, 现有的基于大模型 GUI 自动化测试技术以及相关研究仍然相对较少, 关于如何更好地利用大模型, 使其在 GUI 自动化测试任务上表现出更优秀的性能, 依然存在很大的研究空间. 对于大模型在下游任务的应用, 本文通过横向对比大模型在深度学习库缺陷检测、缺陷自动修复和测试用例自动生成 3 个领域的技术, 进而发现大模型的性能在很大程度上受到模型微调[13,41]以及输入给大模型的提示词质量的影响[104,195]. 因此基于大模型的 GUI 自动化测试技术未来的研究可以在这两个方面进一步深入探索. 首先, 对大模型在 GUI 自动测试任务上进行精细微调, 使模型更好地理解测试任务并与应用程序 GUI 进行交互, 可以进一步提升此类技术的性能表现. 其次, 提示词的构建方式也值得进一步研究. 例如, 研究者可以进一步探索如何根据 GUI 元素的上下文信息以及用户交互方式来设计更精准、具有指导意义的提示词, 并且可以尝试使用应用程序的开发文档或者用户手册进一步提高大模型测试的准确性和实用性. 该方向的探究可能涉及对程序 GUI 设计、用户交互模式以及应用程序的功能目的等方面的深入分析.

值得一提的是, 2023 年 9 月, OpenAI 团队推出了能以视觉图像作为输入的大语言模型 GPT-4V (ision)[196].

GPT-4V 展现了出色的图像理解能力, 能够准确分析用户提供的图像输入, 并以与人类相似的方式应答[197], 这或许十分契合 GUI 自动化测试领域. 此前, 许多基于学习的 GUI 自动化测试技术以被测应用程序的屏幕截图作为输入, 以生成测试输入事件, 这充分展现了以图像输入驱动 GUI 自动化测试的潜力[75]. 通过将应用程序的当前屏幕截图输入 GPT-4V 并根据其生成的操作指引对应用程序进行测试, 基于大模型的工具或许能实现与人类测试员行为较为相似的 GUI 测试, 从而达到较好的 GUI 自动化测试效果. 然而, 目前基于大模型的主要工作使用的都是 OpenAI 公司的闭源模型, 这为公司和企业在实际应用过程中带来了进一步的挑战. 如 QTypist 和 GPTDroid 使用的都是 OpenAI 公司提供的 GPT-3 模型, 并在相关任务场景进行了微调. 但是由于 GPT-3 模型尚未开源, 对于需要本地部署大模型的公司以及无法访问 OpenAI 服务的研究人员而言, 可能需要他们重新在其他开源模型如 Llama2 上进行复现. 然而, 模型的差别是否会影响已有工作 GUI 自动化测试方法的有效性目前尚不可知. 因此, 当下迫切需要研究人员尝试基于开源模型提出一套 GUI 自动化测试的解决方案.

6.3 测试用例自动生成

虽然基于大模型的测试用例生成技术在可读性、覆盖率等指标上均优于以往技术, 但依然面临生成测试的可读性和有效性等方面的挑战. 已有研究人员开始尝试解决这一问题, 例如, CHATTESTER[47]和 ChatUniTest[136]面对 ChatGPT 模型生成的测试用例验证成本高等问题, 尝试引入验证和修复组件, 从而提升大模型生成测试用例的正确性. 然而, 这些工具的实际效果依然受到种种限制. 与此同时, 软件自动修复技术也在尝试应对类似的挑战. 在软件自动修复中, 补丁测试往往占用了最多的时间和算力, 并且在实际的场景中, 构建自动测试补丁的框架的成本通常极其高昂, 如何进一步优化补丁测试框架是值得研究者深入探究的方向. 有趣的是, OpenAI 公司在 2023 年 3 月提出的代码解释器 (code interpreter)[198]可以高效地验证模型生成的代码, 有望解决上述工作缺乏高效的编译器和测试机制来验证生成代码质量的问题. 因此, 能否根据模型的不同下游任务, 设计其对应的代码解释器以进一步提升效率和可靠性, 以及将测试、验证和修复三者的技术进行有机地融合, 是这些领域值得探索的关键问题.

6.4 软件缺陷自动修复

毫无疑问, 基于大模型的自动修复技术达到了历史上最优的修复效率和修复效果, 在生成补丁的可读性和可靠性上都有优异的表现. 在 Xia 等人[171]的研究中发现, 仅使用 CodeX 模型便修复了绝大多数 QuixBugs 缺陷数据集的漏洞. 然而, 真实世界的项目往往非常复杂, 在经过多轮迭代后新旧功能耦合在一起, 导致缺陷往往隐藏在代码非常深的角落, 甚至需要同时修改多个文件中的多个函数才可以将其修复. 目前基于大模型的自动修复技术还主要聚焦于单行或者单块代码修复, 距离在现实中的应用落地仍有遥远的距离. 而两者之间的鸿沟便是: 如何将复杂程序的缺陷信息有效地提供给大模型, 让大模型在理解缺陷场景的条件下, 尝试生成正确的修复 (或者修复建议). 已有的很多工作在缺陷自动修复任务上还仅将大模型作为一种高效的随机补丁搜索器, 而并没有深入挖掘大模型的智能. Xia 等人提出的 ChatRepair[57]在这一方向上已经进行了有价值的探索, 他们使用 ChatGPT 的对话机制迭代地生成补丁和测试, 然后为模型反馈补丁验证信息. 在这个过程中, 模型通过编译和测试结果隐含地学习到了缺陷修复背景, 迭代地优化上一次的错误补丁以进行更高效的修复. 但是, 为模型提供编译信息和测试执行反馈还是非常直接的方式, 是否有更加完整的机制为模型提供修复场景还有待研究人员进一步探索.

6.5 模型快速迭代的挑战与机遇

如同前文对这 4 个领域的分析, 大模型在其中起着至关重要的作用. 当下, 全世界都聚焦于大模型的研究, 模型的训练和开发也日新月异, 其中 OpenAI 公司在 2021 年推出 CodeX 代码大模型后, 经过 1 年 3 个月的时间便推出了 ChatGPT 模型, 并开放给全社会使用. 在此之后仅过去 4 个月时间, 多模态大模型 GPT4 便已经上线. 在软件缺陷自动修复领域, Xia 等人[171]于 2023 年发表的大型实证研究中使用的 code-davinci-002 已经被 OpenAI 废弃[33], 在未来不久, 所有 CodeX 模型都将被 ChatGPT 或 GPT4 模型取代. 短时间来看, 闭源模型的快速迭代为下游任务的实验验证和对其机制特性的进一步研究带来了挑战. 然而对于 GUI 测试领域, GPT4-V 模型的开放显然为研究者提供了新的研究方向和思路. 可以料想到, 在 GPT4-V 模型的帮助下, 基于大模型的 GUI 测试效果能显著超越先前的技术. 此外, 对于测试用例自动生成领域, 其面对的主要挑战之一是 ChatGPT 模型的上下文窗口十分有限.

就在 2023 年 11 月, OpenAI 的开发者大会[199]提出了支持 128k 上下文长度 (300 页长度) 的 GPT4-Turbo 模型, 该模型在很大程度上解决了该领域的主要挑战. 因此我们发现, 目前这些领域很大程度上依赖于模型, 模型的快速迭代既为当前的研究和实验带来了压力, 也为解决领域内的挑战带来了机遇. 此外, 目前很多研究尝试对开源大模型进行训练和微调以获得更优的效果, 然而考虑到缩放定律 (scaling law), 即模型的参数规模越大往往性能越强, 主流开源模型的规模也在逐渐增大. 因此, 在未来尝试对开源模型在软件工程下游任务上进行优化的工作中, 研究者可能需要对比分析不同模型的效果. 然而由于主流开源模型的参数量庞大, 其实验往往需要占用大量 GPU 计算资源并耗费大量的时间, 这可能成为基于开源模型工作的关键挑战之一.

6.6 工业实践与落地

对于使用大模型的技术, 面对复杂的任务和需求, 在工业实践与落地方面仍非常有挑战. 我们将在以下几个方面进行更深入的探讨.

(1) 选择 API 还是 GPU, 尚需讨论

开发人员使用大模型时, 不可避免地涉及公司内部代码数据与模型的交互, 因此, 公司面临核心代码数据泄漏导致资产损失的风险. 对于规模小, 还在起步阶段的公司, 通过使用目前已经产品化的闭源模型 (如 ChatGPT、CodeX、GPT-4 等), 可以以极其低廉的价格调用目前最优的模型. 然而任何公司都存在安全风险, 例如 OpenAI 公司于 2023 年 3 月曾出现泄漏用户对话标题的事件[200]. 因此, 对于业务复杂, 规模庞大的公司来说, 创建自己的大模型技术团队并使用本地运行的私有模型就成为必然的选择. 虽然目前业界主流的开源模型性能依旧不如最新的闭源商业模型, 但是在公司内部, 开发者可以针对公司的代码和任务需求对模型进行训练和微调, 以获得更佳的效果. 例如在自动修复领域, Jiang 等人[189]的研究表明模型在经过缺陷任务数据集微调后性能提升了 31%–1267%, Xia 等人提出的 FitRepair 技术[24]将模型在缺陷项目上进行微调以学习缺陷项目的上下文信息, 进而更好地在项目上下文帮助下对缺陷进行修复.

(2) 实现完全自动化, 尚需深入研究

目前大多工作都强调其技术实现了完全的自动化, 例如目前的缺陷自动修复技术中的 ChatRepair[57]和测试用例生成技术 CHATTESTER[47]. 然而这些基于大模型的工具实际的表现效果和真正工业场景的落地仍有很远的距离. 在 Defects4J 数据集上, ChatRepair 需要多达 20 轮的迭代才可达到比较满意的修复效果, 然而在实际的工业场景中, 让规模巨大的软件迭代运行 20 次测试的时间成本和算力成本非常高昂, 近乎不可能. 此外, CHATTESTER 生成的测试用例也仅有 3 成可以通过执行. 并且这些技术最终生成补丁和测试用例的有效性还必须依赖于人工核验其功能性和完整性. 那么, 对于目前基于大模型的技术而言, 想要完全自动化地完成某一任务仍有非常遥远的距离. 在可预见的未来一段时间, 基于大模型的工业实践落地项目仍需要开发人员的协助和参与, 在实际中更多可能是大模型提升开发人员的效率, 开发人员保障模型输出内容的有效性和安全性.

(3) 将复杂任务拆解为简单的重复任务是可行方案

随着大模型能力的快速发展, 研究者越来越迫切地希望模型能处理复杂的下游任务. 但是在实际中, 模型生成内容具有随机性, 很容易生成一些似是而非的回答或解决方案, 因此无法满足工业场景下对安全性, 稳定性的要求. 然而, 大模型相较于传统技术非常突出的优点是其生成内容的可读性比较好, 容易被开发者接受. 此外, 大模型处理简单重复任务的效率和正确率比较优异. 因此, 在工业场景下完全依赖大模型处理复杂任务目前还尚有一段距离, 但是将复杂任务进行拆分, 使用大模型生成具有初步结构和格式的解决方案以缓解开发者的压力, 是完全可行的思路和方案. 例如使用大模型自动对待测方法生成单元测试[136], 使用大模型对软件缺陷生成初步的修复模板代码[24], 或者自动根据漏洞报告生成测试用例[22]. 总的来说, 目前完全信任大模型让其完成复杂任务可能并非明智之举, 但是可以将复杂任务拆分为重复的简单任务交予大模型处理以帮助开发者提高效率.

7 总 结

本文针对深度学习库缺陷检测、GUI 自动化测试、测试用例自动生成和软件缺陷自动修复这 4 个领域的文

献进行了深入的分析和探究. 在阐述发展脉络的同时, 我们尝试对不同技术流派所面对的挑战进行总结分析. 本文发现, 很多传统技术需要人工专家定制化设计调整策略, 并且其效果容易受到干扰, 有效性受到严峻的挑战. 后续研究者普遍开始尝试应用最新的深度学习和强化学习技术到这些领域中, 并有出色的进展. 然而这些技术所依赖的数据集数量和质量都难以保证. 基于此, 软件工程的研究者迫切需要一种更加强大、更加智能的技术解决复杂程序中的诸多挑战, 而大模型的提出为解决这些挑战带来了新的思路和方向. 诸多研究者将大模型应用到这 4 个领域中, 获得了优异的效果. 但是基于大模型的技术仍面对诸多挑战, 等待研究者进一步探究.

综上, 本文对基于大模型的 4 个软件工程缺陷相关领域进行了详细分析和介绍, 并探讨了基于大模型技术的挑战与机遇. 我们相信在科技创新日新月异的今天, 把握最新技术的动向并应用于软件工程中, 可以进一步推动不同领域的发展, 并为实践中的工业落地提供思路和方向.

References:

[1] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 1219. [doi: 10.1145/3180155.3182526]

[2] Monperrus M. Automatic software repair: A bibliography. ACM Computing Surveys, 2018, 51(1): 17. [doi: 10.1145/3105906]

[3] Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J. Using static analysis to find bugs. IEEE Software, 2008, 25(5): 22–29. [doi: 10.1109/MS.2008.130]

[4] Cole B, Hakim D, Hovemeyer D, Lazarus R, Pugh W, Stephens K. Improving your software using static analysis to find bugs. In: Proc. of the 21st ACM SIGPLAN Symp. on Object-oriented Programming Systems and Applications. Portland: ACM, 2006. 673–674. [doi: 10.1145/1176617.1176667]

[5] Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. In: Proc. of the 22nd ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion. Montreal: ACM, 2007. 815–816. [doi: 10.1145/1297846.1297902]

[6] Fraser G, Arcuri A. EvoSuite: Automatic test suite generation for object-oriented software. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 416–419. [doi: 10.1145/2025113.2025179]

[7] DeMarco F, Xuan JF, Le Berre D, Monperrus M. Automatic repair of buggy if conditions and missing preconditions with SMT. In: Proc. of the 6th Int'l Workshop on Constraints in Software Testing, Verification, and Analysis. Hyderabad: ACM, 2014. 30–39. [doi: 10.1145/2593735.2593740]

[8] Chen LS, Pei Y, Furia CA. Contract-based program repair without the contracts. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 637–647. [doi: 10.1109/ASE.2017.8115674]

[9] Wu YH, Jiang AQ, Li WD, Rabe MN, Staats C, Jamnik M, Szegedy C. Autoformalization with large language models. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2022. 32353–32368.

[10] First E, Rabe MN, Ringer T, Brun Y. Baldur: Whole-proof generation and repair with large language models. arXiv:2303.04910, 2023.

[11] Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). San Francisco: IEEE, 2013. 802–811. [doi: 10.1109/ICSE.2013.6606626]

[12] Hua JR, Zhang MS, Wang KY, Khurshid S. Towards practical program repair with on-demand candidate generation. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 12–23. [doi: 10.1145/3180155.3180245]

[13] Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018. https://hayate-lab.com/wp-content/uploads/2023/05/43372bfa750340059ad87ac8e538c53b.pdf

[14] Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I. Language models are unsupervised multitask learners. 2019. https://insightcivic.s3.us-east-1.amazonaws.com/language-models.pdf

[15] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional Transformers for language understanding. arXiv:1810.04805, 2019.

[16] Liu YH, Ott M, Goyal N, Du JF, Joshi M, Chen DQ, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692, 2019.

[17] ICSE 2023. https://conf.researchr.org/home/icse-2023

[18] ISSTA 2023. https://conf.researchr.org/home/issta-2023

[19] ASE 2023. https://conf.researchr.org/home/ase-2023

[20] ESEC/FSE 2023. https://conf.researchr.org/home/fse-2023

[21] Deng YL, Xia CS, Yang CY, Zhang SD, Yang SJ, Zhang LM. Large language models are edge-case fuzzers: Testing deep learning

libraries via FuzzGPT. arXiv:2304.02014, 2023.

[22] Kang S, Yoon J, Yoo S. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In: Proc. of the 45th Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2312–2323. [doi: 10.1109/ICSE48619.2023.00194]

[23] Liu Z, Chen CY, Wang JJ, Chen MZ, Wu BY, Che X, Wang DD, Wang Q. Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions. arXiv:2310.15780, 2023.

[24] Xia CS, Ding YF, Zhang LM. Revisiting the plastic surgery hypothesis via large language models. arXiv:2303.10494, 2023.

[25] Hou XY, Zhao YJ, Liu Y, Yang Z, Wang KL, Li L, Luo XP, Lo D, Grundy J, Wang HY. Large language models for software engineering: A systematic literature review. arXiv:2308.10620, 2024.

[26] Pan SR, Luo LH, Wang YF, Chen C, Wang JP, Wu XD. Unifying large language models and knowledge graphs: A roadmap. IEEE Trans. on Knowledge and Data Engineering, 2024, 36(7): 3580–3599. [doi: 10.1109/TKDE.2024.3352100]

[27] Yang CY, Deng YL, Lu RY, Yao JY, Liu JW, Jabbarvand R, Zhang LM. WhiteFox: White-box compiler fuzzing empowered by large language models. arXiv:2310.15991, 2024.

[28] Sun ML, Yang YB, Wang Y, Wen M, Jia HX, Zhou YM. SMT solver validation empowered by large pre-trained language models. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 1288–1300. [doi: 10.1109/ASE56229.2023.00180]

[29] Xia CS, Paltenghi M, Le Tian J, Pradel M, Zhang LM. Fuzz4All: Universal fuzzing with large language models. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 126. [doi: 10.1145/3597503.3639121]

[30] Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proc. of the 18th Int'l Conf. on Evaluation and Assessment in Software Engineering. London: ACM, 2014. 38. [doi: 10.1145/2601248.2601268]

[31] Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2665–2690 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.006274]

[32] Xia CS, Zhang LM. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 959–971. [doi: 10.1145/3540250.3549101]

[33] OpenAI platform. 2023. https://platform.openai.com

[34] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, Rodriguez A, Joulin A, Grave E, Lample G. LLaMA: Open and efficient foundation language models. arXiv:2302.13971, 2023.

[35] Chowdhery A, Narang S, Devlin J, *et al*. PaLM: Scaling language modeling with pathways. arXiv:2204.02311, 2022.

[36] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[37] Hoffmann J, Borgeaud S, Mensch A, *et al*. Training compute-optimal large language models. arXiv:2203.15556, 2022.

[38] Shanahan M. Talking about large language models. arXiv:2212.03551, 2023.

[39] Chen M, Tworek J, Jun H, *et al*. Evaluating large language models trained on code. arXiv:2107.03374, 2021.

[40] OpenAI. Introducing ChatGPT. 2023. https://openai.com/blog/chatgpt

[41] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou YQ, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text Transformer. The Journal of Machine Learning Research, 2020, 21(1): 5485–5551.

[42] Lewis M, Liu YH, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, 2020. 7871–7880. [doi: 10.18653/v1/2020.acl-main.703]

[43] Wang WH, Zhang YQ, Sui Y, Wan Y, Zhao Z, Wu J, Yu PS, Xu GD. Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Trans. on Software Engineering, 2022, 48(1): 102–119. [doi: 10.1109/TSE.2020.2979701]

[44] Zeng ZR, Tan HZ, Zhang HT, Li J, Zhang YQ, Zhang LM. An extensive study on pre-trained models for program understanding and generation. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 39–51. [doi: 10.1145/3533767.3534390]

[45] Huang K, Meng XY, Zhang J, Liu Y, Wang WJ, Li SH, Zhang YQ. An empirical study on fine-tuning large language models of code for automated program repair. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 1162–1174. [doi: 10.1109/ASE56229.2023.00181]

[46] Nashid N, Sintaha M, Mesbah A. Retrieval-based prompt selection for code-related few-shot learning. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2450–2462. [doi: 10.1109/ICSE48619.2023.00205]

[47] Yuan ZQ, Lou YL, Liu MW, Ding SJ, Wang KX, Chen YX, Peng X. No more manual tests? Evaluating and improving ChatGPT for unit test generation. arXiv:2305.04207, 2024.

[48] Wei J, Wang XZ, Schuurmans D, Bosma M, Ichter B, Xia F, Chi E, Le QV, Zhou D. Chain-of-thought prompting elicits reasoning in large language models. arXiv:2201.11903, 2023.

[49] Fakhoury S, Chakraborty S, Musuvathi M, Lahiri SK. Towards generating functionally correct code edits from natural language issue descriptions. arXiv:2304.03816, 2023.

[50] Deng YL, Xia CS, Peng HR, Yang CY, Zhang LM. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 423–435. [doi: 10.1145/3597926.3598067]

[51] Prenner JA, Robbes R. Automatic program repair with OpenAI's codex: Evaluating QuixBugs. arXiv:2111.03922, 2021.

[52] Xia CS, Wei YX, Zhang LM. Practical program repair in the era of large pre-trained language models. arXiv:2210.14179, 2022.

[53] Döderlein JB, Acher M, Khelladi DE, Combemale B. Piloting copilot and codex: Hot temperature, cold prompts, or black magic? arXiv:2210.14699, 2023.

[54] Pudari R, Ernst NA. From copilot to pilot: Towards ai supported software development. arXiv:2303.04142, 2023.

[55] Ziegler DM, Stiennon N, Wu J, Brown TB, Radford A, Amodei D, Christiano P, Irving G. Fine-tuning language models from human preferences. arXiv:1909.08593, 2020.

[56] Sobania D, Briesch M, Hanna C, Petke J. An analysis of the automatic bug fixing performance of ChatGPT. arXiv:2301.08653, 2023.

[57] Xia CS, Zhang LM. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385, 2023.

[58] Wang JJ, Huang YC, Chen CY, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. arXiv:2307.07221, 2024.

[59] Sun Y, Chen YH, Wang XG, Tang XO. Deep learning face representation by joint identification-verification. In: Proc. of the 27th Int'l Conf. on Neural Information Processing Systems—Vol. 2. Montreal: MIT Press, 2014. 1988–1996.

[60] Julian KD, Lopez J, Brush JS, Owen MP, Kochenderfer MJ. Policy compression for aircraft collision avoidance systems. In: Proc. of the 35th IEEE/AIAA Digital Avionics Systems Conf. (DASC). Sacramento: IEEE, 2016. 1–10. [doi: 10.1109/DASC.2016.7778091]

[61] Liu SQ, Liu SD, Cai WD, Pujol S, Kikinis R, Feng DG. Early diagnosis of Alzheimer's disease with deep learning. In: Proc. of the 11th IEEE Int'l Symp. on Biomedical Imaging (ISBI). Beijing: IEEE, 2014. 1015–1018. [doi: 10.1109/ISBI.2014.6868045]

[62] Chen CY, Seff A, Kornhauser A, Xiao JX. DeepDriving: Learning affordance for direct perception in autonomous driving. In: Proc. of the 2015 IEEE Int'l Conf. on Computer Vision (ICCV). Santiago: IEEE, 2015. 2722–2730. [doi: 10.1109/ICCV.2015.312]

[63] ABC7.com. Uber gives up testing of self-driving cars in California in wake of fatal Arizona crash. 2018. https://abc7.com/self-driving-uber-crash-video-pedestrian-hit-by-car-autonomous-vehicles/3269690/

[64] Self-driving car. Wikipedia, 2023. https://en.wikipedia.org/wiki/Self-driving_car

[65] Pham HV, Lutellier T, Qi WZ, Tan L. CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 1027–1038. [doi: 10.1109/ICSE.2019.00107]

[66] Wang Z, Yan M, Chen JJ, Liu S, Zhang DD. Deep learning library testing via effective model generation. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 788–799. [doi: 10.1145/3368089.3409761]

[67] Wei AJ, Deng YL, Yang CY, Zhang LM. Free lunch for testing: Fuzzing deep-learning libraries from open source. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 995–1007. [doi: 10.1145/3510003.3510041]

[68] Guo QY, Xie XF, Li Y, Zhang XY, Liu Y, Li XH, Shen C. Audee: Automated testing for deep learning frameworks. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM, 2021. 486–498. [doi: 10.1145/3324884.3416571]

[69] Gu JZ, Luo XC, Zhou YF, Wang X. Muffin: Testing deep learning libraries via neural architecture fuzzing. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1418–1430. [doi: 10.1145/3510003.3510092]

[70] Xie DN, Li YT, Kim M, Pham HV, Tan L, Zhang XY, Godfrey MW. DocTer: Documentation-guided fuzzing for testing deep learning API functions. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 176–188. [doi: 10.1145/3533767.3534220]

[71] Deng YL, Yang CY, Wei AJ, Zhang LM. Fuzzing deep-learning libraries via automated relational API inference. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 44–56. [doi: 10.1145/3540250.3549085]

[72] APP store (apple). Wikipedia, 2023. https://en.wikipedia.org/wiki/App_Store_(Apple)

[73] Google play. Wikipedia, 2023. https://en.wikipedia.org/wiki/Google_Play

[74] Yang Y, Wang X, Zhao CL, Bu ZL. Overview of Android GUI automated testing. Computer Science, 2022, 49(S2): 756–765 (in Chinese with English abstract). [doi: 10.11896/jsjkx.210900231]

[75] Yu SC, Fang CR, Tuo ZY, Zhang QJ, Chen CY, Chen ZY, Su ZD. Vision-based mobile APP GUI testing: A survey. arXiv:2310.13518, 2023.

[76] UI/application exerciser monkey, Android studio. Android developers. 2023. https://developer.android.com/studio/test/other-testing-tools/monkey

[77] Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android APPs. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: ACM, 2013. 224–234. [doi: 10.1145/2491411.2491450]

[78] Hao S, Liu B, Nath S, Halfond WGJ, Govindan R. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile APPs. In: Proc. of the 12th Annual Int'l Conf. on Mobile Systems, Applications, and Services. Bretton: ACM, 2014. 204–217. [doi: 10.1145/2594368.2594390]

[79] Yang W, Prasad MR, Xie T. A grey-box approach for automated GUI-model generation of mobile applications. In: Proc. of the 16th Int' l Conf. on Fundamental Approaches to Software Engineering. Rome: Springer, 2013. 250–265. [doi: 10.1007/978-3-642-37057-1_19]

[80] Choi W, Necula G, Sen K. Guided GUI testing of Android APPs with minimal restart and approximate learning. In: Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications. Indianapolis: ACM, 2013. 623–640. [doi: 10.1145/2509136.2509552]

[81] Mirzaei N, Garcia J, Bagheri H, Sadeghi A, Malek S. Reducing combinatorics in GUI testing of Android applications. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 559–570. [doi: 10.1145/2884781.2884853]

[82] Li YC, Yang ZY, Guo Y, Chen XQ. DroidBot: A lightweight UI-guided test input generator for Android. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering Companion (ICSE-C). Buenos Aires: IEEE, 2017. 23–26. [doi: 10.1109/ICSE-C.2017.8]

[83] Su T, Meng GZ, Chen YT, Wu K, Yang WM, Yao Y, Pu GG, Liu Y, Su ZD. Guided, stochastic model-based GUI testing of Android APPs. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 245–256. [doi: 10.1145/3106237.3106298]

[84] Gibbs sampling. Wikipedia, 2023. https://en.wikipedia.org/wiki/Gibbs_sampling

[85] Gu TX, Sun CN, Ma XX, Cao C, Xu C, Yao Y, Zhang QR, Lu J, Su ZD. Practical GUI testing of Android applications via model abstraction and refinement. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 269–280. [doi: 10.1109/ICSE.2019.00042]

[86] Cai TQ, Zhang Z, Yang P. Fastbot: A multi-agent model-based test generation system. In: Proc. of the 1st IEEE/ACM Int'l Conf. on Automation of Software Test. Seoul: ACM, 2020. 93–96. [doi: 10.1145/3387903.3389308]

[87] Wang J, Jiang YY, Xu C, Cao C, Ma XX, Lu J. ComboDroid: Generating high-quality test inputs for Android APPs via use case combinations. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 469–480. [doi: 10.1145/3377811.3380382]

[88] Liu Z, Chen CY, Wang JJ, Huang YK, Hu J, Wang Q. Guided bug crush: Assist manual GUI testing of Android APPs via hint moves. In: Proc. of the 2022 CHI Conf. on Human Factors in Computing Systems. New Orleans: ACM, 2022. 557. [doi: 10.1145/3491102.3501903]

[89] Anand S, Naik M, Harrold MJ, Yang H. Automated concolic testing of smartphone APPs. In: Proc. of the 20th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering. Cary: ACM, 2012. 59. [doi: 10.1145/2393596.2393666]

[90] Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM. Using GUI ripping for automated testing of Android applications. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 258–261. [doi: 10.1145/2351676.2351717]

[91] Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android APPs. In: Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications. Indianapolis: ACM, 2013. 641–660. [doi: 10.1145/2509136.2509549]

[92] Mahmood R, Mirzaei N, Malek S. EvoDroid: Segmented evolutionary testing of Android APPs. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 599–609. [doi: 10.1145/2635868.2635896]

[93] Mao K, Harman M, Jia Y. Sapienz: Multi-objective automated testing for Android applications. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 94–105. [doi: 10.1145/2931037.2931054]

[94] Dong Z, Böhme M, Cojocaru L, Roychoudhury A. Time-travel testing of Android APPs. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on

Software Engineering. Seoul: ACM, 2020. 481–492. [doi: 10.1145/3377811.3380402]

[95] Li YC, Yang ZY, Guo Y, Chen XQ. Humanoid: A deep learning-based approach to automated black-box Android APP testing. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 1070–1073. [doi: 10.1109/ASE.2019.00104]

[96] Pan MX, Huang A, Wang GX, Zhang T, Li XD. Reinforcement learning based curiosity-driven testing of Android applications. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 153–164. [doi: 10.1145/3395363.3397354]

[97] Peng C, Zhang Z, Lv ZW, Yang P. MUBot: Learning to test large-scale commercial Android APPs like a human. In: Proc. of the 2022 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Limassol: IEEE, 2022. 543–552. [doi: 10.1109/ICSME55016.2022.00074]

[98] Ngiam J, Khosla A, Kim M, Nam J, Lee H, Ng AY. Multimodal deep learning. In: Proc. of the 28th Int'l Conf. on Machine Learning. Bellevue: Omnipress, 2011. 689–696.

[99] YazdaniBanafsheDaragh F, Malek S. Deep GUI: Black-box GUI input generation with deep learning. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2021. 905–916. [doi: 10.1109/ASE51524.2021.9678778]

[100] Liu Z, Chen CY, Wang JJ, Chen MZ, Wu BY, Che X, Wang DD, Wang Q. Chatting with GPT-3 for zero-shot human-like mobile automated GUI testing. arXiv:2305.09434, 2023.

[101] Liu P, Zhang XY, Pistoia M, Zheng YH, Marques M, Zeng LF. Automatic text input generation for mobile testing. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 643–653. [doi: 10.1109/ICSE.2017.65]

[102] Liu Z, Chen CY, Wang JJ, Che X, Huang YK, Hu J, Wang Q. Fill in the blank: Context-aware automated text input generation for mobile GUI testing. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1355–1367. [doi: 10.1109/ICSE48619.2023.00119]

[103] Wu TY, Deng X, Yan J, Zhang J. Analyses for specific defects in Android applications: A survey. Frontiers of Computer Science, 2019, 13(6): 1210–1227. [doi: 10.1007/s11704-018-7008-1]

[104] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1877–1901.

[105] Liu Z, Chen CY, Wang JJ, Chen MZ, Wu BY, Che X, Wang DD, Wang Q. Testing the limits: Unusual text inputs generation for mobile APP crash detection with large language model. arXiv:2310.15657, 2023.

[106] Linares-Vásquez M, Bernal-Cardenas C, Moran K, Poshyvanyk D. How do developers test Android applications? In: Proc. of the 2017 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Shanghai: IEEE, 2017. 613–622. [doi: 10.1109/ICSME.2017.47]

[107] Yoon J, Feldt R, Yoo S. Autonomous large language model agents enabling intent-driven mobile GUI testing. arXiv:2311.08649, 2023.

[108] Nass M, Alegroth E, Feldt R. Improving Web element localization by using a large language model. arXiv:2310.02046, 2023.

[109] Wen H, Wang HM, Liu JX, Li YC. DroidBot-GPT: GPT-powered UI automation for Android. arXiv:2304.07061, 2024.

[110] Almasi MM, Hemmati H, Fraser G, Arcuri A, Benefelds J. An industrial evaluation of unit test generation: Finding real faults in a financial application. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice Track. Buenos Aires: IEEE, 2017. 263–272. [doi: 10.1109/ICSE-SEIP.2017.27]

[111] Shore J, Warden S. The Art of Agile Development. 2nd ed., Sebastopol: O'Reilly Media, 2021.

[112] Beck K. Extreme Programming Explained: Embrace Change. Boston: Addison-Wesley Longman Publishing Co. Inc., 1999.

[113] Daka E, Fraser G. A survey on unit testing practices and problems. In: Proc. of the 25th IEEE Int'l Symp. on Software Reliability Engineering. Naples: IEEE, 2014. 201–211. [doi: 10.1109/ISSRE.2014.11]

[114] Daka E, Campos J, Fraser G, Dorn J, Weimer W. Modeling readability to improve unit tests. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 107–118. [doi: 10.1145/2786805.2786838]

[115] Lin Y, Ong YS, Sun J, Fraser G, Dong JS. Graph-based seed object synthesis for search-based unit testing. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 1068–1080. [doi: 10.1145/3468264.3468619]

[116] Fraser G, Arcuri A. Sound empirical evidence in software testing. In: Proc. of the 34th Int'l Conf. on Software Engineering. Zurich: IEEE, 2012. 178–188. [doi: 10.1109/ICSE.2012.6227195]

[117] Ernst MD. Randoop: Automatic unit test generation for Java. 2023. https://randoop.github.io/randoop/

[118] Selakovic M, Pradel M, Karim R, Tip F. Test generation for higher-order functions in dynamic languages. Proc. of the ACM on Programming Languages, 2018, 2: 161. [doi: 10.1145/3276531]

[119] Arteca E, Harner S, Pradel M, Tip F. Nessie: Automatically testing JavaScript APIs with asynchronous callbacks. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1494–1505. [doi: 10.1145/3510003.3510106]

[120] Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C. The Daikon system for dynamic detection of likely

invariants. Science of Computer Programming, 2007, 69(1–3): 35–45. [doi: 10.1016/j.scico.2007.01.015]

[121] Csallner C, Tillmann N, Smaragdakis Y. DySy: Dynamic symbolic execution for invariant inference. In: Proc. of the 30th Int'l Conf. on Software Engineering. Leipzig: ACM, 2008. 281–290. [doi: 10.1145/1368088.1368127]

[122] Molina F, Ponzio P, Aguirre N, Frias M. EvoSpex: An evolutionary algorithm for learning postconditions. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 1223–1235. [doi: 10.1109/ICSE43902.2021.00112]

[123] Palomba F, Di Nucci D, Panichella A, Oliveto R, De Lucia A. On the diffusion of test smells in automatically generated test code: An empirical study. In: Proc. of the 9th Int'l Workshop on Search-based Software Testing. Austin: ACM, 2016. 5–14. [doi: 10.1145/2897010.2897016]

[124] Watson C, Tufano M, Moran K, Bavota G, Poshyvanyk D. On learning meaningful assert statements for unit test cases. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1398–1409. [doi: 10.1145/3377811.3380429]

[125] Mastropaolo A, Scalabrino S, Cooper N, Palacio DN, Poshyvanyk D, Oliveto R, Bavota G. Studying the usage of text-to-text transfer Transformer to support code-related tasks. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 336–347. [doi: 10.1109/ICSE43902.2021.00041]

[126] Mastropaolo A, Cooper N, Palacio DN, Scalabrino S, Poshyvanyk D, Oliveto R, Bavota G. Using transfer learning for code-related tasks. IEEE Trans. on Software Engineering, 2023, 49(4): 1580–1598. [doi: 10.1109/TSE.2022.3183297]

[127] Yu H, Lou YL, Sun K, Ran DZ, Xie T, Hao D, Li Y, Li G, Wang QX. Automated assertion generation via information retrieval and its integration with deep learning. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 163–174. [doi: 10.1145/3510003.3510149]

[128] Nie PY, Banerjee R, Li JJ, Mooney RJ, Gligoric M. Learning deep semantics for test completion. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2111–2123. [doi: 10.1109/ICSE48619.2023.00178]

[129] Tufano M, Drain D, Svyatkovskiy A, Sundaresan N. Generating accurate assert statements for unit test cases using pretrained Transformers. In: Proc. of the 3rd ACM/IEEE Int'l Conf. on Automation of Software Test. Pittsburgh: ACM, 2022. 54–64. [doi: 10.1145/3524481.3527220]

[130] Dinella E, Ryan G, Mytkowicz T, Lahiri SK. TOGA: A neural method for test oracle generation. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 2130–2141. [doi: 10.1145/3510003.3510141]

[131] Liu ZX, Liu K, Xia X, Yang XH. Towards more realistic evaluation for neural test oracle generation. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 589–600. [doi: 10.1145/3597926.3598080]

[132] Tufano M, Drain D, Svyatkovskiy A, Deng SK, Sundaresan N. Unit test case generation with Transformers and focal context. aarXiv:2009.05617, 2021.

[133] Panichella A, Panichella S, Fraser G, Sawant AA, Hellendoorn VJ. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In: Proc. of the 2020 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Adelaide: IEEE, 2020. 523–533. [doi: 10.1109/ICSME46990.2020.00056]

[134] Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 919–931. [doi: 10.1109/ICSE48619.2023.00085]

[135] Schäfer M, Nadi S, Eghbali A, Tip F. An empirical evaluation of using large language models for automated unit test generation. arXiv:2302.06527, 2023.

[136] Xie ZK, Chen YH, Zhi C, Deng SG, Yin JW. ChatUniTest: A ChatGPT-based automated unit test generation tool. arXiv:2305.04764, 2024.

[137] Chen B, Zhang FJ, Nguyen A, Zan DG, Lin ZQ, Lou JG, Chen WZ. CodeT: Code generation with generated tests. arXiv:2207.10397, 2022.

[138] Lahiri SK, Fakhoury S, Naik A, Sakkas G, Chakraborty S, Musuvathi M, Choudhury P, Von Veh C, Inala JP, Wang CL, Gao JF. Interactive code generation via test-driven user-intent formalization. arXiv:2208.05950, 2023.

[139] Mankowitz DJ, Michi A, Zhernov A, *et al*. Faster sorting algorithms discovered using deep reinforcement learning. Nature, 2023, 618(7964): 257–263. [doi: 10.1038/s41586-023-06004-9]

[140] GPT-4 "discovered" the same sorting algorithm as alphadev by removing "mov s p" | hacker news, 2024. https://news.ycombinator.com/item?id=36247549

[141] The New York Times. A smarter APP is watching your wallet. 2023. https://www.nytimes.com/2021/03/09/business/apps-personal-finance-budget.html

[142] Webster RW, Hess D. A real-time software controller for a digital model railroad system. In: Proc. of the 1993 IEEE Workshop on Real-time Applications. New York: IEEE, 1993. 126–130. [doi: 10.1109/RTA.1993.263102]

[143] Brown D. Hospitals turn to artificial intelligence to help with an age-old problem: Doctors' poor bedside manners. Washington Post, 2021. https://www.washingtonpost.com/technology/2021/02/16/virtual-ai-hospital-patients

[144] Liblit B, Aiken A, Zheng AX, Jordan MI. Bug isolation via remote program sampling. ACM SIGPLAN Notices, 2003, 38(5): 141–154. [doi: 10.1145/780822.781148]

[145] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 437–440. [doi: 10.1145/2610384.2628055]

[146] Lin D, Koppel J, Chen A, Solar-Lezama A. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: Proc. of the 2017 ACM SIGPLAN Int'l Conf. on Systems, Programming, Languages, and Applications: Software for Humanity. Vancouver: ACM, 2017. 55–56. [doi: 10.1145/3135932.3135941]

[147] Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans. on Software Engineering, 2015, 41(12): 1236–1256. [doi: 10.1109/TSE.2015.2454513]

[148] Mao XG, Lei Y, Dai ZY, Qi YH, Wang CS. Slice-based statistical fault localization. Journal of Systems and Software, 2014, 89: 51–62. [doi: 10.1016/j.jss.2013.08.031]

[149] Eric Wong W, Debroy V, Choi B. A family of code coverage-based heuristics for effective fault localization. Journal of Systems and Software, 2010, 83(2): 188–208. [doi: 10.1016/j.jss.2009.09.037]

[150] Abreu R, Zoeteweij P, van Gemund AJC. Spectrum-based multiple fault localization. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. Auckland: IEEE, 2009. 88–99. [doi: 10.1109/ASE.2009.25]

[151] Perez A, Abreu R, van Deursen A. A test-suite diagnosability metric for spectrum-based fault localization approaches. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 654–664. [doi: 10.1109/ICSE.2017.66]

[152] Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable statistical bug isolation. ACM SIGPLAN Notices, 2005, 40(6): 15–26. [doi: 10.1145/1064978.1065014]

[153] Liu C, Fei L, Yan XF, Han JW, Midkiff SP. Statistical debugging: A hypothesis testing-based approach. IEEE Trans. on Software Engineering, 2006, 32(10): 831–848. [doi: 10.1109/TSE.2006.105]

[154] Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. Long Beach: ACM, 2005. 273–282. [doi: 10.1145/1101908.1101949]

[155] Abreu R, Zoeteweij P, Golsteijn R, van Gemund AJC. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 2009, 82(11): 1780–1792. [doi: 10.1016/j.jss.2009.06.035]

[156] Campos J, Riboira A, Perez A, Abreu R. GZoltar: An eclipse plug-in for testing and debugging. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 378–381. [doi: 10.1145/2351676.2351752]

[157] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B. Evaluating and improving fault localization. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 609–620. [doi: 10.1109/ICSE.2017.62]

[158] Steimann F, Frenkel M, Abreu R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. Lugano: ACM, 2013. 314–324. [doi: 10.1145/2483760.2483767]

[159] Xie XY, Chen TY, Kuo FC, Xu BW. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. on Software Engineering and Methodology, 2013, 22(4): 31. [doi: 10.1145/2522920.2522924]

[160] Xuan JF, Monperrus M. Learning to combine multiple ranking metrics for fault localization. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. Victoria: IEEE, 2014. 191–200. [doi: 10.1109/ICSME.2014.41]

[161] Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Le Traon Y. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification (ICST). Xi'an: IEEE, 2019. 102–113. [doi: 10.1109/ICST.2019.00020]

[162] Xiong YF, Wang J, Yan RF, Zhang JC, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 416–426. [doi: 10.1109/ICSE.2017.45]

[163] Zhang XY, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proc. of the 28th Int'l Conf. on Software Engineering. Shanghai: ACM, 2006. 272–281. [doi: 10.1145/1134285.1134324]

[164] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 298–309. [doi: 10.1145/3213846.3213871]

[165] Xuan JF, Monperrus M. Test case purification for improving fault localization. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 52–63. [doi: 10.1145/2635868.2635906]

[166] Yang AZH, Le Goues C, Martins R, Hellendoorn V. Large language models for test-free fault localization. In: Proc. of the 46th

IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 17. [doi: 10.1145/3597503.3623342]

[167] Wong WE, Gao RZ, Li YH, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016, 42(8): 707–740. [doi: 10.1109/TSE.2016.2521368]

[168] Zakari A, Lee SP, Abreu R, Ahmed BH, Rasheed RA. Multiple fault localization of software programs: A systematic literature review. Information and Software Technology, 2020, 124: 106312. [doi: 10.1016/j.infsof.2020.106312]

[169] De Souza HA, Chaim ML, Kon F. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. arXiv:1607.04347, 2017.

[170] Liu K, Koyuncu A, Kim D, Bissyandé TF. TBar: Revisiting template-based automated program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 31–42. [doi: 10.1145/3293882.3330577]

[171] Xia CS, Wei YX, Zhang LM. Automated program repair in the era of large pre-trained language models. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1482–1494. [doi: 10.1109/ICSE48619.2023.00129]

[172] Ghanbari A, Benton S, Zhang LM. Practical program repair via bytecode mutation. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 19–30. [doi: 10.1145/3293882.3330559]

[173] Liu BB, Dong W, Wang J. Survey on intelligent search and construction methods of program. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2180–2197 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.005529]

[174] Le Goues C, Nguyen T, Forrest S, Weimer W. GenProg: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2012, 38(1): 54–72. [doi: 10.1109/TSE.2011.104]

[175] Wen M, Chen JJ, Wu RX, Hao D, Cheung SC. Context-aware patch generation for better automated program repair. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 1–11. [doi: 10.1145/3180155.3180233]

[176] Nguyen HDT, Qi DW, Roychoudhury A, Chandra S. SemFix: Program repair via semantic analysis. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). San Francisco: IEEE, 2013. 772–781. [doi: 10.1109/ICSE.2013.6606623]

[177] Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proc. of the 24th Int'l Conf. on Software Engineering. Orlando: ACM, 2002. 467–477. [doi: 10.1145/581339.581397]

[178] Lutellier T, Pham HV, Pang L, Li YT, Wei MS, Tan L. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 101–114. [doi: 10.1145/3395363.3397369]

[179] Jiang N, Lutellier T, Tan L. CURE: Code-aware neural machine translation for automatic program repair. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 1161–1173. [doi: 10.1109/ICSE43902.2021.00107]

[180] Zhu QH, Sun ZY, Xiao YA, Zhang WJ, Yuan K, Xiong YF, Zhang L. A syntax-guided edit decoder for neural program repair. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 341–353. [doi: 10.1145/3468264.3468544]

[181] Ye H, Martinez M, Monperrus M. Neural program repair with execution-based backpropagation. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1506–1518. [doi: 10.1145/3510003.3510222]

[182] Jiang N, Lutellier T, Lou YL, Tan L, Goldwasser D, Zhang XY. KNOD: Domain knowledge distilled tree decoder for automated program repair. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1251–1263. [doi: 10.1109/ICSE48619.2023.00111]

[183] Yue RR, Meng N, Wang QX. A characterization study of repeated bug fixes. In: Proc. of the 2017 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Shanghai: IEEE, 2017. 422–432. [doi: 10.1109/ICSME.2017.16]

[184] Dallmeier V, Zimmermann T. Extraction of bug localization benchmarks from history. In: Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Atlanta: ACM, 2007. 433–436. [doi: 10.1145/1321631.1321702]

[185] Jiang JJ, Ren LY, Xiong YF, Zhang LM. Inferring program transformations from singular examples via big code. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 255–266. [doi: 10.1109/ASE.2019.00033]

[186] Jiang YJ, Liu H, Niu N, Zhang L, Hu YM. Extracting concise bug-fixing patches from human-written patches in version control systems. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 686–698. [doi: 10.1109/ICSE43902.2021.00069]

[187] Li Y, Wang SH, Nguyen TN. DLFix: Context-based code transformation learning for automated program repair. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 602–614. [doi: 10.1145/3377811.3380345]

[188] Fan ZY, Gao X, Mirchev M, Roychoudhury A, Tan SH. Automated repair of programs from large language models. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1469–1481. [doi: 10.1109/ICSE48619.2023.00128]

[189] Jiang N, Liu K, Lutellier T, Tan L. Impact of code language models on automated program repair. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1430–1442. [doi: 10.1109/ICSE48619.2023.00125]

[190] Jia YQ, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. In: Proc. of the 22nd ACM Int'l Conf. on Multimedia. Orlando: ACM, 2014. 675–678. [doi: 10.1145/2647868.2654889]

[191] Chen TQ, Li M, Li YT, Lin M, Wang NY, Wang MJ, Xiao TJ, Xu B, Zhang CY, Zhang Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv:1512.01274, 2015.

[192] Wiklund K, Eldh S, Sundmark D, Lundqvist K. Impediments for software test automation: A systematic literature review. Software Testing, Verification and Reliability, 2017, 27(8): e1639. [doi: 10.1002/stvr.1639]

[193] Garcia SE. Usability testing: Creative techniques for answering your research questions. In: Proc. of the Extended Abstracts of the 2020 CHI Conf. on Human Factors in Computing Systems. Honolulu: ACM, 2020. 1–2. [doi: 10.1145/3334480.3375064]

[194] Haas R, Elsner D, Juergens E, Pretschner A, Apel S. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 1281–1291. [doi: 10.1145/3468264.3473922]

[195] Petroni F, Rocktäschel T, Riedel S, Lewis P, Bakhtin A, Wu YX, Miller A. Language models as knowledge bases? In: Proc. of the 2019 Conf. on Empirical Methods in Natural Language Processing and the 9th Int'l Joint Conf. on Natural Language Processing (EMNLP-IJCNLP). Hong Kong: Association for Computational Linguistics, 2019. 2463–2473. [doi: 10.18653/v1/D19-1250]

[196] OpenAI. GPT-4 technical report. arXiv:2303.08774, 2024.

[197] Yang ZY, Li LJ, Lin K, Wang JF, Lin CC, Liu ZC, Wang LJ. The dawn of LMMs: Preliminary explorations with GPT-4V(ision). arXiv:2309.17421, 2023.

[198] ChatGPT plugins. 2023. https://openai.com/blog/chatgpt-plugins

[199] GPT-4 turbo. QpenAI help center. 2023. https://help.openai.com/en/articles/8555510-gpt-4-turbo

[200] Browne R. OpenAI CEO admits a bug allowed some ChatGPT users to see others' conversation titles. 2023. https://www.cnbc.com/2023/03/23/openai-ceo-says-a-bug-allowed-some-chatgpt-to-see-others-chat-titles.html

附中文参考文献:

[31] 姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述. 软件学报, 2021, 32(9): 2665–2690. [doi: 10.13328/j.cnki.jos.006274]

[74] 杨艺, 王嬉, 赵春蕾, 步志亮. Android GUI 自动化测试综述. 计算机科学, 2022, 49(S2): 756–765. [doi: 10.11896/jsjkx.210900231]

[173] 刘斌斌, 董威, 王戟. 智能化的程序搜索与构造方法综述. 软件学报, 2018, 29(8): 2180–2197. [doi: 10.13328/j.cnki.jos.005529]

香佳宏(1999－), 男, 硕士, CCF 学生会员, 主要研究领域为大模型驱动的自动程序修复, 模糊测试.

彭湃(1977－), 男, 本科, 主要研究领域为软件测试, 系统及软件工程.

徐霄阳(2003－), 男, 本科生, CCF 学生会员, 主要研究领域为大模型驱动的自动程序修复.

张钊(1976－), 男, 本科, 主要研究领域为系统及软件工程, 软件测试, 研发质量管理.

孔繁初(2003－), 男, 本科生, CCF 学生会员, 主要研究领域为大模型驱动的自动程序修复.

张煜群(1986－), 男, 博士, 助理教授, 博士生导师, CCF 专业会员, 主要研究领域为模糊测试, 软件成分分析, 污点分析, 基于大模型的软件工程.