从源代码语义生成基于性质的回归测试

Clemens Bauer Johann Glock信息学系 软件能力中心 *Hagenberg* 有限责任公司 克拉根福大学 Hagenberg，奥地利 克拉根福，奥地利
johann.glock@scch.at clemens.bauer@aau.at

Martin Pinzger信息学系 克拉根福大学克拉根福，奥地利
martin.pinzger@aau.at

arXiv:2611.32636v1 [cs.SE] 24 Nov 2026

摘要—**传统单元测试（被测类）在固定的具体输入下检查被测方法。因此，如果某个输入尽管存在缺陷仍产生预期输出，测试就可能遗漏回归问题。基于属性的测试（PBT）针对大量随机生成的输入检查全称量化的规格，但手工编写性质和输入生成器十分困难。现有方法通过概括现有的或生成的单元测试套件来自动创建基于属性的测试。它们要么在重复测试中对具体输入进行过度近似，要么沿着具体测试运行应用单路径符号分析。**

本文提出了一种通过多路径符号执行（SE）直接从源代码语义中推导基于属性的回归测试的方法。对于通过目标方法的每条已探索路径，该方法都会记录路径语义，其中包括路径条件、符号返回值和状态变更映射。每条路径语义都会导出一个霍尔三元组，并编译为可执行属性。由于符号执行会使类的部分内容保持未指定状态，该方法还使用基于搜索的被测类为整个类补充生成的性质。

我们在 EqBench 以及 TOGBench 中的 11 个项目上，将我们的 Java 实现与 EvoSuite 和测试泛化工具 Teralizer 进行了评估。在 EqBench 上，我们的方法为所有方法中的 82.0% 保留了经验证的属性。其平均变异得分为 63.8%，高于 EvoSuite 的 55.2% 和 Teralizer 的 60.3%。在 TOGBench 上，目标选择、符号执行限制和属性验证使经验证的属性仅占所有方法的 2.8%。由此得到的平均变异得分为 48.6%，低于 EvoSuite 的 50.9% 和 Teralizer 的 51.1%。测试生成期间发生的失败使少数类没有有效测试，从而导致了这一差距。

索引词——**基于属性的测试、符号执行、自动化测试生成**

I. 引言

EvoSuite [1] 和 Randoop [2]等自动化单元测试生成器会生成传统单元测试（CUT），以实现结构覆盖率，并断言在具体输入下观察到的行为。然而，高结构覆盖率并不能保证故障检测 [3], [4]。当变更或变异体沿着已执行的路径改变某个操作时，如果故障没有改变该测试输入对应的输出，CUT 仍会通过。未能揭示已执行故障的这种情况称为偶然正确性 [5], [6]。图1的步骤1展示了一个测试套件示例，其中的 CUT 实现了完整的分支覆盖，却未能检测出被测方法的行为变化。数据驱动的测试参数化会将相似的 CUT 或记录的调用合并为参数化单元测试（PUT），但仍然只检查固定列表中的输入 [7]–[9]。基于属性的测试（PBT） [10], [11] 则将输入生成器与参数化断言配对，并在每次运行时针对新采样的输入检查该断言，因此每个测试杀死的变异体比 CUT 更多 [12]。

然而，基于属性的测试在实践中仍然很少见 [12]，[13]。开发人员难以在没有参考实现的情况下描述一般性的后置条件，也难以编写满足输入前置条件的生成器 [14], [15]。将现有单元测试手动转换为性质还需要重写测试设置代码和判定器代码 [13]。因此，近期方法通过泛化现有或生成的单元测试来自动完成基于属性的测试创建 [16], [17]。Peleg 等人 [16]将重复的测试执行轨迹归组为场景，并基于其具体值实例化预定义的抽象模板。由于这些模板会对观测到的示例进行过度近似，生成的性质可能在有效代码上失败。Glock 等人 [17] 则沿着每个被测类的具体执行路径运行单路径符号分析，以提取路径精确规范。然而，从被测类出发会将泛化限制在被测类已经遍历的执行路径，以及其断言明确检查的行为范围内。他们的工具 Teralizer 还要求将每个断言映射回单个被测试的方法调用，并且仅支持无副作用、非 void 方法以及不包含字段更新的方法。

本文介绍了 EvoSuite-PBT，这是一种从目标方法的源代码而非其测试中推导基于属性的回归测试的方法。EvoSuite-PBT 通过多路径符号执行（SE）探索该方法中的可行路径 [18], [19]。与符号方法
差分分析中的摘要 [20], 我们的方法记录每条终止路径 p 的路径语义 $sem(p) = \langle \pi,\ \rho,\ \sigma \rangle$，即路径条件 π、符号值 ρ以及对已修改实例字段的状态变更映射 σ （图 1，步骤 2）。当符号执行引擎准确地对路径建模时，每条路径语义都会导出一个全正确性霍尔三元组 [21]，该三元组将前状态值冻结在逻辑变量 $\bar{v}$ 中，并同时规约返回值和后置状态字段更新。我们将每个三元组编译为一个可执行属性。图1的步骤 3 显示，前置条件 $\pi_{2.b}$ 会变成生成器 gen2，它会筛选出满足 金额 =余额的输入。后置条件则变成 test2 中针对返回值 $\rho(\bar{v})$ 和字段 σ(余额)$(\bar{v})$ 的断言。



图 1. withdraw 的运行示例。点见证 CUT 实现了分支覆盖，但未能检测到边界变异体 amount >= balance。

遇到循环、未建模的本机调用和复杂对象时，符号执行并不完备 [19],，因此提取出的性质构成了类的部分规约。由于当路径条件过于严格时，基于过滤器的生成器可能无法采样到足够多的有效输入 [10], [14],，我们的方法会对每个候选属性进行少量尝试，并丢弃未通过的属性。随后，我们会用为该类剩余部分生成的基于搜索的被测类，补充剩余属性。

我们基于 Evo- Suite [1] 及其动态符号执行引擎 [22], Symbolic PathFinder [23],和 jqwik，实现了面向 Java 的方法。我们在数值基准测试 EqBench [24] 以及来自 TOGBench [25] 的 11 个 Java项目上评估了该实现，并将其与基线 EvoSuite 和 Teralizer（Glock 等人 [17]提出的测试泛化工具）进行了比较。在 EqBench 上，我们的方法为全部方法中的 82.0% 保留了经验证的属性。其平均变异得分为 63.8%，高于 EvoSuite 的 55.2% 和 Teralizer 的 60.3%。平均测试强度同样达到 65.0%，高于 EvoSuite 的 56.3% 和 Teralizer 的 61.5%。这些属性是取得领先的原因。在 TOGBench 上，目标选择、符号执行在对象构造和约束求解方面的限制，以及属性验证，将经验证的属性限制在全部方法的 2.8%。平均变异得分为 48.6%，低于 EvoSuite 的 50.9% 和 Teralizer 的 51.1%。少数类在测试生成过程中出现故障，导致了这一差距。本文作出以下贡献：

- 一种通过多路径符号执行提取返回值和实例字段更新的符号路径语义⟨π， ρ， σ⟩并将诱导的霍尔三元组编译为可执行属性的方法（第 III-A–III-E 节）。
- 一种混合测试生成流程及其开源 Java 实现 EvoSuite-PBT，将经过验证的路径派生属性与面向整个类的基于搜索的测试用例相结合（第 III-F–III-G 节）。
- 在 EqBench 和 TOGBench 上与 EvoSuite 和 Teralizer 进行实证评估，衡量直接提取带来的故障检测收益，以及面向对象代码中各阶段的适用性障碍（第 IV 节）。

II. 背景

本节回顾了我们的方法所结合的两种测试范式，以及将它们联系起来的程序分析。内容涵盖传统单元测试、使用 jqwik 进行的基于属性的测试，以及符号执行。

A. 传统单元测试

在传统单元测试中，被测类会在隔离环境下使用固定参数调用目标方法，并将得到的返回值或对象状态与预期值进行比较 [26]。由于单个输入很少能覆盖方法的所有分支，测试套件通常会为每个方法包含多个被测类。若多个重复的被测类具有相同的断言结构，则可将它们整合为一个参数化单元测试，让同一测试主体针对数据源 [27] 提供的各种具体输入元组运行。尽管参数化消除了重复的测试代码，测试预言机仍受限于测试类中列出的有限组具体输入—输出示例。

B. 基于属性的测试

基于属性的测试会针对满足前置条件 [10], [14] 的随机生成输入，检查方法是否符合一个全称量化的规范。我们的实现面向*jqwik*，[1] 这是一款运行于 JUnit 5 平台、适用于 Java 虚拟机（JVM）的基于属性的测试引擎。在 jqwik 中，性质是一个带有 @Property 注解的方法，其参数带有 @ForAll 注解。每个参数都会从特定于其类型的生成器（称为 Arbitrary）中获取值；该生成器会在提供随机样本的同时，生成指定的边界情况。生成器既可以通过 @IntRange 等范围注解进行声明式约束，也可以通过带有 @Provide 注解的提供器方法进行编程式约束；这些方法会构造并筛选 Arbitrary 实例。当生成器使用过滤谓词时，jqwik 会对候选输入进行采样，并丢弃不满足谓词的输入。如果连续拒绝的候选输入过多，jqwik 就会以 TooManyFilterMissesException 异常中止该性质。

[1]https://jqwik.net

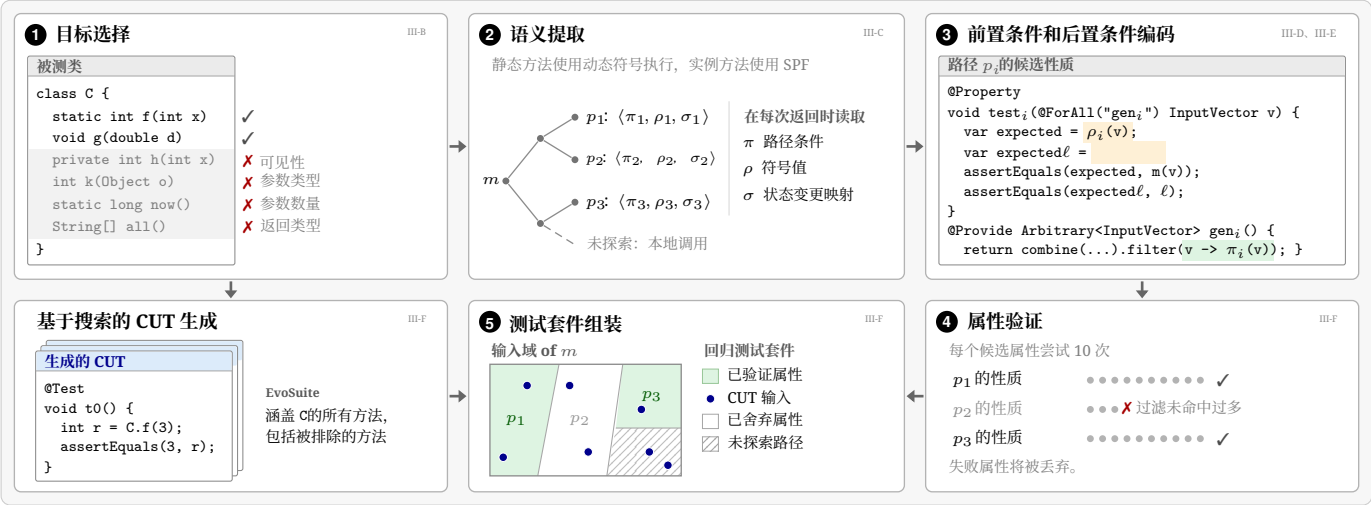

图 2。EvoSuite-PBT 概览。灰色标签标出了描述每个步骤的章节。目标方法 m 的每条探索路径 p_i 都会产生路径语义⟨π_i， ρ_i， σ_i⟩ 以及一个候选性质，其中 π_i、 ρ_i 和 σ_i 表示它们转换为 Java 表达式后的结果。

C. 符号执行

符号执行 [18], [19] 会基于代表任意值而非具体值的符号输入来分析程序。在一条执行路径上，SE 引擎维护一个由符号存储、路径条件和指令计数器组成的符号状态。符号存储将程序变量和对象字段映射到关于入口输入的符号表达式，而路径条件则合取沿路径累积的分支约束。在每个条件分支处，引擎会分叉符号状态，将分支条件或其否定与路径条件合取，并查询约束求解器以剪除不可行路径。当执行到达返回指令时，符号存储和操作数栈会提供返回值以及任何被修改的实例字段对应的符号表达式。

在实践中，SE 受到路径爆炸和约束求解 [19]的限制。循环和嵌套分支可能产生指数级或无限数量的可行路径，因此需要对搜索深度、输入大小或符号状态抽象 [19], [28] 设置界限。约束求解器在浮点运算、字符串和非线性约束方面也面临可判定性与性能限制，而且标准 SE 无法跟踪跨越未建模本地方法的执行过程 [19], [28]。动态符号执行（DSE）或具体符号执行通过沿具体执行路径驱动 SE，缓解了其中一些限制 [19],[22], [29]。DSE 在具体输入上运行方法，收集所执行路径上的符号分支约束，并对分支条件取反，以求解出能够引导执行走向未覆盖路径的新输入。

III. 方法

EvoSuite-PBT 会生成回归测试套件，其中结合了从提取出的程序语义中推导出的基于属性的测试，以及传统单元测试。该方法不会根据现有或生成的测试套件中的断言和输入进行泛化，而是 [16], [17], EvoSuite-PBT 通过多路径符号执行从目标方法中提取性质，并使用回退 CUT 进行补充。对于目标方法中探索的每条路径，该方法都会记录对方法行为的符号化描述。第 III-A 节将这种描述形式化为一个霍尔三元组。

图2展示了 EvoSuite-PBT 如何提取这些三元组，并将其编译为可执行的 jqwik 性质。目标选择（第III-B节）选择该方法可以处理的方法。语义提取（第III-C节）为每条被探索的路径计算一条路径语义，每条路径语义都会产生一个候选属性。前置条件编码（第III-D节）将每个路径条件编译为一个生成器，而后置条件编码（第III-E节）则将符号值和状态更新编译为性质的主体。生成器会对满足路径条件的输入进行采样。断言会比较返回值和已写入的字段与符号值及状态更新；二者均在采样输入传入调用之前进行求值。

目标选择和符号执行引擎的局限性使类的部分内容处于未指定状态。因此，EvoSuite-PBT 会验证候选属性，并使用 EvoSuite 的基于搜索的生成器为整个类生成的 CUTs 对其进行补充。属性验证（第III-F节）会对每个候选属性运行 10 次，并丢弃运行失败的属性。测试套件组装（第III-F节）将通过验证的属性与生成的 CUTs 结合，形成由 JUnit 5 和 jqwik 源代码组成的回归测试套件。对于目标方法的输入域，每个已验证属性都会对其路径的输入空间进行采样，而 CUTs 只检查固定的具体输入。

A. 作为霍尔三元组的路径语义

设 m 为被测方法， p 为经过m 且以返回语句结束的一条可行路径。符号输入 $\bar{x} = (x_1, \ldots\ldots, x_n)$ 是 m 的参数；如果 m 是实例方法，则还包括它所引用的任何

字段。 p 的路径语义是三元组$sem(p) = \langle \pi,\ \rho,\ \sigma \rangle$，其组成部分如下：

- 路径条件 π 是符号执行引擎沿 p收集的关于 $\bar{x}$ 的分支约束的合取。
- 符号值 ρ是关于 $\bar{x}$ 的表达式，表示 m在 p上返回的值。
- 状态变更映射 σ 是一个从 $\bar{x}$ 到关于 $\bar{x}$的表达式的有限部分映射。其定义域 $\mathrm{dom}(\sigma) \subseteq \bar{x}$ 是 $\bar{x}$中沿 p修改的实例字段子集，而 $\sigma(\ell)$ 表示字段 ℓ的最终值。

对于关于 $\bar{x}$ 的表达式 e 以及由值组成的元组 $\bar{v}$，我们用$e(\bar{v})$表示 e 在替换 $\bar{x} \mapsto \bar{v}$下的值；$\mathrm{Sat}(\pi) = \{\bar{v} \mid \pi(\bar{v})\}$ 表示满足 π的输入集合。

路径语义为所分析版本的 m导出一个霍尔三元组：

$$\{\, \bar{x} = \bar{v} \wedge \pi(\bar{v}) \,\} \quad r := m(\bar{x}) \quad \{\, Q_p(\bar{v}, r) \,\} \tag{1}$$

以及后置条件

$$Q_p(\bar{v}, r) \;\equiv\; r = \rho(\bar{v}) \;\wedge \bigwedge_{\ell \in \mathrm{dom}(\sigma)} \ell = \sigma(\ell)(\bar{v}), \tag{2}$$

如果 m 不返回值，则第一个合取项不存在。逻辑变量 $\bar{v}$ 冻结了前状态。之所以需要它们，是因为 p 可能写入它同时读取的字段，因此后置条件中出现的 ℓ 表示该字段的后状态值，而 $\sigma(\ell)(\bar{v})$则表示方法入口处的值。

对于纯静态方法，每条路径上的 $\sigma = \emptyset$ 以及 $Q_p(\bar{v},\ r)$ 都会归约为 $r = \rho(\bar{v})$。图1以实例方法 withdraw 为例说明了一般的有状态情形。withdraw 的符号输入是参数 $amount$ 和字段 $balance$，每条路径都会读取这两者。由于 amount 和 balance 的类型均为 double，JVM 将比较 $amount > balance$ 编译为浮点比较指令，SPF 会为该指令创建三个状态（>、 <和 =），从而得到三种路径语义。

我们为每种路径语义生成一个性质。给定 $\bar{v} = (a,\ b)$，三元组 (1) 将路径 $p_{2.a}$ 实例化为

$$\{\, amount = a \wedge balance = b \wedge a < b \,\}$$
$$r := \texttt{withdraw}(amount)$$
$$\{\, r = b - a \;\wedge\; balance = b - a \,\}.$$

路径 $p_{2.b}$ 的三元组具有相同的后置条件，但其前置条件包含 $a = b$ 而非 $a < b$ （第 III-D 节）。将此案例保留为单独的性质，可明确测试边界金额 =余额，其中，将 >= 写成 >这类差一错误会改变行为。

对于 EvoSuite-PBT 从 m 提取该三元组的版本而言，三元组 (1) 满足完全正确性。每个满足 π 的输入都会沿着 p 执行， p 最终都会返回，而返回时的符号状态表示返回值和已写入的字段。这一论证基于一个假设：引擎能够无误地捕获 p 的语义；如果 m 是非确定性的，或执行了引擎无法建模的结构，那么该假设就不成立。若该假设成立，则每个三元组的有效性彼此独立。探索质量决定了这些三元组对 m 的规约程度。由于 m 是确定性的，不同路径的路径条件互斥，因此提取出的三元组将其覆盖的输入域划分为若干部分。它们的合取构成了对 m 的部分规约，而回退 CUT 则覆盖未被规约的输入区域。

B. 目标选择

目标选择根据签名适用性筛选类的方法，选出参数和返回值均属于 EvoSuite-PBT 可建模和编码的类型的方法。EvoSuite-PBT 会选择满足以下四项标准的方法：

- 可见性。 该方法不是私有方法。尽管可以通过反射访问私有方法，但开发者通常会通过调用它的公共方法 [30] 来测试它。
- 参数类型。 每个参数的类型都是原始类型或 String。这些限制源于符号执行和底层约束求解器的使用。
- 参数数量。对于静态方法，必须至少有一个参数，才能启动符号执行分析。
- 返回类型。 返回类型必须是基本类型或 String 类型。对于实例方法，也允许使用 void。

这些标准将性质生成限制为类中方法的一个子集。EvoSuite 的基于搜索的生成器采用较宽松的选择标准，并且还会为被排除的方法生成被测类。因此，被测类会覆盖符合条件的方法和被排除的方法，而性质仅为通过目标选择的方法补充被测类。

C. 语义提取

EvoSuite-PBT 使用 EvoSuite 的动态符号执行（DSE）引擎为静态方法提取路径语义，使用 Symbolic PathFinder（SPF） [23], (Java PathFinder（JPF）的扩展）为实例方法提取路径语义。所选引擎会记录分支约束，并在适用时记录沿执行路径发生的状态变化。EvoSuite-PBT 通过在目标栈帧的返回指令处设置监听器来介入此过程。在这一时刻，监听器从路径条件收集器中读取 π，将 ρ 作为符号操作数栈顶端的表达式，并从符号存储的条目中获取 σ。

符号执行（第 II-C 节）的局限性使探索不完整，因此 EvoSuite-PBT 对 m 的行为进行了欠近似。首先，循环会产生随着每次迭代不断增长、数量可能无界的路径，因此无法在短时间内完成探索。其次，符号执行引擎无法忠实地建模所有语言结构。调用本地方法就是一个例子。

D. 前置条件编码

前置条件编码将 π 编译为生成器，使 jqwik 仅对遵循该路径的输入进行采样。EvoSuite-PBT 支持两种编码方式，并根据

π的形式在两者之间进行选择。如果约束的形式为 $x \bowtie c$ 或 $c \bowtie x$，并且包含输入参数 x、常量 c以及$\bowtie \in \{=, <, \leq, >, \geq\}$，则该约束为简单约束。其他所有约束均为复杂约束，尤其是关系约束，例如 $x_1 < x_2$；不等式约束，例如 $x \neq 0$；以及涉及单个输入的非线性约束。

这种区分用于选择两种编码方式之一。仅由简单约束组成的路径条件可以使用 jqwik 注解表示，无需额外的辅助代码。其他路径条件则需要自定义生成器代码，该代码通过辅助方法提供。

a) 基于注解的编码：基于注解的编码旨在减少生成的测试代码规模。它要求路径条件仅由简单约束组成。基于这一假设，可以使用 jqwik 为属性参数提供的注解对路径条件进行建模。对于符号输入参数 x，令 C_x为 π 中提及 x的约束。每个简单约束c表示 x定义域上的一个区间 $I(c)$，而 EvoSuite-基于属性的测试将 x的输入空间计算为

$$I(x) = \bigcap_{c \in C_x} I(c), \tag{3}$$

从所声明类型的完整范围开始。随后，使用 jqwik 提供的范围注解（如 @IntRange 和 @FloatRange）对所得区间进行建模。例如，路径条件 $\pi = x \geq 0$会编译为 @IntRange(min = 0)。带有 $C_x = \emptyset$的输入参数仍不受约束，并会接收一个普通的 @ForAll 注解。

b) 基于提供器的编码：对于其他每个路径条件，EvoSuite-PBT 都会生成一个带有 @Provide 注解的提供器方法，用于提供 InputVector 对象的 Arbitrary。InputVector 是一种容器对象，可使编码不受参数数量和类型的影响。EvoSuite-PBT 会像上文所述，将π 中的简单约束融入各输入对应的任意值生成器定义，并将每个复杂约束转换为过滤谓词。为得到该谓词，它会将符号约束转换为 Java 表达式，并用对 InputVector 中各元素的引用替换符号变量。

基于提供器的生成器是可靠的，因为 jqwik 仅在采样输入通过所有过滤器后才会使用它。不过，jqwik 会先采样，再验证输入。如果输入未通过过滤器，jqwik 就会拒绝该输入并重新采样，因此，严格的 π 会提高拒绝率，并可能因 TooManyFilterMissesException 异常而中止该性质。对于最严重的情况——等式约束 $x = c$——EvoSuite-PBT 会通过生成 Arbitraries.of(c) 而非使用过滤器来缓解问题，从而将该输入的拒绝率降至零。若要在不使用过滤器的情况下处理更多类型的复杂约束，生成器就会逐渐变成约束求解器 [10], [31], [32]。每次尝试都求解 π 会增加测试执行时间，而求解耗时超过超时限制的性质则无法评估。

为了为实例方法的接收者对象创建一个任意值，EvoSuite-PBT 会选择参数最少的构造函数。EvoSuite-PBT 从 jqwik 任意值生成器中获取构造函数参数，并通过该构造函数映射这些参数。然而，这并不能保证生成器找到满足路径条件的对象；在这种情况下，生成的性质会因 TooManyFilterMissesException 异常而失败。反射可以将任意值生成器直接注入字段，但这可能构造出违反类不变式的对象状态。如何合成符合此类未知不变式的对象状态留待未来工作（第 VII 节）解决。图1（步骤 3）展示了 withdraw 的路径 $p_{2.b}$ 对应的提供器方法，其路径条件为 金额 = 余额。该提供器将一个 double 任意值生成器映射到 BankAccount 构造函数，并将生成的实例与 amount 任意值生成器配对，封装在 InputVector 中。它的过滤谓词对关系路径条件进行编码，并直接读取实例的 balance 字段。EvoSuite-PBT 直接访问非私有字段，并在字段为私有时改用 getter方法。如果私有字段没有 getter方法，EvoSuite-PBT 就无法映射针对该字段的约束，从而导致生成的性质失败。

E. 后置条件编码

后置条件编码会编译 Q_p 为性质中的断言。该性质使用生成的输入调用目标方法，从而得到实际返回值，并在相同输入上计算 ρ，从而得到预期返回值。对于实例方法，方法体还会将调用后每个字段 $\ell \in \text{dom}(\sigma)$ 与 $\sigma(\ell)$ 进行比较，从而完整检查合取式（2）。由于方法体在调用前状态而不是后置状态上计算 ρ 和 $\sigma(\ell)$，因此实现了（1）中的逻辑冻结变量 $\bar{v}$。图1（步骤3）展示了为 withdraw 的路径 $p_{2.b}$ 生成的性质。该性质在调用前计算 $\rho_{2.b}$，并将结果存储在 expected 中。由于 $\sigma_{2.b}$（$balance$）等于 $\rho_{2.b}$，图中将 expected 重用为预期的后置状态 balance。第一个断言将 expected 与 withdraw 的返回值进行比较，第二个断言则将其与后置状态的 balance 字段进行比较。生成的断言会以 0.01 的差值比较双精度值，图中省略了这一点。

生成后置条件语句的核心在于将符号表达式转换为 Java 表达式，EvoSuite-PBT 通过遍历符号执行引擎的表达式树来实现这一过程。访问者根据所选的前置条件编码解析变量引用，引用的对象可以是属性参数，也可以是 InputVector对象。算术、位运算和比较运算符会映射为相应的 Java 运算符；符号执行引擎以 SMT-LIB 风格表示的超越函数运算符和舍入运算符，则会映射为 java.lang.Math 中相应的方法。例如，EvoSuite-PBT 将符号表达式`(sqrt(var0) + 1.0)` 转换为

```
((Math.sqrt(arg0)) + 1.0)
```

基于注解的编码，并且

```
((Math.sqrt(((double) inputVec.get(0)))) + 1.0)
```

在基于提供器的编码下。

这种翻译的两个特征决定了测试预言机的准确性。首先，该转换是部分转换。它仅支持 String 作为引用类型，并将 equals 和 charAt 等字符串操作翻译为 Java。数组和其他引用类型在支持的片段中没有对应表示，因此访问者会发出信号，而不是生成近似结果，并丢弃当前性质。其次，该转换会复现被测方法所执行的 JDK 操作，因此期望值和实际值都来自相同的数值运算。JDK 对 java.lang.Math 的超越函数的规定精度仅达到末位前后一至两个单位，因此，基于此类表达式的性质会继承这种容差，并且在不同 JVM 实现之间可能表现不同。

F. 属性验证和测试套件组装

已编译属性仍可能无法执行：其过滤谓词可能拒绝过多采样输入，或者其生成器可能无法在执行限制内构造有效输入。因此，在生成最终测试套件之前，EvoSuite-PBT 会在 jqwik 中通过执行每个候选属性 10 次来进行验证。在此检查期间失败的任何性质，例如因引发 TooManyFilterMissesException 异常而失败的性质，都会被丢弃，从而避免生成器损坏的性质进入生成的测试套件。

通过此验证检查的性质仅指定了符号执行在目标方法中探索的路径。为了覆盖被测类中其余的方法和路径，EvoSuite-PBT 会针对整个类运行 EvoSuite 的基于搜索的生成器，并将经验证的性质与生成的被测类测试合并，形成最终的回归测试套件。

G. 实现

我们基于 EvoSuite [1]实现了 EvoSuite-PBT。相较于近期工具竞赛 [33]中的其他候选工具，我们选择了 EvoSuite，因为它性能出色、易于获取，而且能够生成 jqwik 所需的 JUnit 5 测试。除了 EvoSuite 内置的动态符号执行引擎外，我们还添加了 Symbolic PathFinder (SPF)，用于评估实例方法。EvoSuite-PBT 以已编译类为输入，并生成 JUnit 5 和 jqwik 源代码。这些源代码无需 EvoSuite-PBT 即可编译和运行。编译和运行这些性质所需的所有运行时辅助程序都包含在一个独立的依赖项 [34]中。我们在下一节所述的评估中使用了这一实现来验证我们的方法。

IV. 评估

我们在两个不同的数据集上评估了 EvoSuite-PBT，以回答以下研究问题：

RQ1（适用性）。EvoSuite-PBT 在多大程度上能够为待测方法生成性质？有哪些因素会限制它？

表 I 评估中使用的基准测试集。LOC 表示 CLOC 报告的代码行数。

Name	描述	LOC
EqBench	等价与不等价程序对	23,939
JSON-java	JSON 编码器和解码器	4,195
async-http-client	异步 HTTP 客户端	15,753
commons-dbutils	JDBC 实用工具类	3,074
commons-geometry	欧几里得几何和球面几何	15,246
commons-jcs	JCache 缓存实现	6,158
commons-numbers	数值工具	5,416
commons-validator	数据验证例程	7,619
commons-vfs2-jackrabbit1	WebDAV 提供程序 Commons VFS	705
http-request	HTTP 请求便捷库	1,391
scribejava-core	OAuth 客户端库	3,771
springside4-metrics	应用指标库	1,213
总计		88,480

研究问题2（有效性）。就测试有效性而言，EvoSuite-PBT 生成的测试套件与 EvoSuite 和 Teralizer 生成的测试套件相比如何?

我们首先在 EqBench [24]，上评估了 EvoSuite-PBT。EqBench 是一个合成数据集，包含执行算术运算的 Java 方法。由于 EqBench 不包含面向对象代码，我们还在 TOGBench [25] 数据集中的 11 个项目上评估了 EvoSuite-PBT。TOGBench 是一个用于自动化测试预言机生成的基准测试集，由 25 个真实世界 Java 项目中开发者编写的单元测试构建而成。我们选择 TOGBench，是因为它于 2026 年发布，且其中所有项目都以 Java 8 为目标平台。我们选取了能够运行全部三种工具的 11 个项目。我们将生成的回归测试套件与两个基线——EvoSuite [1] 和 Teralizer [17]——进行了比较。表 I 列出了所选项目及其规模。

EvoSuite 是第一个基线，因为 EvoSuite-PBT 基于它构建（第 III-G 节），并且对其被测类使用相同的基于搜索的生成器。此比较隔离了性质生成的影响。Teralizer 是第二个基线，因为它也从程序语义中推导性质。它通过单路径符号分析将现有被测类泛化为性质。每次 Teralizer 运行都会对相应 EvoSuite 运行中的被测类进行泛化并保留这些被测类。因此，将 Teralizer 与 EvoSuite 进行比较，可以体现泛化性质对相同被测类故障检测能力的提升。

我们运行了 EvoSuite 和 EvoSuite-PBT，为每个类设置 60 秒的搜索预算，并使用固定随机种子。该预算涵盖 EvoSuite-PBT 的搜索阶段。10次尝试检查随后作为后处理步骤运行（第 III-F 节），不计入搜索预算。我们针对每个数据集分别重复了三次评估。所有评估均在 Ubuntu 24.04.4 LTS x86-64虚拟机上进行，配置为_AMD EPYC 9655 4 核 CPU（2.5GHz）和 16 GB 内存。对于 Teralizer，我们仅启用了其测试泛化功能，并将其配置为使用 IMPROVED算法。随后我们

表II EvoSuite-PBT 在各数据集上的适用性。计数为三次运行的平均值。

	EqBench	TOGBench
方法总数	1,256	7,107
为 SE 选取的方法	1,116	510
SE分析成功	1,116	385
至少具有一个性质的方法	1,062	274
至少具有一个保留属性的方法	1,030	198
生成的性质	5,142	20,745
在10次尝试检查中被丢弃的性质	1,046	20,260
生成过程中无法执行的性质	0	12
生成后剩余的性质	4,096	474
评估期间执行的性质	4,096	358
评估期间失败的性质	860	25

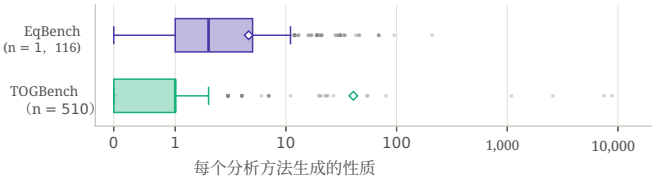

图3。在对称对数尺度上按经SE分析的每个方法统计生成的性质，包括没有性质的方法，线性阈值为1。计数为三次运行的平均值。菱形表示平均值。

分别对两种工具生成的性质执行了 200 次 jqwik 尝试，每次超时设置为 4 秒。该评估的复现实验包已在 Zenodo 上提供 [35]。

A. 适用性

本节通过统计EvoSuite-PBT至少生成一个性质的方法数量，并对未生成任何性质的原因进行分类，来回答RQ1。表II报告了每个数据集的这些计数，数值为三次运行的平均值。图3展示了经SE分析的每个方法所生成性质数量的分布。

a) EqBench： EvoSuite-PBT 为 EqBench 的大多数方法生成了性质。目标选择（第III-B节）从 1,256 个方法中选出了 1,116 个，排除了其余 140 个，因为它们至少违反了四项标准中的一项。符号执行分析了所有目标方法，且未发生错误。EvoSuite-PBT 至少为其中 1,062 个经分析的方法生成了一项性质，而符号执行未能为其余 54 个找到路径。经过10次尝试检查后，1,030 个经分析的方法至少保留了一项性质，占全部方法的 82.0%。

EqBench 中经分析的方法所获性质数量的中位数为 2。一半的方法获得了 1 到 5 项性质。平均而言，每次分析生成 4.6 项性质。离群值对应的方法最多有 210 项性质，这些方法包含循环，因而会产生大量穿过方法的路径。

10次尝试检查舍弃了 EqBench 五分之一的性质。EvoSuite-PBT 为 EqBench 生成了 5,142 个性质。在生成过程中，10次尝试检查舍弃了其中的 1,046 个。这些性质中的大多数因 TooManyFilterMissesException 异常而失败。在进行 200 次尝试的评估中，剩余的 4,096 个性质里有 860 个尽管通过了 10次尝试检查，仍然失败了。其中大多数超过了 4 秒的超时限制。

b) TOGBench： 目标选择和符号执行将 EvoSuite-PBT 的分析范围限制在 TOGBench 项目中少数方法上。目标选择从 7,107 个方法中选出了 510 个，排除了其余 6,597 个，因为它们至少违反了四项标准中的一项。符号执行成功分析了 385 个目标方法。对于其余 125 个方法，符号执行因 JPF 崩溃、约束求解器的局限或插桩测试框架生成失败而未能完成分析。EvoSuite-PBT 至少为 274 个经分析的方法生成了一项性质，而符号执行未能为其余 111 个找到路径。经过 10次尝试检查后，198 个经分析的方法至少保留了一项性质，占全部方法的 2.8%。

TOGBench中大多数被分析的方法最多只获得一个性质。方法获得的性质数量中位数为1。半数方法获得了0到1个性质。单个方法获得的性质最多为9,386个。这些离群值将每次方法分析的平均值提高到了40.7个性质。

10次尝试检查丢弃了TOGBench的几乎所有性质。EvoSuite-PBT为TOGBench项目生成了20,745个性质。在生成过程中，10次尝试检查丢弃了其中的20,260个。由于无效字符串生成器，另有18,002个性质执行失败。还有12个性质无法执行，因为EvoSuite客户端线程在处理其类时崩溃了。这些崩溃导致相关类没有任何测试。在评估期间，剩余的474个性质中有358个成功运行。JUnit执行期间发生的内存溢出错误阻止了另外116个性质的执行。在已执行的性质中，有25个尽管通过了10次尝试检查，却在200次尝试内失败，其中大多数是抛出了TooManyFilterMissesException异常。剩余的333个性质均无错误地通过了执行。

RQ1 的答案。EvoSuite-PBT 为 EqBench 中 95.2% 的待分析方法以及 TOGBench项目中 71.2% 的待分析方法至少生成了一项性质。经过 10次尝试检查后，这些方法中分别有 92.3% 和 51.4% 仍保留性质，分别占全部方法的 82.0% 和 2.8%。在 TOGBench 项目中，目标选择排除了 92.8% 的方法，因为这些方法至少违反了其四项标准中的一项。由于 JPF 崩溃、约束求解器的局限性，或无法生成插桩测试框架，SE 在 24.5% 的目标方法上失败。在生成的性质中，有 97.7% 未通过 10次尝试检查，主要原因是无效字符串生成器。

B. 有效性

本节通过变异分析 [36],[37] 回答研究问题2。变异分析会在程序中注入称为变异体的小型语法变化，并检查测试套件能否区分每个变异体与原始程序。它假设程序员编写的程序几乎正确，并且能够检测简单故障的测试也能检测复杂故障，这一假设称为耦合效应 [36], [37]。我们使用 PIT [38]，并采用其

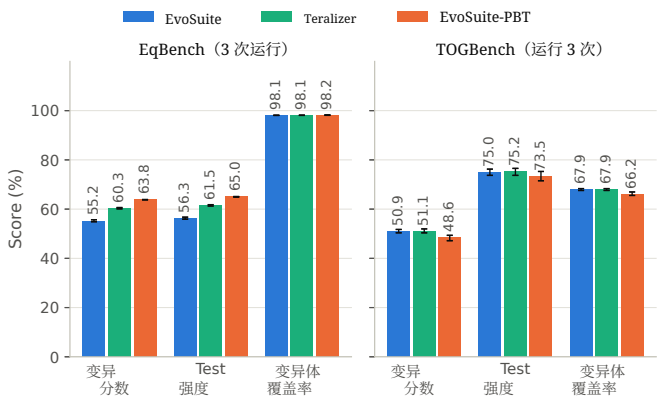

图 4. EvoSuite、Teralizer 和 EvoSuite-PBT 在 EqBench 和 TOGBench 上的平均变异得分、测试强度和变异体覆盖率。误差线表示三次运行中的最小值和最大值。

默认变异算子。PIT 以确定性方式生成变异体，因此三种工具面对的是同一组变异体。PIT 首先在未经变异的字节码上运行所有测试，排除未通过的测试，并记录剩余测试各自覆盖了哪些变异体。然后，它只针对覆盖每个变异体的测试运行该变异体。如果某个剩余测试的行为发生变化（例如测试失败、超时，或导致测试 JVM 耗尽资源或崩溃），则该测试会杀死该变异体。变异得分统计所有生成的变异体，包括未被覆盖的变异体：

$$MS = \frac{\#killed}{\#total}. \tag{4}$$

为了仅测量对已执行代码的故障检测，我们还报告了 PIT 的测试强度 [38]，它对应于 Inozemtseva 和 Holmes 的归一化有效性 [3]：

$$TS = \frac{\#killed}{\#covered}. \tag{5}$$

图 4 展示了三种工具在两个数据集上的变异得分、测试强度和变异体覆盖率。两个基线达到的变异体覆盖率几乎相同，因为每个 Teralizer 套件都保留了其 EvoSuite 输入中的被测类。

a) EqBench：EvoSuite-PBT 在 EqBench 上生成的测试套件比两个基线都更有效。其平均变异得分为 63.8%，平均测试强度为 65.0%，分别比 EvoSuite 高 8.6 和 8.7 个百分点。与 Teralizer 相比，这两项指标均提高了 3.5 个百分点。

这些提升并非源于更广的覆盖率。在 EqBench 上，EvoSuite-PBT 的变异体覆盖率为 98.2%，两个基线的变异体覆盖率均为 98.1%。由于这三个测试套件覆盖的变异体几乎相同，EvoSuite-PBT 更高的测试强度意味着其测试能够杀死更多这些变异体。下面的统计表明，其性质解释了这一差异。我们统计了每次运行某工具都能杀死、但 EvoSuite 的任何一次运行都无法杀死的变异体数量。EvoSuite-PBT 杀死了 2,259 个此类变异体，是 Teralizer 的 1,169 个的 1.9 倍。在平均每次运行中，其中有 2,188 个变异体会导致 EvoSuite-PBT 的测试失败，而对于其中 66.9% 的变异体，检测它们的测试是一个性质。EvoSuite-PBT 的性质在 504 个类中的 328 个类里杀死了变异体，而 Teralizer 的泛化性质则在其中 150 个类里做到了这一点。

对于至少有一个性质通过 10次尝试检查的类，EvoSuite-PBT 的领先优势进一步扩大。在这 504 个类中的 447 个类上，EvoSuite-PBT 达到 66.5% 的变异得分，而 EvoSuite 和 Teralizer 分别为 57.4% 和 62.8%。因此，其相对于 EvoSuite 的领先优势扩大到 9.2 个百分点，相对于 Teralizer 则扩大到 3.7 个百分点。在其余 57 个类上，三个工具的变异得分均介于 23.3% 和 23.4% 之间。

按变异算子统计，EvoSuite-PBT 在 EqBench 的九个算子中有八个取得了最高的变异得分。其最大增益出现在 Math算子上，该算子替换算术运算符，并产生了 23,961 个变异体中的 15,317 个。对于该算子，EvoSuite-PBT 比 EvoSuite 高 10.8 个百分点，比 Teralizer 高 3.7 个百分点。它仅在 160 个 VoidMethodCall变异体上低于 EvoSuite，得分分别为 1.0% 和 2.3%。

EvoSuite-PBT 在 EqBench 上的领先优势完全体现在静态方法上，它通过动态符号执行对这些方法进行探索（第 III-C 节）。EqBench 的 23,961 个变异体中，有 23,794 个来自这些静态方法。针对这些变异体，EvoSuite-PBT 的变异得分达到 64.3%，而 EvoSuite 和 Teralizer 分别为 55.6% 和 60.7%。

b) TOGBench：在 TOGBench 项目上，EvoSuite-PBT 的效果不及两个基线。其平均变异得分为 48.6%，比 EvoSuite 低 2.3 个百分点，比 Teralizer 低 2.5 个百分点。其平均测试强度为 73.5%，分别低 1.5 和 1.8 个百分点。其变异体覆盖率为 66.2%，比两个基线的 67.9% 低 1.7 个百分点。与 Teralizer 相比，EvoSuite-PBT 每次运行少杀死 494 个变异体，相当于相对下降 4.9%。

六个类造成了这一差距。它们使 EvoSuite-PBT 每次运行少杀死 494 个变异体，超过其相对于 EvoSuite 的 459 个净差距。没有这六个类时，EvoSuite-PBT 将比 EvoSuite 高出 0.2 个百分点。对于 JSON-java 和 commons-validator，我们不得不提高 EvoSuite 的客户端超时，以便用于 EvoSuite-PBT，因为后者涵盖了后处理，否则其中一些类将没有测试。即便如此，日志仍显示这六个类中有五个出现了生成失败。BigFraction 的测试生成在三次运行中的两次发生崩溃，而 ArithmeticUtils 的断言生成在所有运行中都发生了超时。NtlmEngine 的测试会断言一个随机挑战值，因此它们在未变异的程序上也会失败，PIT 会将其排除。这些失败影响的是被测类，而 EvoSuite-PBT 会在自己的搜索过程中生成这些被测类；相比之下，每次 Teralizer 运行都会重用相应 EvoSuite 运行中的被测类。

性质对 TOGBench 的贡献很小。目标选择、符号执行限制和 10次尝试检查使剩余性质仅涉及所有方法的 2.8%（IV-A 节）。因此，EvoSuite-PBT 的性质每次运行可杀死 93 个变异体，分布在 831 个类中的 21 个类中。Teralizer 的泛化性质覆盖的类更少，每次运行在 9 个类中杀死 17 个变异体。尽管如此，Teralizer 的变异得分仍比 EvoSuite 高 0.2 个百分点，相当于每次运行额外杀死 35 个变异体。

EvoSuite-PBT 在 TOGBench 上的不足主要源于没有保留属性的类。EvoSuite-PBT 为 831 个类中的 69 个类保留了一个性质，这些类包含 3,867 个变异体。由于 PIT 会分别对内部类进行变异，这 69 个类的内部类中的 496 个变异体被计入其余 762 个类。在这 69 个类中，EvoSuite-PBT 的变异得分为 63.5 %，比 EvoSuite 低 0.5 个百分点，比 Teralizer 低 1.5 个百分点。在其余 762 个类中，它比 EvoSuite 低 2.8 个百分点。

在 TOGBench 上，EvoSuite-PBT 在 11 个算子中的 9 个上得分低于 EvoSuite。两个例外是 BooleanFalseReturnVals 和 ConditionalsBoundary，前者提升了 1.5 个百分点，后者提升了 0.5 个百分点。其最大劣势出现在 Math 上，得分为 51.9%，而 EvoSuite 和 Teralizer 分别为 59.4% 和 60.7%。对于所有其他算子，Teralizer 与 EvoSuite 的变异得分相差至多 0.2 个百分点。

在 TOGBench 上，EvoSuite-PBT 在静态方法上的劣势小于在实例方法上的劣势。在静态方法上，EvoSuite-PBT 的得分比 EvoSuite 低 1.3 个百分点；在实例方法上则低 3.5 个百分点。在静态方法中，性质是 2.9% 的被杀死变异体的首个杀死测试；在实例方法中，这一比例为 0.3%。

与 Teralizer 的比较体现了从被测方法中提取路径语义与泛化现有CUT之间的权衡 [17]。在 EqBench 和 commons-numbers 中，当 SE 能够对方法进行建模时，直接提取能够覆盖分支覆盖型被测类未遍历的路径和边界条件，并且不需要将断言映射到方法调用。相比之下，测试泛化会复用现有被测类中的有效接收者对象和具体输入，而直接提取必须在 SE 过程中构造这些对象和值 [19], [28]。

研究问题2的回答。在 EqBench 上，EvoSuite-PBT 生成了最有效的测试套件，平均变异得分为 63.8%，平均测试强度为 65.0%。这些数值分别比 EvoSuite 高出 8.6 和 8.7 个百分点，比 Teralizer 各高出 3.5 个百分点。其性质是取得这一领先优势的原因。在具有保留属性的类中，它相对于 EvoSuite 的领先优势扩大到 9.2 个百分点。在 TOGBench 上，EvoSuite-PBT 的变异得分比 EvoSuite 和 Teralizer 分别低 2.3 和 2.5 个百分点。测试生成期间发生的失败导致了这一差距，因为在损失最大的六个类中，有五个类因此没有有效测试。所有方法中仅有 2.8% 保留了性质，因此这一差距主要源于不具有保留属性的类。

V. 讨论

本节解读评估结果，并评估其泛化程度。随后，阐述对开发者和研究人员的启示，并讨论效度威胁。

A. 启示

a) 对开发者的启示：EvoSuite-PBT 可帮助开发者增强执行算术运算的方法的回归测试套件。在 EqBench 以及 commons-numbers 的 gamma 和 combinatorics 模块中，TOGBench 上，它使变异得分比 EvoSuite 提高了 5.1 到 10.2 个百分点。它的性质源自源代码，而不是测试，因此开发者可以将其添加到现有测试套件中，或使用它们测试尚未覆盖的代码。每个性质都对当前实现的行为进行了编码，包括其中的故障。因此，开发者在依赖任何性质之前，应先根据预期行为对其进行检查。这项工作使他们不必手动编写后置条件和输入生成器，而开发者认为这很困难 [14], [15]。

b) 对研究人员的启示：在 EqBench 上，多路径符号执行比 Teralizer 的单路径泛化高出 3.5 个百分点，因为它能够指定被测类均未遍历的路径。因此，只要符号执行能够对被测方法建模，基于性质的回归测试研究就应直接从源代码中推导性质。对于面向对象代码，有效对象的生成限制了这一方法。在 TOGBench 上，目标选择排除了 92.8% 的方法，部分原因是它只接受基本类型和 String 参数。在生成的性质中，97.7% 未能通过 10次尝试检查，这主要是由于无效字符串生成器所致。测试泛化绕过了对象构造，因为它以已经能够构造有效接收者对象的被测类为起点。而直接提取则需要相关技术，以便在符号执行期间以符号方式构造有效对象，并在输入生成器中以具体方式构造有效对象。对这些技术的研究将使直接提取能够应用于面向对象代码（第 VII 节）。

B. 效度威胁

a) 内部效度：基于搜索的测试生成受时间限制，因此挂钟时间调度会导致不同运行之间存在非确定性。为缓解这一威胁，我们对每个工具在每个数据集上使用固定随机种子和每个类 60 秒限制，重复评估三次，因此三次运行之间的差异仅来自挂钟时间的搜索时长。

b) 外部效度：我们实现的 EvoSuite-基于属性的测试面向 Java，因此将我们的实证发现限制在 Java 程序范围内。该方法本身也适用于其他提供符号执行引擎和基于属性的测试框架的语言。另一个威胁是，EqBench 是由算术方法构成的合成数据集，因此在该数据集上的结果可能无法推广到真实世界代码。为缓解这一威胁，我们还在来自 TOGBench 的 11 个真实世界 Java 项目上评估了全部三个工具，这些项目包含面向对象代码。

c) 构念效度：构念效度面临的一个威胁是，变异分数和测试强度是否能够反映对真实故障的检测效果。我们通过使用 PIT 进行变异分析来缓解这一威胁 [38]，PIT 的变异体已被证明能够揭示 Java 程序中的真实故障 [39]。

VI. 相关工作

以往的工作通过推断符号约束或重构相似测试，将传统单元测试抽象为参数化单元测试 [27] 。Fraser 和 Zeller [40] 用符号参数替换生成的被测类中的具体输入，并通过变异分析推断前置条件和后置条件。AutoPUT [7] 以及

Azamnouri 和 Paydar [8] 则将相似的被测类合并为基于其字面值的 JUnit 理论测试或参数化测试。尽管这些方法依赖于变异分析或固定的字面值，我们的方法直接根据源代码路径语义推导全称量化的性质。

其他方法从运行时执行轨迹构造参数化测试，而非仅依赖静态测试代码。Tiwari 等人 [9]提出了 PROZE，该方法通过插桩测试执行来记录目标方法的运行时参数值，将每个仅含一个断言的被测类转换为 JUnit 5 参数化测试，并通过方法源使用观测到的参数元组作为数据，再检查现有断言对这些参数元组是否均成立。Kampmann 和 Zeller [41]通过记录函数参数和全局变量写入，从系统测试运行中提取单元级参数化测试；他们将系统输入子串对应的参数设为可变参数，在单元级对这些参数进行模糊测试，并将导致崩溃或提升覆盖率的输入回传至系统级。PROZE 会针对现有断言重放一组固定的已记录参数元组，而系统测试提取方法会对提取出的参数进行模糊测试，却不提供单元级后置条件测试预言机；与之不同，我们的方法通过符号执行，同时推导出特定路径的输入生成器和后置条件断言。

规格挖掘方法从程序执行中推断前置条件、后置条件和对象不变量，而不是直接生成参数化测试。Ernst 等人 [42]介绍了 Daikon：它通过插桩处理程序，观察测试套件中各变量的取值，并报告在已观察到的运行中始终成立的候选不变量模板。由于基于模板的推断依赖测试套件的覆盖率，稀疏的执行轨迹可能产生过于严格的不变量，反映出测试套件伪影。Csallner 等人 [43] 在 DySy 中通过将符号执行与具体测试运行并行进行，针对已执行的路径推导路径条件和符号后置条件，从而解决这一局限。然而，Daikon 和 DySy 都只根据现有测试套件的执行来推断规格，不会将推断出的三元组编译为可执行的性质生成器。

近期研究通过 API 文档或泛化单元测试来自动创建基于性质的测试 [16],[17], [44]。Vikram 等人 [44]使用自然语言 API 文档提示大语言模型，在没有现有单元测试的情况下合成生成器和性质断言，但当文档存在歧义时，模型生成的测试可能无法编译，或断言不健全的性质。Peleg 等人 [16]则记录重复执行被测类时的具体执行轨迹，折叠重复的调用序列，并使用 SMT 求解器实例化预定义的抽象模板。由于模板实例化会对观测到的示例进行过度近似，稀疏的示例集可能会产生在有效输入上失败的性质。Glock 等人 [17]在 Teralizer 中通过沿每个被测类所执行的具体路径进行单路径符号分析，将现有或生成的被测类泛化为性质。由于测试泛化从被测类出发，因此只能为被测类已经遍历的执行路径，以及其断言所检查的行为生成性质。Teralizer 还要求将每个测试断言映射到单个被测方法调用，并且仅支持不含字段更新的纯非 void 方法。相比之下，我们的方法通过多路径符号执行直接提取目标方法的路径语义，并在诱导的霍尔三元组中对实例字段更新进行建模。

VII. 结论

本文提出了 EvoSuite-PBT，这是一种自动化测试生成方法，通过多路径符号执行直接从源代码语义中推导基于属性的回归测试，并将其与基于搜索的被测类相结合。对于穿过目标方法的每条已探索路径，EvoSuite-PBT 都会记录路径条件、符号返回值和实例字段状态变化，将其作为路径语义，并据此构造一个全正确性霍尔三元组。每个三元组都会编译为一个可执行的 jqwik 性质，其前置条件通过范围注解或过滤谓词约束输入生成，后置条件则断言返回值和已修改字段。通过生成时验证检查的性质会与 EvoSuite 生成的全类被测类相结合，以覆盖符号执行未能说明的行为。

在 EqBench 和 11 个 TOGBench 项目上，将 EvoSuite-PBT 与 EvoSuite 和 Teralizer 进行评估后发现，直接从方法的路径语义推导性质，能够提升执行算术运算的方法的测试效果。在 EqBench 上，平均变异得分升至 63.8 %，高于 EvoSuite 的 55.2% 和 Teralizer 的 60.3%。在 TOGBench 的数值模块上，其变异得分较 EvoSuite 最多提高了 10.2 个百分点。然而，在全部 11 个 TOGBench 项目中，目标选择、符号执行限制和属性验证使经验证的属性仅覆盖了所有方法的 2.8%。

未来，我们计划解决 EvoSuite-PBT 的三项局限。第一，我们计划改进对引用参数和字段的有效对象状态的合成 [19], [28] ，以支持具有原始类型和 String类型签名以外签名的方法。第二，我们将用约束感知生成器替代拒绝采样 [31], [32] ，以减少 TooManyFilterMissesException异常导致的失败和评估超时。第三，我们将采用测试套件缩减和参数化 [7], [8], [17] ，合并那些因经验证的属性而变得冗余的被测类。

数据可用性

本文的复现实验包已发布于 Zenodo [35]。生成的性质所需的运行时辅助程序作为单独的依赖项提供 [34]。

致谢

本研究全部或部分由奥地利科学基金（FWF）资助，项目编号为 10.55776/P36698。为实现开放获取，作者已对本次投稿产生的任何作者接受稿版本采用 CC BY 公共版权许可。本文报告的研究部分由 BMIMI、BMWET 和上奥地利州

资助，资助框架为 SCCH 能力中心 INTEGRATE（FFG 资助编号 892418），该中心属于 FFG COMET卓越技术能力中心计划。我们使用 OpenAI GPT、Anthropic Claude 和 Google Gemini 模型辅助写作、编程和文献检索。作者审阅并完善了生成的内容，并对最终稿件负全部责任。

参考文献

[1] G. Fraser 和 A. Arcuri，《EvoSuite：面向对象软件的自动测试套件生成》，载于第 *19* 届 *ACM SIGSOFT* 软件工程基础研讨会暨第 *13* 届欧洲软件工程会议（*ESEC/FSE*）论文集。ACM，2011 年，第 416–419 页。[2] C. Pacheco 和 M. D. Ernst，《Randoop：面向 Java 的反馈导向随机测试》，载于第 *22* 届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用会议（*OOPSLA*）附属会议论文集。ACM，2007 年，第 815–816 页。[3] L. Inozemtseva 和 R. Holmes，《覆盖率与测试套件有效性并无强相关性》，载于第 *36* 届 *IEEE/ACM*软件工程国际会议（*ICSE*）论文集。ACM，2014 年，第 435–445 页。[4] H. Hemmati，《代码覆盖率标准有多有效？》，载于*2015* 年 *IEEE* 软件质量、可靠性与安全国际会议（*QRS*）论文集。IEEE，2015 年，第 151–156 页。[5] J. M. Voas，《PIE：一种动态故障驱动技术》，*IEEE* 软件工程汇刊，第 18 卷，第 8 期，第 717–727 页，1992 年。[6] R. M. Hierons，《在边界值分析中避免偶然正确性》，*ACM* 软件工程与方法论汇刊，第 15 卷，第 3 期，第 227–241 页，2006 年。[7] K. Tsukamoto、Y. Maezawa 和 S. Honiden，《 AutoPUT：一种将封闭式单元测试改造为参数化单元测试的自动化技术》，载于第 *33* 届 *ACM* 应用计算年度研讨会（*SAC*）论文集。ACM，2018 年，第 1944–1951 页。[8] A. Azamnouri 和 S. Paydar，《通过测试参数化压缩自动生成的单元测试套件》，载于第 *9* 届软件工程基础国际会议（*FSEN*）论文集。Springer，2021 年，第 215–221 页。[9] D. Tiwari、Y. Gamage、M. Monperrus 和 B. Baudry，《PROZE：利用运行时数据生成参数化单元测试》，载于 *2024* 年 *IEEE* 源代码分析与处理国际会议（*SCAM*）论文集。IEEE，2024 年，第 166–176 页。[10] K. Claessen 和 J. Hughes，《 QuickCheck：用于 Haskell 程序随机测试的轻量级工具》，载于第 *5* 届 *ACM SIGPLAN* 函数式编程国际会议（*ICFP*）论文集。ACM，2000 年，第 268–279 页。[11] J. Hughes，《QuickCheck 测试：乐趣与收益》，载于第 *9* 届声明式语言实用方面国际研讨会（*PADL*）论文集。Springer，2007 年，第 1–32 页。[12] S. Ravi 和 M. Coblenz，《 Python 中基于属性的测试的实证评估》，*ACM* 编程语言期刊论文集，第 9 卷，OOPSLA2 期，第 3897–3923 页，2025 年。[13] C. Wauters、R. Opdebeeck 和 C. De Roover，《Python 测试用例向基于属性的测试的演进》，载于第 *19* 届 *IEEE* 软件测试、验证与确认国际会议（*ICST*）论文集。IEEE，2026 年，第 429–440 页。[14] H. Goldstein、J. W. Cutler、D. Dickstein、B. C. Pierce 和 A. Head，《基于属性的测试实践》，载于第 *46* 届 *IEEE/ACM*软件工程国际会议（*ICSE*）论文集。ACM，2024 年，第 187:1–187:13 页。[15] A. L. Corgozinho、M. T. Valente 和 H. Rocha，《开发者如何实现基于属性的测试》，载于第 *39* 届 *IEEE* 软件维护与演进国际会议（*ICSME*）论文集。IEEE，2023 年，第 380–384 页。[16] H. Peleg、D. Rasin 和 E. Yahav，《通过示例生成测试》，载于第 *19* 届验证、模型检查与抽象解释国际会议（*VMCAI*）论文集。Springer，2018 年，第 406–429 页。[17] J. Glock、C. Bauer 和 M. Pinzger，《Teralizer：将传统单元测试泛化为基于属性的测试的语义驱动方法》，arXiv 预印本 arXiv:2512.14475，2025 年。

[18] J. C. King，《符号执行与程序测试》，《ACM通信》，第19卷，第7期，第385–394页，1976年。[19] R. Baldoni、E. Coppa、D. C. D’Elia、C. Demetrescu和I. Finocchi，《符号执行技术综述》，《ACM计算综述》，第51卷，第3期，第50:1–50:39页，2018年。[20] S. Person、M. B. Dwyer、S. Elbaum和C. S. Pˇasˇareanu，《差分符号执行》，载于第16届ACM SIGSOFT国际软件工程基础研讨会（FSE）论文集。ACM，2008年，第226–237页。[21] C. A. R. Hoare，《计算机编程的公理化基础》，《ACM通信》，第12卷，第10期，第576–580页，1969年。[22] J. P. Galeotti、G. Fraser和A. Arcuri，《通过自适应动态符号执行扩展基于搜索的测试生成器》，载于2014年ACM SIGSOFT国际软件测试与分析研讨会（ISSTA）论文集。ACM，2014年，第421–424页。[23] C. S. Pˇasˇareanu和N. Rungta，《Symbolic PathFinder：Java字节码的符号执行》，载于第25届IEEE/ACM国际自动化软件工程会议（ASE）论文集。ACM，2010年，第179–180页。[24] S. Badihi、Y. Li和J. Rubin，《EqBench：等价与非等价程序对数据集》，载于第18届IEEE/ACM软件仓库挖掘国际会议（MSR）论文集。IEEE，2021年，第610–614页。[25] T. Tasnim、M. B. Dwyer和S. B. Hossain，《TOGBench制品：AIware-2026》，Figshare数据集，2026年。[26] P. C. Jorgensen和B. DeVries，《软件测试：工匠方法》，第5版。CRC出版社，2021年。[27] N. Tillmann和W. Schulte，《参数化单元测试》，载于第10届欧洲软件工程会议与第13届ACM SIGSOFT国际软件工程基础研讨会联合会议（ESEC/FSE）论文集。ACM，2005年，第253–262页。[28] S. Anand、C. S. Pˇasˇareanu和W. Visser，《JPF–SE：Java PathFinder的符号执行扩展》，载于第13届系统构造与分析工具和算法国际会议（TACAS）论文集。Springer，2007年，第134–138页。[29] J. P. Galeotti、G. Fraser和A. Arcuri，《通过动态符号执行改进基于搜索的测试套件生成》，载于第24届IEEE国际软件可靠性工程研讨会（ISSRE）论文集。IEEE，2013年，第360–369页。[30] M. Aniche，《有效软件测试：开发者指南》。Manning出版社，2022年。[31] K. Claessen、J. Duregård和M. H. Pałka，《生成均匀分布的受约束随机数据》，《函数式编程期刊》，第25卷，第e8:1–e8:31页，2015年。[32] L. Lampropoulos、D. Gallois-Wong、C. Hri¸tcu、J. Hughes、B. C. Pierce和L.-y. Xia，《Beginner’s Luck：一种用于基于属性的生成器的语言》，载于第44届ACM SIGPLAN编程语言原理研讨会（POPL）论文集。ACM，2017年，第114–129页。[33] F. Kifetew、L. Yun和D. Prandi，《SBFT工具竞赛2025——Java测试用例生成赛道》，arXiv预印本arXiv:2504.09168，2025年。[34] 匿名，《evosuite-pbt-utils》，匿名源代码仓库，2026年，访问日期：2026-11-02。 [在线]。网址：https://anonymous.4open.science/r/evosuite-pbt-utils-6367/ [35] A. Anonymous，《“根据源代码语义生成基于属性的回归测试”的复现实验包》，2026年11月。 [在线]。网址：https://doi.org/10 5281 23976317 。/zenodo。R. [36] A. DeMillo、R. J. Lipton和F. G. Sayward，《测试数据选择提示：为从业程序员提供帮助》，《计算机》，第11卷，第4期，第34–41页，1978年。[37] T. A. Budd、R. J. Lipton、F. G. Sayward和R. A. DeMillo，《用于程序测试的原型变异系统设计》，载于1978年AFIPS全国计算机会议（NCC）论文集。AFIPS出版社，1978年，第623–627页。[38] H. Coles、T. Laurent、C. Henard、M. Papadakis和A. Ventresque，《PIT：实用的Java变异测试工具（演示）》，载于第25届ACM SIGSOFT国际软件测试与分析研讨会（ISSTA）论文集。ACM，2016年，第449–452页。[39] M. Kintis、M. Papadakis、A. Papadopoulos、E. Valvis、N. Malevris和Y. Le Traon，《变异测试工具的效果如何？对Java变异测试工具的实证分析：人工分析与真实故障》，《实证软件工程》，第23卷，第2463页，2018年，第4期，第2426页。

[40] G. Fraser和A. Zeller，《生成参数化单元测试》，载于*2011*年*ACM SIGSOFT*国际软件测试与分析研讨会（*ISSTA*）论文集。ACM，2011年，第364–374页。[41] A. Kampmann和A. Zeller，《提取参数化单元测试》，载于第*41*届*IEEE/ACM*软件工程国际会议：伴随论文集（*ICSE-Companion*）。IEEE，2019年，第248–249页。[42] M. D. Ernst、J. H. Perkins、P. J. Guo、S. McCamant、C. Pacheco、M. S. Tschantz和C. Xiao，《用于动态检测可能不变式的Daikon系统》，《计算机编程科学》，第69卷，第1–3期，第35–45页，2007年。[43] C. Csallner、N. Tillmann和Y. Smaragdakis，《DySy：用于不变量推断的动态符号执行》，载于第*30*届*ACM/IEEE*软件工程国际会议（*ICSE*）论文集。ACM，2008年，第281–290页。[44] V. Vikram、C. Lemieux、J. Sunshine和R. Padhye，《大语言模型能编写优质的基于属性的测试吗？》，arXiv预印本arXiv:2307.04346，2024年。