计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.009757] [CSTR: 32024.14.csa.009757]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 Happens-Before 关系的 Android 应用并发不稳定测试检测①

张　雨[1], 张文天[2], 张未晞[2], 尚　颖[2]

[1](长春职业技术学院 信息学院, 长春 130033)
[2](北京化工大学 信息科学与技术学院, 北京 100029)
通信作者: 尚　颖, E-mail: shangy@mail.buct.edu.cn

摘　要: Android 应用异步消息执行顺序的不确定性是导致其不稳定的主要原因. 现有不稳定测试研究大多通过随机确定异步消息的执行顺序以触发不稳定测试, 其检测效果不佳且效率较低. 本文提出一种基于 Happens-Before (HB) 关系的 Android 应用并发不稳定测试检测方法, 通过分析 Android 应用测试用例执行轨迹中异步消息间的 HB 关系, 进而确定异步消息的可执行区间; 并设计最大差异化调度策略, 有指导性地确定异步消息执行顺序, 使调度后的测试执行轨迹上异步消息执行序与原测试执行轨迹差异最大化, 进而尝试改变测试执行结果, 检测测试的不稳定性. 为验证本文方法的有效性, 针对 40 个 Android 应用程序的 50 个不稳定测试用例进行实验, 实验结果表明, 本文方法可检测全部不稳定测试用例, 相比当前主流工具检测效果提升 6%, 且平均检测时间缩短 31.78%.

关键词: Android 应用; 不稳定测试检测; 异步消息; HB 关系; 差异化调度

引用格式: 张雨,张文天,张未晞,尚颖.基于 Happens-Before 关系的 Android 应用并发不稳定测试检测.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9757.html

Flaky Test Detection Based on Happens-Before Relationship for Android Applications

ZHANG Yu[1], ZHANG Wen-Tian[2], ZHANG Wei-Xi[2], SHANG Ying[2]

[1](School of Information, Changchun Polytechnic, Changchun 130033, China)
[2](College Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: The uncertain execution order of asynchronous messages in Android applications is the main reason for their flakiness. Most existing flaky test studies trigger instability testing by randomly determining the execution order of asynchronous messages, which is ineffective and inefficient. This study proposes a concurrent flaky test detection based on the Happens-Before (HB) relationship for Android applications. After analyzing the HB relationship between asynchronous messages in the execution trace of Android application test cases, the proposed method determines the asynchronous message workscope. Then, it designs a scheduling strategy with maximum differentiation to determine the asynchronous message execution order under guidance to maximize the difference between the asynchronous message execution order and the original test execution trace on the test execution trace after scheduling. Then, the method tries to change test execution results to detect flakiness in the test. For effectiveness verification of the method, experiments are conducted on 50 test cases of 40 Android applications, and the experimental results show that the method can detect all the flaky tests, improving the detection effect by 6% and shortening the average detection time by 31.78% compared with the current state-of-the-art techniques.

Key words: Android application; flaky test detection; asynchronous message; happens-before (HB) relationship; differentiated scheduling

① 基金项目: 国家自然科学基金 (62077003)
收稿时间: 2024-07-11; 修改时间: 2024-08-01; 采用时间: 2024-08-20; csa 在线出版时间: 2024-11-28

1 引言

不稳定测试 (flaky test) 是指测试用例在同一环境下运行时, 不确定性地通过或失败[1–3]. 不稳定测试的存在使得测试人员无法依赖测试结果来准确判断软件是否存在缺陷, 严重影响了开发效率[4–6] .

谷歌在 2017 年发布的一项调查显示: 他们的 420 万次测试中约有 6.3 万次测试是不稳定的. 他们将测试分为 3 种规模: 小型、中型和大型, 其中大型测试中有 14% 存在不稳定问题[7]. 微软对 5 个软件项目为期 1 个月的监控发现, 4.6% 测试用例是不稳定的; 同时对 58 名开发人员进行调查表明, 不稳定测试是影响软件部署效率的第 2 大原因[8]. Eck 等人[9]对 Mozilla 公司的 21 位专业开发人员进行了调研, 他们认为 26% 不稳定测试是由于并发导致的, 并且开发人员解决该问题所需的时间是平均时间的 4 倍之多. 不稳定测试目前已成为软件产业界和学术界关注的重要研究课题. 亟需研究高效的不稳定测试检测方法以提高测试的可靠性和准确性.

Android 应用的并发编程模式将处理 UI 事件的主线程和执行复杂任务的后台线程相混合[10], 用以提高 Android 应用的易用性及交互性. 虽然这种混合模型允许开发人员平衡 Android 应用的响应能力和性能, 但它也会使测试执行因异步事件执行的不确定性而导致测试不稳定[11].

Android 应用的并发不稳定测试尚未有高效的检测方法报道, 目前主要有两类方法, 随机探索和指导性探索. 随机探索意指测试用例重复运行, 例如王潇等人[12]通过重复多次 (例如 100 次) 执行测试用例, 随机确定异步消息执行顺序. Zhen 等人 [13]通过在一定的测试执行次数内重复运行测试用例 (比如 50 次、100 次), 判断测试执行过程中结果是否发生了改变. 如果其中有任意两次测试结果不同, 即一次为成功, 一次为失败, 则该测试为不稳定测试. 这类不稳定测试检测方法较为简单, 只需指定次数并重复执行测试用例, 但检测时间成本过高且检测效果及效率不佳; 另一部分研究者提出通过扰动测试执行环境, 动态确定异步消息的可能执行顺序[14–17]. 但探索过程随机性很大, 不稳定测试检测的效果及效率不佳. 指导性探索是指有指导性地确定异步消息的执行序, 例如 Zhen 等人[18]通过分析测试过程中所有异步消息的调度空间, 在空间内根据测试语句确定异步消息的调度位置, 以指导异步消息执行序的探索, 但由于它在确定调度空间时采取的策略没有考虑消息间可能存在的依赖关系, 调度空间不够准确, 导致其检测效果不佳且效率较低.

HB 关系是一种用于定义事件间执行顺序的关系, 可用于辅助确定事件的执行域, 为事件执行顺序的探索提供精准指导. 针对 Android 应用事件间的 HB 关系分析已有初步研究, 例如: Hsiao 等人[19]针对 Android 数据竞争检测问题, 提出消息间的 HB 关系规则. Wu 等人[20]提出了 SARD, 在程序中静态分析两个事件之间的依赖关系, 从而识别精确的 HB 关系. Salehnamadi 等人[21]提出了 ER Catcher, 通过分析 Android 应用程序的事件间的 HB 关系识别潜在的事件竞争. Hu 等人[22]提出了一种静态检测方法, 通过分析应用程序的事件处理逻辑和事件间的 HB 关系, 识别可能导致竞争的情况. 但上述 HB 关系的分析大多是通过静态分析应用程序代码中事件回调函数之间存在的数据依赖、控制依赖, 从而确定事件回调函数之间的 HB 关系进而推断出事件以及消息之间的 HB 关系, 这些方法并没有针对消息本身相互间存在的依赖关系进行分析, 不仅分析时间成本高而且无法准确地分析出消息间存在的所有 HB 关系. 并且 Android 不稳定测试的研究重点在于异步消息执行序的确定, 因此只需分析出所有与异步消息相关的 HB 关系.

因此, 本文针对 Android 应用中的不稳定测试问题, 研究一种基于 HB 关系的检测方法. 该方法采取指导性探索策略, 依据 Android 线程通信机制中的消息通信原理, 归纳整理一套针对消息对象的 HB 关系规则; 通过分析测试用例执行轨迹上消息间的 HB 关系, 构建其消息间的 HB 关系集, 以更准确地识别每个异步消息的调度空间, 指导后续异步消息执行序的确定; 同时, 为尽快触发测试用例中潜在的不稳定情况, 本文基于 HB 关系探讨一种最大差异化调度策略, 探索每个异步消息在其调度空间内的最大差异调度位置, 有指导性地确定异步消息的执行顺序, 使调度后的测试执行轨迹与原轨迹在异步消息执行序上的差异最大化, 以在测试用例的再次执行过程中改变测试执行结果, 触发测试的不稳定, 有效检测不稳定测试. 本文的贡献如下.

1) 提出异步消息可执行区间的判定方法. 给出 Android 应用测试执行过程中消息对象间的 HB 关系定义, 并基于 Android 应用测试执行轨迹构建其消息

间 HB 关系集, 进而识别异步消息的可执行区间.

2) 提出差异最大化调度策略, 指导生成可能改变测试执行结果的异步消息执行序列, 以触发测试不稳定.

3) 设计实验针对 40 个应用程序的 50 个不稳定测试进行检测. 结果表明本文方法成功检测到了 50 个全部不稳定测试, 相比于目前主流工具 FlakeScanner[18] 检测率提高了 6%, 且平均检测时间缩短了 31.78%.

2 相关工作

2.1 Android 应用测试

Android 应用测试主要涉及对应用的功能、性能、兼容性等进行测试, 旨在确保应用程序在不同设备和场景下的稳定性、功能正确性. Android 应用测试通常在物理设备或模拟器上运行, 通过调用系统中的 API 来控制测试中的应用程序. 这些测试通常在一个单独的线程中执行以模拟用户交互, 该线程也称为测试线程. Android 系统支持多种测试框架以帮助开发人员自动化测试, 如 Robotium[23]、Espresso[24]等, 能够模拟各种操作手势, 并对各种控件进行操作. 例如 Robotium 提供了丰富的自动化功能, 其 API 可用于编写 UI 测试用例, 允许开发人员模拟用户在应用程序上的复杂操作, 如点击按钮、输入文本, 可进行断言验证、同时具备灵活的异步等待机制, 允许测试线程等待异步操作的完成, 确保测试在合适的时机继续执行.

Android 应用程序用户界面 (UI) 自动化测试通常应用特定的工具或测试框架编写测试用例脚本、UI 测试用例, 用于模拟用户在应用程序上的操作, 如点击按钮、输入文本、滑动屏幕等, 其主要目的是确保应用程序的用户界面在不同设备和环境下的稳定性和一致性.

Android 测试用例由一系列测试语句组成, 这些测试语句通常包括对 Android 组件的操作、应用程序方法的调用、断言等. 可以表示为 tc=<S_1, S_2,···, S_n>, 其中 tc 表示测试用例, S_i 表示测试语句. 一条测试语句的执行会生成若干个消息 (message). 因此, 测试用例执行轨迹 Trace 可表示为测试执行过程中产生的所有消息的生命周期方法执行序列.

2.2 Android 线程通信机制

Android 应用程序响应用户操作或执行其他任务的过程中, 涉及线程之间的通信, 以实现线程间的相互协作, 从而顺利完成任务. Android 线程主要分为 UI 线程和异步线程, UI 线程又称为主线程, 负责处理与用户界面相关的操作, 包括响应用户输入、处理 UI 事件等. 异步线程是指在后台执行的工作线程, 用于处理耗时操作, 以避免在主线程中执行耗时任务而导致 UI 被阻塞、卡顿或失去响应. UI 线程和异步线程间的通信由 Android 通信机制负责, 通过在不同线程之间发送和接收, 实现线程间的数据传递和协作. Android 通信机制主要包括: (1) message (消息): 包含线程间传递的数据, 分为两类, 从异步线程发往 UI 线程的称为异步 message, 异步线程间发送和 UI 线程发送至异步线程的称为同步 message. 为方便区分, 本文将 M 表示同步 message (如图 1 中的绿色圆框), m 表示异步 message (如图 1 中的灰色圆框). message 在执行过程中分为 *send* (发送) 和 *dispatch* (处理) 两个生命周期阶段. (2) Handler (处理器): 每个 Handler 都与一个线程 t 关联, 用于发送 message 到其他线程以及处理线程 t 接收的 message. (3) Looper (循环器): 当线程接收到 message 后会将其放入消息队列中, Looper 负责从消息队列中不断取出 message 交给 Handler 处理, 每个线程只能有一个 Looper 实例.

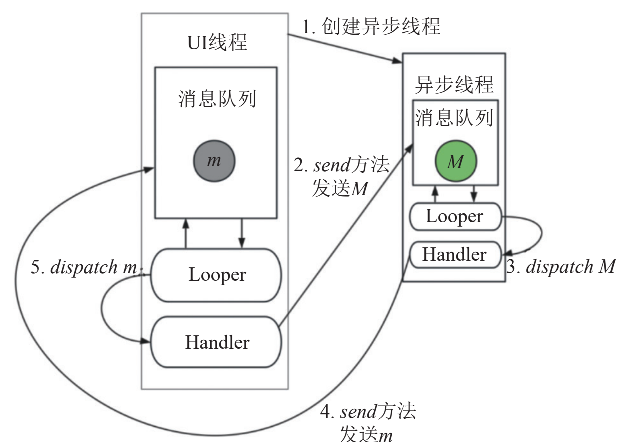

图 1 Android 线程通信机制

线程间具体通信流程如图 1 所示, UI 线程遇到文件 I/O、网络请求等耗时任务时会首先创建一个异步线程 (第 1 步), 由于某些耗时任务需要立即执行, UI 线程会通过 Handler 生成同步 message M (图中绿色圆圈) 并调用 *send* 方法将 M 发送至异步线程 (第 2 步), 通知异步线程执行耗时任务, 异步线程接收到 M 后通过 Looper 从消息队列中取出 M 并 *dispatch* 至自身的 Handler 处理 (第 3 步), 然后开始执行耗时任务. 异步

线程执行完耗时任务后会通过 Handler 生成异步 message *m* (图中灰色圆圈) 并将执行结果放入 *m* 当中, 然后通过 *send* 方法将 *m* 发送给 UI 线程的消息队列 (第 4 步). 最后, UI 线程的 Looper 会从消息队列中取出 *m* 并 *dispatch* 至自身的 Handler 处理 (第 5 步), 获取耗时任务的执行结果.

3 基于 HB 关系的不稳定测试检测

在 Android 应用的测试执行过程中, 有多个异步 message 负责将异步任务的执行结果发送给 UI 线程, 但是异步 message 到达 UI 线程的顺序可能因异步任务执行时间的不确定性而发生改变, 导致测试结果发生变化, 进而导致测试不稳定. 因此通过分析测试执行过程中 message 间的固定执行序能得出 message 间的 HB 关系, 而每个存在 HB 关系的异步 message 都有其独立的调度空间, 通过在调度空间内最大程度上改变异步 message 的执行序有助于检测不稳定测试.

3.1 不稳定测试检测总体框架

本文首先定义规则描述 message 间 HB 关系, 并根据 HB 关系分析异步 message 的可执行区间, 进而提出最大差异化调度策略来指导异步 message 调度, 尽可能多地改变异步 message 执行序, 检测测试是否不稳定.

方法框架如图 2 所示. 包含两个核心模块: (1) message 间 HB 关系分析. 依据 Android 线程通信机制中的消息通信原理, 定义消息对象的 HB 关系规则, 并分析测试执行轨迹中异步 message 相关 HB 关系, 从而构建出该测试用例执行轨迹上的消息 HB 关系集, 可图示化为 message HB 关系图, 为后续的不稳定测试检测工作提供支持; (2) 异步消息执行顺序的微小变化都可能引发 Android 应用程序行为的显著变化, 进而导致测试失败. 因此, 本文所提方法第 2 模块重点在于探索异步消息的执行顺序. 即, 基于 HB 关系确定异步 message 的可执行区间, 进而设计最大差异化调度策略, 有指导性地生成可能改变测试执行结果的异步消息执行顺序, 使得调度后的测试执行轨迹与原轨迹在异步消息执行序上的差异最大化, 以尽可能改变测试执行结果, 进而检测出测试的不稳定.

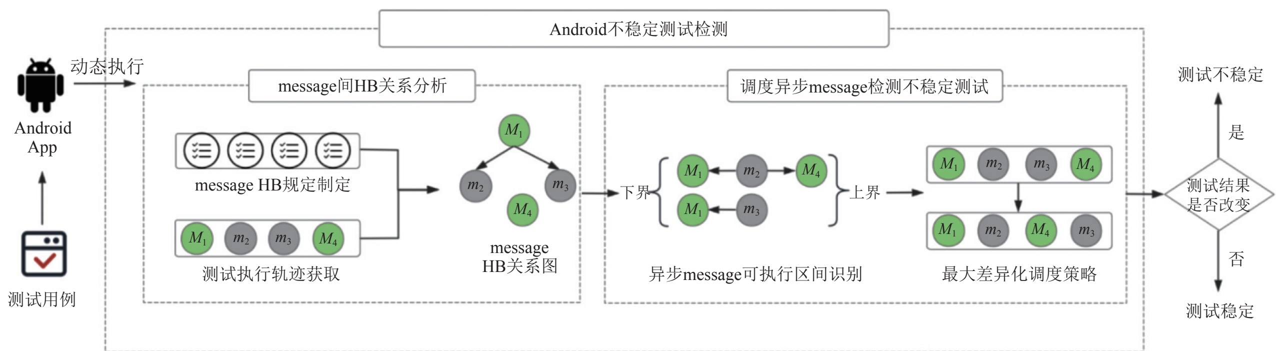

图 2 总体框架

3.2 测试轨迹 message 间 HB 关系分析

通过记录测试用例执行过程中的测试语句、方法调用、消息生命周期、线程等信息可获取完整的测试用例执行轨迹. 本文关注测试用例执行轨迹上消息的两段生命周期方法——发送 (*send*) 和处理 (*dispatch*), 并依据消息的生命周期方法执行线程、执行时间、测试轨迹中所属的执行方法等, 针对 2 条 message 的发送线程与处理线程进行分类, 具体分为以下 5 种情况.

(1) m_1 和 m_2 的发送线程 *send* 为 t_1, 处理线程 *dispatch* 为 t_2, m_1 先于 m_2 发送且 m_1 的发送延迟时间 $delay_1$ 不大于 m_2 的发送延迟时间 $delay_2$, 由于消息队列的先进先出处理原则, t_2 上 m_1 一定先于 m_2 处理;

(2) m_1 和 m_2 的发送线程 *send* 与处理线程 *dispatch* 均为 t_1, m_1 先于 m_2 发送且 m_1 的发送延迟时间 $delay_1$ 不大于 m_2 的发送延迟时间 $delay_2$, 由于消息队列的先进先出处理原则, t_1 上 m_1 一定先于 m_2 处理;

(3) m_1 和 m_2 的发送线程 *send* 与处理线程 *dispatch* 均为 t_1, 且 m_1 先于 m_2 发送且 m_2 的发送延迟时间 $delay_2$ 为 0, m_1 的发送延迟时间 $delay_1$ 大于 m_1、m_2 发送时间的差值, 那么 t_1 上消息队列一定先取出 m_2 进行处理, 即 m_2 一定先于 m_1 处理.

(4) m_1 的发送线程 *send* 为 t_1, m_1 的处理线程 *dispatch* 与 m_2 的发送线程 *send* 均为 t_2, m_2 的处理线程 *dispatch* 为 t_3, m_1 先于 m_2 发送且 m_1 的发送延迟时间 $delay_1$ 不大于 m_2 的发送延迟时间 $delay_2$, 那么 t_2 上处理 m_1 一定先于 t_3 上处理 m_2.

(5)m_1 的发送线程 *send* 和 m_2 的处理线程 *dispatch* 均为 t_1, m_1 的处理线程 *dispatch* 与 m_2 的发送线程 *send* 均为 t_2, m_1 先于 m_2 发送且 m_1 的发送延迟时间 $delay_1$ 不大于 m_2 的发送延迟时间 $delay_2$, 那么 t_2 上处理 m_1 一定先于 t_1 上处理 m_2.

上述 5 种情况可表示为以下 5 条 message 间 HB 关系规则, 如图 3 所示, 其中<表示操作发生序, $a<b$ 表示在轨迹上 a 在 b 之前执行, $<_{HB}$ 表示 HB 关系, $a<_{HB}b$ 表示不论测试执行多少次, a 一定在 b 之前执行.

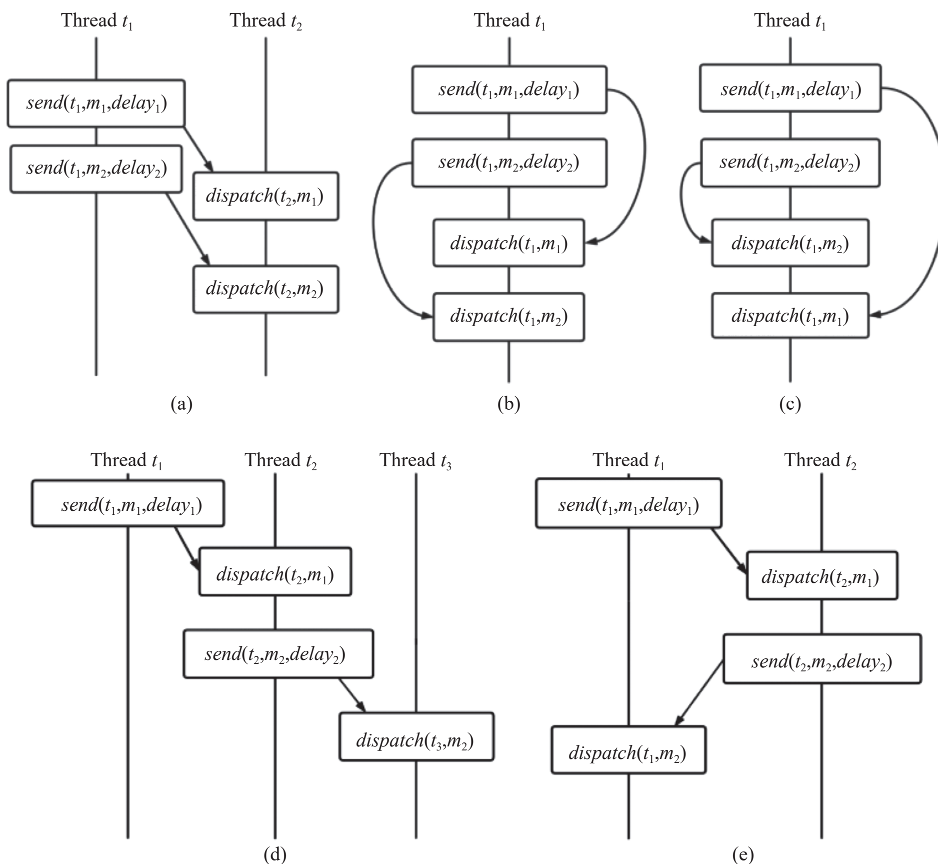

图 3 message HB 规则

规则 1: 如果 $send(t_1, m_1, delay_1)<send(t_1, m_2, delay_2)$ 且 $delay_1 \leqslant delay_2$, 那么 $dispatch(t_2, m_1)<dispatch(t_2, m_2)$, 则 $m_1<_{HB}m_2$ (图 3(a)).

规则 2:如果 $send(t_1, m_1, delay_1)<send(t_1, m_2, delay_2)$ 且 $delay_1 \leqslant delay_2$, 那么 $dispatch(t_1, m_1)<dispatch(t_1, m_2)$, 则 $m_1<_{HB}m_2$ (图 3(b)).

规则 3: 如果 $send(t_1, m_1, delay_1)<send(t_1, m_2, delay_2)$ 且 $delay_2=0$, $delay_1>time(send.m_1)-time(send.m_2)$, 那么 $dispatch(t_1, m_2)<dispatch(t_1, m_1)$, 则 $m_2<_{HB}m_1$ (图 3(c)).

规则 4: 如果 $send(t_1, m_1, delay_1)<send(t_2, m_2, delay_2)$ 且 $delay_1 \leqslant delay_2$, 那么 $dispatch(t_2, m_1)<dispatch(t_3, m_2)$, 则 $m_1<_{HB}m_2$ (图 3(d)).

规则 5: 如果 $send(t_1, m_1, delay_1)<send(t_2, m_2, delay_2)$ 且 $delay_1 \leqslant delay_2$, 那么 $diapatch(t_2, m_1)<dispatch(t_1, m_2)$, 则 $m_1<_{HB}m_2$ (图 3(e)).

其中,<表示偏序. 基于以上规则, 对于给定 Android 测试用例的执行轨迹, 可获取与异步 message 具有 HB 关系的同步/异步 message, 首先运行输入的 Android 应用以及测试用例, 记录其测试执行轨迹, 并获取测试执行轨迹中消息; 然后依据本节中提出的 HB 关系分析规则, 分析测试执行轨迹两条消息间的 HB 关系; 最后输出完整的消息 HB 关系集.

3.3 最大差异化调度策略检测不稳定测试

为有效检测不稳定测试, 本文设计了基于 HB 关系的最大差异化调度策略, 该策略的核心思想在于, 有指导性地生成可能改变测试执行结果的异步 message 执行顺序, 使得调度后的测试执行轨迹与原轨迹在异步 message 执行序上的差异最大化, 以尽可能触发 Android 应用中的不稳定情况. 通过最大差异化调度策略, 尽可能地改变应用程序行为进而改变测试结果, 更高效地检测不稳定测试.

具体而言, 分为 3 步: 异步 message 可执行区间识别; 测试执行轨迹 message 执行序列的差异度度量; 测试用例 message 执行序列最大差异化调度.

1) 可执行区间识别: 通过分析 message 之间的 HB 关系和执行顺序, 确定每个异步 message 在 UI 线程上最早可以开始执行的位置 (即执行下界) 和最晚可能执行的位置 (即执行上界), 为每个异步 message 确定一个可执行时间段, 即其可执行区间.

本文分为两类情况讨论异步 message 的执行区间. 一类为其 HB 关系集中存在前驱和后继的异步 message, 则将其前驱 message 集中第 1 个与该异步 message 存在 HB 关系的 message 定义为执行下界; 执行上界定为其之后执行的 message 中第 1 个与该异步 message 存在 HB 关系的 message; 另一类为只存在前驱节点的异步 message, 将其可执行下界定义为在其之前执行的 message 中最后一个与该异步 message 存在 HB 关系的 message, 且不设执行上界.

2) 差异度度量: 为度量测试执行轨迹调整前后的差异, 我们将测试轨迹表示为由异步 message 构成的执行序列, 本文着重考虑了异步 message 在新旧测试执行轨迹中的执行位置变化. 异步 message 执行位置差异度越大, 说明调度策略对测试执行轨迹的影响越显著, 从而更有可能检测出测试的不稳定. 如式 (1) 所示, 任一异步 message m_i 在调度前后的位置差异度 $PDC(m_j)$ 可表示为:

$$PDC(m_j) = \alpha \times |pos_{\rm new}(m_i) - pos_{\rm old}(m_j)| + \beta \times |(pos_{\rm new}(m_i) - pos_{\rm old}(m_j))/pos_{\rm old}(m_j)| \quad (1)$$

其中, 表示 m_i 在原测试执行轨迹中的执行位置; $pos_{\rm new}(m_i)$ 表示 m_i 调度后在新测试执行轨迹中的执行位置; $|pos_{\rm new}(m_i)–pos_{\rm old}(m_j)|$表示 m_i 在新旧测试执行轨迹中执行位置的绝对差异; m_i 执行位置变化的相对幅度表示为: $|(pos_{\rm new}(m_i)–pos_{\rm old}(m_j))/pos_{\rm old}(m_j)|$, 即新旧执行位置之间的差异相对于旧位置的比例, 这有助于体现出绝对差异较小但在相对意义上差异较大的位置变化; α 和 β 都表示权值且 $\alpha+\beta=1$, 本文将 α 和 β 都暂定为 0.5.

3) 最大差异化调度: 在测试用例重复动态执行过程中, 基于测试轨迹差异度度量, 通过遍历每个异步消息的可执行区间, 计算在不同位置调度该消息后的差异度, 并选择差异度最大的位置作为调度位置. 不断迭代以找到每个异步消息的差异度最大的调度位置, 上述过程描述如算法 1 所示.

算法 1. 异步消息执行位置探索算法

输入: 原测试执行轨迹 Traceold, 异步消息可执行区间 Intervalm.
输出: 异步消息在新轨迹中的执行位置 maxDiffPositions.

```
1. function FINDMAXDIFFMSGPOSITIONS(Traceold, Intervalm)
2. Tracenew = [] /*初始化新测试执行轨迹*/
3. maxDiff = –1 /*初始化最大差异度为–1*/
4. maxDiffPositions = [] /*消息执行位置*/
5. msgSeq = EXTRACTMSGSEQ(Traceold) /*提取初始异步消息序列*/
6. for i in range (0, LENGTH(msgSeq) – 1) do
7.    newMsg = newMsgSeq[i]
8.    oldMsgSeq = EXTRACTMSGSEQ(Traceold)
/*从旧轨迹中提取异步消息序列*/
9.    oldPos = FINDMSGPOS(oldMsgSeq, newMsg)
/*在旧轨迹中查找该消息的位置*/
10.   for j in range(0, LENGTH(Intervalm[i]) – 1) do
/*遍历 m_i 的可执行区间*/
11.      Tracenew = GETTRACE(newMsg, j) /*获取新测试执行轨迹*/
12.      newMsgSeq = EXTRACTMSGSEQ(Tracenew)
/*从新轨迹中提取异步消息序列*/
13.      newPos = FINDMSGPOS(newMsgSeq, newMsg)
/*在新轨迹中查找该消息的位置*/
14.      currDiff = INVERSE(newMsgSeq, oldMsgSeq)
                 +PDC(oldPos, newPos) /*计算差异度*/
15.      if currDiff <= maxDiff then /*当前差异度不大于最大差异度*/
16.        j++
17.      else
18.        maxDiff = currDiff
19.        j++
20.      end if
21.   end for
22.   maxDiffPositions.add(newPos) /*存储差异度最大时消息的位置*/
23.   Traceold = Tracenew 24: maxDiff = –1 /*迭代*/
25. end for
26. return maxDiffPositions
27. end function
```

获取异步消息的差异度最大的调度位置分析出异步消息的最大差异调度位置后, 通过对异步线程的挂

起和释放操作, 在测试用例的执行过程中动态调度异步消息, 使其在预定的调度位置执行. 通过多次迭代和动态调度, 改变测试执行轨迹上异步消息的执行序, 使得新生成的测试执行轨迹与原测试执行轨迹差异度最大. 在调度过程中, 观察测试用例的测试结果是否发生改变, 若发生改变, 则该测试为不稳定测试.

4 实验与评估

在本节中, 评估了本文方法检测 Android 应用不稳定测试的有效性. 并设计实验与当前技术进行对比.

4.1 测试对象

为了评估本文方法, 本文在已有不稳定测试用例集 FlakyAppRepo[18]的基础上扩充了 20 个 Android 应用. 在 Github 上搜索 flaky test、flake 等关键词, 并手动检查搜索到的存储库, 选择包含不稳定测试的 Android 应用项目. 该 20 个 Android 应用项目均为目前流行的开源项目, 且带有测试用例. 如表 1 所示, 其中#Stars 为 GitHub 上该项目的点赞量以表示其流行度.

表 1 测试应用集

应用项目名	版本	#Stars	类别
Owntracks	2.3.0	1.2k	地图和导航
Amahi	1.1.0	159	工具
Mega	7.8	1.3k	工具
ShadowSocks	4.6.5	34.2k	网络
CycleStreets	1.0	205	导航
CatimaLoyalty	2.18.1	587	金融
Cryptomator	1.10.0	588	通信
DeltaIcons	1.8.5	458	娱乐
JZ-Darkal	1.0	4.3k	网络
TrebleShot	2.0.4	126	生产力
KickStarter	3.16.1	5.7k	工具
EduVpn	3.1.1	110	网络
WildFireChat	1.1.0	2.4k	通信
HabitRPG	4.0.2	1.3k	娱乐
QtScrcpy	12.8k	2.1.2	工具
ProtonVPN	4.8.38	1.6k	网络
Zulip	1.3.2	409	生产力
Image Picker	2.4.5	1.1k	相机
ProtonMail	3.0.17	1.6k	邮件
BookDash	1.1.0	692	图书

4.2 研究问题

本文的评估旨在解决以下问题.

RQ1: 本文方法检测不稳定测试有效性如何?

RQ2: 本文方法检测不稳定测试效率如何?

1) RQ1 不稳定测试检测有效性分析

为了评估本文方法检测不稳定测试的有效性, 本文面向 40 个 Android 应用项目的 50 个已知不稳定的测试用例设计实验, 应用本文方法和目前主流的两种检测工具 FlakeScannner[18]和 RERUN[13] (其中 RERUN 方法默认重复运行 100 次) 进行不稳定测试检测. 其中本文方法与 RERUN 工具相比增加了 HB 关系分析以及异步 message 调度, 与 FlakeScanner 工具相比增加了 HB 关系分析.

结果如表 2 所示. RERUN 方法仅检测出了上述 50 个已知不稳定测试用例中的 10 个, FlakeScanner 成功检测出其中的 47 个, 而本文方法成功检测出了全部 50 个已知不稳定的测试用例. 由此可见, 本文方法在检测不稳定测试的有效性上优于当前不稳定测试检测工具.

同时为了分析出本文方法在检测不稳定测试的效果上优于 FlakeScanner 的原因, 本文对 FlakeScanner 未检测到但本文方法检测出的 3 个不稳定的测试用例 glideMemoryCategory、testRepeatedSave、fixtureWithMapped 进行了研究. 通过记录 FlakeScanner 和本文方法对上述 3 个测试用例进行不稳定测试检测时调度异步 message 所生成的 message 执行序列并进行对比后发现, 由于 FlakeScanner 在调度异步 message 时采取类似 FIFO 策略的调度算法, 导致部分异步 message 间的执行顺序是无法交换的. 对于这 3 个测试用例, 本文方法通过调度异步 message 生成的异步 message 执行顺序是 FlakeScanner 无法通过其自身调度算法得到的.

2) RQ2 不稳定测试检测效率分析

进一步通过调度次数、检测时间、异步消息数和总消息数 4 个指标对 3 种方法进行对比. 本文方法在检测过程中执行的消息数更少、调度次数更少且平均检测时间更短.

首先就检测时间而言, 在检测不稳定测试成功的前提下, 本文方法的平均检测时间为 45.9 s, 相比于 FlakeScanner 缩短了 31.78%, 相比于 RERUN 缩短了 91.88%. 其次, 相比于 lakeScanner, 本文方法在检测过程中平均执行的异步消息数和总消息数分别为 31.92 个和 1 251.22 个, 其数量相比于 FlakeScanner 分别减少了 36.39% 和 40.43%, 调度过程中对异步消息的平均调度次数也减少了 33.54%.

实验结果表明, 本文方法在检测过程中, 平均能在更短的时间内更加精确地调度异步消息, 触发应用中的不稳定情况, 检测出不稳定测试.

表 2　本文方法 (MM)、FlakeScanner (FS) 和 RERUN (RR) 的测试结果对比

序号	测试用例方法名	调度次数 (次)		检测时间 (s)			异步消息 (个)			总消息数 (个)			不稳定		
		FS	MM	RR	FS	MM	RR	FS	MM	RR	FS	MM	RR	FS	MM
1	overflowMenu···	11	4	180	78	62	44	14	11	724	363	277	—	√	√
2	checkLongCli···	4	3	172	48	32	31	8	6	687	379	252	—	√	√
3	testStartActivity	3	5	780	70	54	21	7	5	3 982	2 782	2 075	—	√	√
4	testBackButto···	3	3	120	36	26	41	10	7	705	353	241	—	√	√
5	testSwitchThe···	6	10	364	26	30	29	11	15	568	297	361	—	√	√
6	generateRSA···	4	3	19	33	21	31	10	6	473	267	130	√	√	√
7	localConnect···	4	4	110	38	23	29	10	5	398	255	150	—	√	√
8	testOpenFile	6	8	11	25	28	58	10	12	3 765	1 023	1 362	√	√	√
9	testEncryptDe···	14	6	1 370	33	24	41	20	10	2 586	1829	1 162	—	√	√
10	glideMemory···	53	33	1 492	42	31	53	53	29	3 451	3 674	1868	—	—	√
11	testMoveFiles···	19	11	281	21	12	129	30	22	5 421	1932	1 352	—	√	√
12	deleteSshFile···	291	162	2 910	229	143	492	321	192	19 284	15 421	8 310	—	√	√
13	migrateAll	91	69	848	78	40	294	119	64	5 992	2 955	1 102	—	√	√
14	testSimpleSave	8	7	183	21	21	23	11	8	423	227	112	—	√	√
15	testDeleteSmb···	51	63	392	39	54	119	54	62	1 002	523	687	—	√	√
16	repeatedSave···	11	9	281	30	21	72	21	19	1 672	430	389	—	√	√
17	saveCommon···	9	3	101	25	13	24	10	6	1 298	982	728	—	√	√
18	checkFontSize···	28	15	292	72	39	91	38	23	7 902	4 782	3 221	—	√	√
19	localConnecti···	6	3	24	32	15	28	14	4	1 029	603	104	√	√	√
20	canScrollTerm···	29	18	194	27	11	289	38	19	9 931	1 028	789	—	√	√
21	addHostThen···	76	51	289	121	79	362	80	48	8 738	1 783	932	—	√	√
22	testGetLocalI···	4	10	939	20	36	19	9	11	361	176	209	—	√	√
23	testRepeatedS···	72	57	1 910	129	74	72	72	49	6 289	6 305	3 782	—	—	√
24	writeToParcel	18	8	228	37	23	182	21	15	3 918	492	386	—	√	√
25	getAndSetName	21	14	812	32	21	187	37	21	4 491	751	539	—	√	√
26	getImageLoader	5	9	15	23	30	28	9	12	893	317	391	√	√	√
27	stripPathTest	9	5	104	21	14	59	17	7	2 918	1 028	537	—	√	√
28	extractZipWith..	40	32	21	67	56	89	53	41	8 273	4 928	3 781	√	√	√
29	testZipHelper···	4	3	238	12	10	29	11	9	718	212	187	—	√	√
30	getExtractorIn···	18	18	592	43	44	582	32	24	11 283	731	482	—	√	√
31	getFileNameT···	91	63	11	242	147	329	124	82	7 192	2 891	1 293	√	√	√
32	testDir0File1···	82	54	931	64	45	192	121	78	5 192	3 918	2019	—	√	√
33	testGetPathsIn···	7	3	291	15	10	94	18	7	7 281	1 182	419	—	√	√
34	testCopyFile1	31	17	784	43	29	82	57	29	6 719	4 728	2 682	—	√	√
35	assertSha1Equ···	8	6	16	30	22	74	17	11	1 981	498	301	√	√	√
36	saveConnecti···	3	8	439	28	59	18	7	10	521	201	283	—	√	√
37	currenciesAre···	111	72	2 029	132	77	399	128	92	9 283	3 180	2 819	—	√	√
38	testDown	19	9	491	73	41	48	32	11	1 239	912	382	—	√	√
39	saveAccount	162	113	2 903	482	326	578	197	148	20 381	6 920	5 618	—	√	√
40	fixtureWith···	14	10	593	57	38	28	28	19	1 192	1 202	729	—	—	√
41	shouldNotDel···	70	49	9	43	25	238	86	32	13 892	4 918	1876	√	√	√
42	calculatorMai···	10	6	17	22	12	48	25	13	2 827	1 228	571	√	√	√
43	amountIsDisp···	29	38	323	39	55	84	37	42	3 799	1 192	1 378	—	√	√
44	amountIsNot···	4	2	295	21	14	28	11	7	1 182	501	398	—	√	√
45	cloneComman···	20	14	453	59	31	191	37	18	13 881	1972	912	—	√	√
46	editCommand···	180	17	2 051	274	32	333	182	32	9 871	5 821	1 029	—	√	√
47	deleteComma···	71	69	225	73	65	216	93	92	1 828	817	791	√	√	√
48	splitCommand···	6	17	362	38	105	43	12	24	827	295	489	—	√	√
49	shouldNavigat···	30	8	443	39	11	54	34	11	3 991	1 921	682	—	√	√
50	manageParties	96	63	312	82	64	123	113	76	5 820	3 891	1 992	—	√	√
平均值/总数		39.24	26.08	565	67.28	45.9	134.96	50.18	31.92	4 761.48	2 100.32	1 251.22	10	47	50

注: 限于表格篇幅, 部分测试用例方法名未完整展示, 用···表示

5 结论与展望

不稳定测试严重削弱了软件测试的可靠性和准确性. 本文针对 Android 应用测试用例执行轨迹, 提出了一种基于 HB 关系的 Android 应用不稳定测试检测方法. 通过分析测试用例执行轨迹上消息间的 HB 关系, 进行基于 HB 关系的最大差异化调度策略研究, 以有效检测不稳定测试. 在 50 个已知不稳定测试用例上进行实验, 与当前主流两种工具对比, 结果表明, 本文方法能够高效检测全部不稳定测试检测.

未来将继续探索 message 间的 HB 关系, 设计更优的调度策略, 进一步优化 Android 应用不稳定测试检测方法.

参考文献

1 Parry O, Kapfhammer GM, Hilton M, *et al*. A survey of flaky tests. ACM Transactions on Software Engineering and Methodology, 2022, 31(1): 17.
2 Gruber M, Lukasczyk S, Kroiß F, *et al*. An empirical study of flaky tests in python. Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation. Porto de Galinhas: IEEE, 2021. 148–158.
3 Hashemi N, Tahir A, Rasheed S. An empirical study of flaky tests in JavaScript. Proceedings of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). Limassol: IEEE, 2022. 24–34.
4 Alshammari A, Morris C, Hilton M, *et al*. Flakeflagger: Predicting flakiness without rerunning tests. Proceedings of 43rd IEEE/ACM International Conference on Software Engineering (ICSE). IEEE, 2021. 1572–1584. [doi: 10.1109/ICSE43902.2021.00140]
5 Luo QZ, Hariri F, Eloussi L, *et al*. An empirical analysis of flaky tests. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Hong Kong: ACM, 2014. 643–653.
6 Ahmad A, Leifler O, Sandahl K. Empirical analysis of practitioners' perceptions of test flakiness factors. Software: Testing, Verification and Reliability, 2021, 31(8): e1791. [doi: 10.1002/stvr.1791]
7 Listfield J. Google testing blog: Where do our flaky tests come from? https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html. (2017-04-17)[2024-08-07].
8 Lam W, Godefroid P, Nath S, *et al*. Root causing flaky tests in a large-scale industrial setting. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. Beijing: ACM, 2019. 101–111.
9 Eck M, Palomba F, Castelluccio M, *et al*. Understanding flaky tests: The developer's perspective. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn: ACM, 2019. 830–840.
10 Meike GB. Android Concurrency. Boston: Addison-Wesley Professional, 2016.
11 Thorve S, Sreshtha C, Meng N. An empirical study of flaky tests in Android APPs. Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution. Madrid: IEEE, 2018. 534–538.
12 王潇, 杨秋辉, 刘芳, 等. Randoop 和 Evosuite 生成的测试用例的质量分析. 计算机应用研究, 2020, 37(S1): 218–220.
13 Zhen D, Abhishek T, Yu XL, *et al*. Concurrency-related flaky test detection in Android Apps. arXiv:2005.10762.
14 Silva D, Teixeira L, d'Amorim M. Shake it! Detecting flaky tests caused by concurrency with shaker. Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). Adelaide: IEEE, 2020. 301–311.
15 Adamsen CQ, Mezzetti G, Møller A. Systematic execution of Android test suites in adverse conditions. Proceedings of the 2015 International Symposium on Software Testing and Analysis. Baltimore: ACM, 2015. 83–93.
16 Shi A, Gyori A, Legunsen O, *et al*. Detecting assumptions on deterministic implementations of non-deterministic specifications. Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation. Chicago: IEEE, 2016. 80–90.
17 Zhang S, Jalali D, Wuttke J, *et al*. Empirically revisiting the test independence assumption. Proceedings of the 2014 International Symposium on Software Testing and Analysis. San Jose: ACM, 2014. 385–396.
18 Zhen D, Abhishek T, Yu XL, *et al*. Flaky test detection in Android via event order exploration. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens: ACM, 2021. 367–378.
19 Hsiao CH, Yu J, Narayanasamy S, *et al*. Race detection for event-driven mobile applications. ACM SIGPLAN Notices, 2014, 49(6): 326–336. [doi: 10.1145/2666356.2594330]
20 Wu DY, Liu J, Sui YL, *et al*. Precise static happens-before analysis for detecting UAF order violations in android. Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification. Xi'an: IEEE, 2019. 276–287.
21 Salehnamadi N, Alshayban A, Ahmed I, *et al*. ER catcher: A static analysis framework for accurate and scalable event-race detection in Android. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne: IEEE, 2020. 324–335.
22 Hu YJ, Neamtiu I. Static detection of event-based races in Android Apps. Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems. Williamsburg: ACM, 2018. 257–270.
23 Robotium. https://github.com/RobotiumTech/robotium. [2024-08-07].
24 Espresso. https://developer.android.com/training/testing/espresso?hl=zh-cn. [2024-08-07].

(校对责编: 王欣欣)