arXiv:2601.16008v1 [cs.PL] 22 Jan 2026

通过基于编译器的精细特征排序，优先考虑配置相关性

Federico Bruzzone 米兰大学，
意大利米兰**Walter Cazzola** 米兰
大学，意大利米兰

Luca Favini
米兰大学，意大利米兰

摘要

现代编程语言，尤其是 Rust，提供了先进的语言构造，可将特征聚合起来构建高度可配置的软件系统，并通过配置标识这些特征。然而，这些语言也给程序分析、优化和测试带来了重大挑战，因为配置的组合爆炸往往使穷举探索难以实现。在本文中，我们提出了首个用于配置优先级排序的基于编译器的方法。我们的方法包含四个主要步骤：**1.** 从 Rust 编译器中提取定制的中间表示；**2.**构建两种互补的基于图的数据结构；**3.** 使用中心性度量对特征进行排序；以及**4.** 结合特征影响的代码范围，对排序结果进行精细化。随后，根据得到的特征排序生成固定数量的最相关配置。通过使用 SAT 求解器，并向其提供该图的合取范式表示，可确保生成配置的有效性。我们将此方法形式化，并通过对 Rust 编译器进行插桩，将其实现为原型RustyEx。针对排名较高的开源 Rust 项目进行的实证评估表明，RustyEx 能够在资源受限的情况下高效生成用户指定的配置集合，同时从构造上保证可靠性。结果表明，基于中心性指导的配置优先级排序能够有效且实用地探索大型配置空间，为配置感知分析与优化的未来研究铺平道路。

2012 年 ACM 学科分类 软件及其工程 →编译器；软件及其工程 → 软件设计工程；软件及其工程 → 正确性

关键词与短语 高度可配置系统、可变系统测试、基于静态分析的配置优先级排序、Rust。

数字对象标识符 10.4230/LIPIcs.CVIT.2016.23

补充材料 可在 Zenodo 上获取复现材料包。软件：
`https://doi.org/10.5281/zenodo.17691776`

经费支持 本研究部分由意大利大学与研究部项目“T-LADIES”（PRIN 2020TL3X8X）资助。

1 引言

前提。 高度可配置软件系统[1] 通常旨在满足广泛的需求。这些系统提供一组可通过不同方式组合的特征，以创建各种产品 [4]。可变性丰富的软件系统开发借鉴了产品线工程的原则，通常称为面向特征的编程 [78]。C/C++和 Java 等编程语言提供了通过预处理器指令、条件编译和注解来管理软件可变性的机制。

[1]这些系统也称为产品族或软件产品线（SPLs） [21]

© Federico Bruzzone、Walter Cazzola 和 Luca Favini；依据知识共享署名 4.0 国际许可协议授权 第42届重要主题会议（CVIT 2016）。编辑：John Q. Open 和 Joan R. Access；文章编号：23；页码：23:1–23:29 莱布尼茨国际信息学论文集 达格施图尔城堡—莱布尼茨信息学中心，达格施图尔出版社，德国

Rust [59] 也旨在通过其属性系统[2]支持高度可配置软件系统的开发，具体而言，是通过 `cfg` 属性实现的。

问题陈述。 软件可变性可能非常复杂，因此难以推理所有可能的配置 [1]。正如 Krueger 所指出的，可能配置的数量会随着特征数量的增加而呈指数级增长，这进一步加剧了这一问题 [45]。例如，Linux 内核是一个著名的高度可配置软件系统，具有大量特征 [85] ，它利用了 C/C++以及近来 Rust 对可变性的原生支持。多年来，内核及类似系统得到了广泛研究，许多研究都指出了大型配置空间带来的挑战 [62, 91, 27, 58]。

研究现状 由于配置数量的组合爆炸，穷举方法已难以实施 [5]。因此，需要一种策略来减少待分析的配置数量。多种方法都旨在实现这一目标。采样技术 [73, 2, 47], 、组合测试 [71, 54], , 尤其是 t-wise 测试 [22, 75, 71]，试图在保证覆盖率的同时缩小配置空间。其他研究提出了优先级排序方法 [29, 81, 72, 27], , 采用相似性 [2], 、非功能属性 [80], 或基于中心性的度量 [63, 74, 49, 6]等标准。正如 Classen等人 [20], 所报告的，许多可配置系统都是安全关键系统，因此优先级排序更加重要：目标是在减少测试数量的同时，尽可能早地检测出故障。

现有方法的局限性。尽管做出了这些努力，如何找到恰当的优先级划分标准仍是一个悬而未决的问题。现有方法**1.** 仅适用于特征模型， [9, 77], 未考虑代码中的依赖关系，且**2.** 未考虑每个特征影响的代码范围。因此，可能会遗漏对系统行为至关重要的配置。此外，对有原则的配置优先级排序的需求远不止于测试。高度可配置系统还必须应对：

1. 编译器优化和性能分析，在这些场景中，要么只能切实评估部分变体，要么可以利用配置相关性指导优化策略 [94,84, 90, 95],**2.** 程序理解和调试，开发人员需要对具有代表性或关键性的配置进行推理 [37, 96],**3.** 可变性管理和重构，在这些场景中，结构性变更应以特征在代码中的实际影响为指导 [53, 52], 和**4.** 回归分析，在这些场景中，优先级划分有助于确定系统演进后哪些配置最有可能揭示行为差异 [79, 88]。

在所有这些情境中，将特征一视同仁或依赖随机启发式方法都不足以解决问题。

提案。 据我们所知，本文提出了首个通用方法，通过基于编译器的特征精化排序，优先考虑配置的相关性。为此，我们推出了 RustyEx，这是一款全自动工具，旨在从高度可配置的 Rust 软件系统中识别最相关的配置。[3] 我们将

[2]Rust 的属性是一种语法元数据，可以附加到代码的各个部分。Rust 支持两种属性：外部 和内部。前者用于标注项 声明（例如函数、结构体和枚举）、表达式 和语句。后者用于标注模块 和块表达式 （特定情况下）。我们将项、表达式和语句 统称为项。更多详情请参阅 `https://doc.rust-lang.org/reference` 。[3]尽管本文使用 Rust，但我们提出的配置优先级排序方法与语言无关，适用于任何提供原生可变性支持的语言，例如 C/C++ 或 Java。

这些配置定义为包含最相关特征的配置。特征的相关性通过其在特征依赖图中的中心性，以及其影响的代码范围来衡量。如图 1 所示，我们通过执行过程间静态分析来插桩 Rust 编译器， [68, 69][4] 从而提取统一中间表示（UIR）。UIR 通过从 Rust 抽象语法树（AST）中移除无关信息并创建原子而得到。[5] RustyEx 对 UIR 执行静态分析，以提取两种主要数据结构：依赖图 [60] 和多树 [25]。前者是一个加权有向图，称为特征依赖图（见图 2 中的 ❷ 和 ❷），用于表示特征之间的依赖关系。后者是 UIR 的一个诱导子图，称为原子依赖树（见图 2 中的 ❸ 和 ❸）。该结构捕捉基于词法作用域的原子间依赖关系，并结合这些原子所影响的代码范围。为解决前文所述的局限性，我们提出了一种结合这两种结构来识别最相关配置的方法。该方法利用中心性度量作为结构度量指标 [30, 61, 65], ，并将图转换为命题公式和合取范式（CNF） [20]公式，同时采用特征排序精化技术。RustyEx 依靠 SAT 求解器，根据精化后的特征排序确定满足 CNF 公式的配置。这确保不仅减少配置数量，还能优先考虑最相关的配置。我们在排名较高的开源 Rust 项目上验证了该方法：运行 RustyEx，并将生成配置的数量设为固定值。为全面评估，我们报告了各项目所提结构的详细度量指标。该方法在构造上保证可靠性，因此所有生成的配置均有效。

贡献。 本文的主要贡献如下：

- 首个通过基于编译器的特征精化排序，为高度可配置软件系统中的配置确定优先级的通用方法；对该方法进行形式化，从 UIR 的提取到构建互补数据结构及配置优先级排序的
- 算法；在 RustyEx 中对我们的方法进行详细实现。RustyEx 是一款用于 Rust 软件的全
- 自动工具；在排名靠前的开源 Rust 项目上对 RustyEx 进行了广泛评估，展示了它在识
- 别最相关配置并确定其优先级方面的有效性；以及对我们方法可靠性的形式化证明，确
- 保所有生成配置均有效。
-

论文结构。 本文其余部分的组织如下。第 2 节介绍必要的背景知识。第 3 节详细介绍 RustyEx 的设计，第 4 节展示我们的评估。第 6 节讨论相关工作，第 7 节总结全文。

2 背景

我们介绍 Rust 及其所有权系统的背景，以及中心性度量。

[4]过程间分析是一种静态分析技术，用于分析程序在多个函数间的控制流和数据流。[5]从现在起，我们将带有给定配置谓词注释的单个原子 称为一个项 。

2.1 Rust 编程语言。

Rust 是一种系统编程语言，专注于安全性、速度和并发性。它无需垃圾回收即可确保内存安全，这意味着纯 Rust 程序不会出现空指针解引用和数据竞争。Rust 的所有权系统集成在其类型系统中，灵感来自线性逻辑 [35, 36] 和线性类型 [98, 66], ，并强制规定每块内存（一个值）在任何时刻都只有一个所有者 （变量绑定） [19, 17]。当所有者离开作用域时，内存会自动释放，从而支持用户定义的析构函数，并支持资源获取即初始化（RAII）模式 [89]。所有权可以通过引用转移（移动）或暂时共享（借用）。Rust 支持两种借用方式：多个不可变 借用，或单个可变 借用，但两者绝不会同时存在。这些约束由编译器通过借用检查强制执行，可确保内存安全，并通过确保引用生命周期不会超出其所有者的生命周期来防止悬垂指针。为了支持底层操作，Rust 提供了 **`unsafe`** 块；在这些块中，编译器的安全保证会暂停，避免未定义行为的责任则由程序员承担。在这些块之外，Rust 会严格执行安全规则，因此安全代码中不可能出现未定义行为。Rust 通过其属性系统支持软件可变性，尤其是 `cfg` 属性，它允许根据指定的配置谓词进行条件编译。Rust 中的一个特征由在某个 `cfg` 外部属性中标注了相同配置名称 的所有项 （参见脚注 1）构成。产品是否包含某个特征，取决于编译时对配置谓词 的求值，例如 *`#[cfg(any(unix, windows))]`*。 Cargo——Rust 的包管理器——允许开发人员在 `Cargo.toml` 文件中使用配置名称 声明特征，并通过逻辑运算符将其组合成配置谓词。特征依赖关系，即特征模型 [41, 83, 42], 的跨树约束 ，也可以在同一文件中指定。

2.2 中心性度量

自社会网络分析兴起以来，中心性度量 一直被用于社会学和心理学，以识别网络中的关键节点 [82, 8, 43]。此后，它们成为图论和网络分析的核心内容 [16, 24, 46]。网络 通常建模为一个图，其中包含 n 个节点，编号为 $i \in \{1, 2, \ldots\ldots, n\}$，并用邻接矩阵 $A \in \mathbb{R}^{n \times n}$ 表示；其中， $A_{ij} \neq 0$ 表示节点 i 与节点 j 之间存在一条边，否则为 $A_{ij} = 0$。正如 Bloch 等人 [13] 所指出的，并非所有中心性度量都会考虑边方向和权重，不过这些因素也可以纳入考量。形式上，中心性度量是一个函数ϕ : $\mathbb{R}^{n \times n} \to \mathbb{R}^n$，其中 $\phi_i(A)$ 用于量化节点 i 在网络 A 中的相关性。作为一种基数不变量，它可以根据节点的得分对节点进行序数排序（$\phi_i(A)$）。中心性度量大致分为两类：基于几何的和基于谱的 [14]。本节余下部分将介绍一些最常用的中心性度量。

几何中心性度量。 几何中心性度量通过距离函数来赋予相关性——也就是说，根据给定节点在各个距离上的节点数量。度中心性计算与某个节点相连的边数：入度为 $d^-(i)$，出度为 $d^+(i)$。接近中心性由 Bavelas [8], 提出，将节点的相关性定义为与所有其他节点的距离之和的倒数。由于原始公式在非连通图上会失效（因为距离为无穷大），一种广泛采用的变体为：

$$\mathrm{Clo}_i(A) = \frac{1}{\sum_{\substack{j=1 \\ d(i,j)<\infty}}^{n} d(i,j)}$$

其中，$d(i, j)$表示节点 i 与 j之间的最短路径。调和中心性 [57] 定义为某个节点与所有其他节点之间最短路径长度的倒数之和：

$$\mathrm{Har}_i(A) = \sum_{j=1}^{n} \frac{1}{d(i,j)}$$

与接近中心性不同，该指标在非连通图中也有明确定义；对于孤立节点，由于对所有 $j \neq i$，$d(i, j) = \infty$，因此总和为 0。中介中心性 [33] 衡量一个节点出现在节点对之间最短路径上的频率：

$$\mathrm{Bet}_i(A) = \sum_{s \neq i \neq t} \frac{\sigma_{st}(i)}{\sigma_{st}}$$

其中，σ_{st} 是从 s 到 t的最短路径数，而 $\sigma_{st}(i)$是经过节点 i的路径数。Newman [64] 利用 Dijkstra 算法计算最短路径，将接近中心性 扩展到加权网络。Opsahl 等人 [70] 进一步通过结合边权重与路径数量，推广了最短路径度量，使其既适用于接近中心性 ，也适用于中介性 中心性。

谱中心性度量。 图的谱 中心性度量利用图的邻接矩阵的主左特征向量来评估节点重要性。特征向量中心性 [15]为与其他高分节点相连的节点赋予更高分数。它定义为与邻接矩阵的最大特征值 λ相对应的特征向量 $\boldsymbol{v}$ A：

$$\lambda \boldsymbol{v} = A\boldsymbol{v}$$

如果 A 是一个随机矩阵，其主特征值为 1。然而，特征向量中心性在非连通图 [11] 上表现不佳。*Katz* 中心性 [43] 通过考虑连接某个节点与所有其他节点的不同长度路径数量来解决这一问题。其定义如下：

$$\boldsymbol{k} = \mathbf{1}(I - \beta A)^{-1}$$

其中，$\mathbf{1}$ 是全 1 向量，I 是单位矩阵，β 是满足$\beta < 1/\lambda$的阻尼因子，其中 λ 是 A的主特征值。

3 深入探究 RustyEx

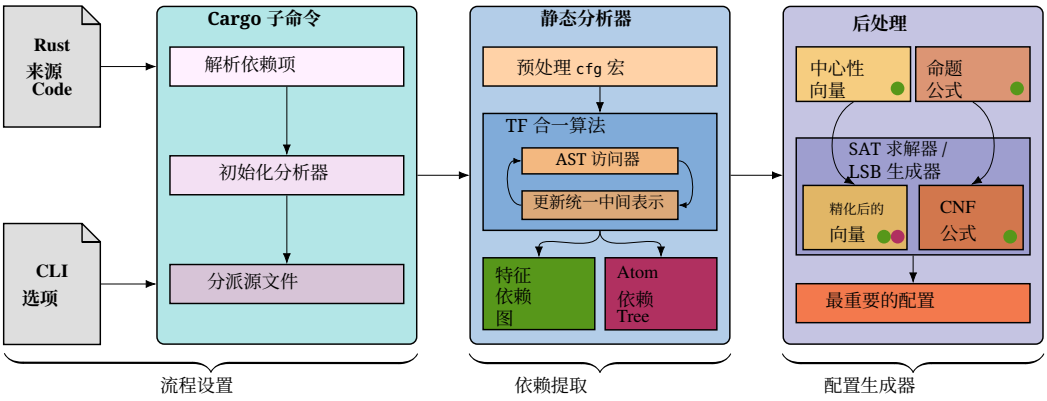
如第 1 节所述，RustyEx 是一种全自动工具，可对 Rust 编译器进行插桩，以便：1. 提取 Rust 软件中的特征依赖，2.根据特征对代码的影响为其分配权重，3. 应用中心性度量对特征进行排序，以及 4.优先处理配置。

3.1 流程概述

RustyEx 分三个主要阶段进行分析：1. 流程设置，2. 依赖[redacted].配置[redacted]

流程设置。 Rust 项目广泛使用 `cfg` 属性进行条件编译，既用于功能门控——通过 `Cargo` 启用或禁用特征——也用于平台特定代码选择等目的 [3, 18, 50]。为确保易用性，RustyEx 与 Rust 工具链无缝集成。在此阶段，它接受一个 Rust 项目以及命令行界面

图 1 RustyEx 的各个阶段（灵感来源： [51]）



选项，用于自定义分析，例如要生成的配置数量（N）和所选的中心性度量。流程设置使用 `Cargo` 解析内部 crate 依赖，这些依赖关系对分析至关重要。[6] 随后，它根据提供的选项在内存中初始化分析，并配置源文件的分派，使其使用经过插桩的 `rustc`。

依赖提取。RustyEx 对 Rust 编译器进行插桩，从基于 AST 得到的 UIR 中提取特征依赖图（见图 2 中的❷和❷）以及原子依赖树（见图 2 中的❸和❸）。第 3.2 节讨论了这些结构及其提取过程的细节。我们通过实现 `rustc_driver_impl::Callback` trait 将其集成到编译器中；该 trait 提供四个钩子：`config`、 `after_crate_root_parsing`、 `after_expansion`和 `after_analysis`。[7] `rustc` 会在 AST 创建后、名称解析前执行急切宏展开[8] 。虽然 `after_crate_root_parsing` 本可用于宏展开前的分析，但它无法访问尚未解析的子模块。因此，我们接入 `after_expansion`，它会在急切宏展开、名称解析和 AST 验证之后调用。为避免过早求值 `cfg` 属性，我们在 `config` 回调中注册自定义文件加载器。该加载器会重命名 `cfg` 宏，以延迟其求值。在 `after_expansion` 回调中，我们对 AST 进行过程间分析，以提取 UIR，并使用第 3.2 节所述的算法 1 中展示的加权不动点算法为其节点加权。这些权重用于估算每个特征对代码的影响。

配置生成。提取加权特征依赖图和原子依赖树后，RustyEx 利用它们识别最相关的配置。加权特征依赖图有两项关键用途：1. 通过基于几何的方法和/或基于谱的方法计算中心性度量以对特征进行排序（图 1 后处理阶段左上方的框）；以及2. 生成相应的命题公式（图 1 后处理阶段右上方的框）。这些任务将在第 3.3 节详细说明。随后将命题公式转换为合取范式。接下来，原子依赖树会细化特征排序（图 1 后处理阶段中间的框）。最后，SAT 求解器会选出满足该合取范式公式的前 N 个配置，并优先选择那些

[6]为提升性能，RustyEx 仅分析目标 crate 的源文件，并使用标准 Rust 编译器编译外部依赖关系。这降低了计算开销，但可能会略微影响准确性 [97]。[7]详情请参阅 `https://doc.rust-lang.org/nightly/nightly-rustc/` 。[8]急切展开会尽可能早地展开宏参数，而不考虑宏调用的类型。

精细排序最高的配置——即最相关的配置（图 1 后处理阶段底部的框）。

3.2 从抽象语法树到 UIR

概述。在抽象语法树中，`cfg` 属性节点及其关联的项作为彼此独立的子节点，出现在同一个祖先节点下。构建 UIR 包含两个关键步骤：统一和加权。UIR 这一名称源自统一步骤，该步骤将每个 `cfg` 属性及其关联的项合并为一个称为原子的单一节点。UIR 节点是代数数据类型 [48, 76, 93, 44, 10], ，具体而言是 Σ-类型——也就是说，节点可以是原子，也可以是相关的普通抽象语法树节点。只有当加权步骤为普通抽象语法树节点赋予非零相关性时，这些节点才会被纳入 UIR，例如，参数上的泛型界限通常会被排除。比如，在图 2 的❶和❶中，函数 `foo` 及其第 1 行上的 `#[cfg(feature= "a")]` 属性会被统一为一个原子节点，其中心是带注释项 `foo`。相反，函数 `bar` 的主体节点在 UIR 中仍是普通抽象语法树节点，因为它们会影响其权重，有助于量化代码中有多少部分受到相关 `cfg` 特征的影响。

形式化。我们将抽象语法树定义为一对 $(N_{ast},\ E_{ast})$，其中 N_{ast} 和 E_{ast} 分别是节点集和边集。令 $N_{rel} \subseteq N_{ast}$ 表示相关节点集。如果某个节点会影响依赖特性的元素的语义，则该节点被视为相关节点。例如，`let` 语句是相关的，而 `extern crate` 或 `use` 声明则不是。

原子集 定义如下：

$$A = \{(p, t) \mid ann(t) = p \land t \in T \land p \in P\}$$

其中：

1. P 是所有 `cfg` 谓词的集合，每个谓词都定义为递归的 Σ类型，具有以下结构：

$$p = \begin{cases} \texttt{single}(f) & \text{for } f \in F \\ \texttt{not}(f) & \text{for } f \in F \\ \texttt{any}(p_1, p_2) & \text{for } p_1, p_2 \in P \\ \texttt{all}(p_1, p_2) & \text{for } p_1, p_2 \in P \end{cases}$$

2. $T \subseteq N_{ast}$ 是所有项的集合，并且
3. $ann:\ T \to P$ 是一个满射函数，将项映射到 `cfg` 谓词。

UIR 是 AST 的增强型 诱导子图，其中节点附有 `cfg`谓词，且边方向反转。形式上，UIR 定义为 $\mathcal{U} = (N,\ E,\ w_N)$，其中：

- $N = A \cup N_{rel}$ 包含 $A \cap N_{rel} = \emptyset$、原子 和相关的 AST 节点，构成 Σ类型结构，$E = \{(i,\ j) \mid (i,\ j) \in E'\}$，其中 $E' \subseteq E_{ast}$ 是从 AST 派生的反向边集合，并且
- $w_N:\ N \to \mathbb{N}^+$ 根据项的类型为节点分配权重。

由于 UIR 节点可能是复合的 Σ 类型（例如原子），仅靠 $E' \subseteq E_{ast}$ 无法捕获所有必要的结构。为保留连通性，若 $(i,\ j) \in E_{ast}$ 且存在 $a \in A \subseteq N$使得 $i = term(a)$ 或 $j = term(a)$ 成立，我们将 E' 扩展为：

$$E' \leftarrow E' \cup \{\, ((ann(i),\ i),\ j) \lor (i,\ (ann(j),\ j)) \,\},$$

这里， $term: A \to T$ 从原子中提取项。这确保当一个节点被合并到原子中时，其原始 AST 边仍保留在 UIR 中，就如同该节点仍然单独相关一样。

合一算法。 合一算法对抽象语法树执行深度优先遍历，以识别 `cfg` 属性及其相关项，并将它们合并为原子。RustyEx 通过 `rustc_ast::visit::Visitor` trait 实现访问者模式 [34] 。为在遍历期间追踪父子关系，它维护一个项栈，将每个遍历到的项压入栈中。由于 `cfg` 属性总是作为项最左侧的子节点出现，遇到属性时便会触发统一过程。该过程包括：

1. 从 `cfg` 属性中提取配置谓词，2. 将谓词解析为自定义的内部表示，称为 `ComplexFeature`，以及3. 从栈中弹出相应的项，以便将其与谓词统一，形成一个原子节点。

例如， `cfg` 图 2 中第 3 行的属性❶ 被解析为

```
ComplexFeature::Any(vec![
    ComplexFeature::Simple(
        Feature {
            name: "b".to_string(),
            not: false
        }
    ),
    ComplexFeature::Simple(
        Feature {
            name: "c".to_string(),
            not: false
        }
    )
])
```

并与项统一 `bar` ，从而形成相应的原子节点。项遍历完成后，会将其从栈中弹出，并添加一条从子节点指向其父节点的 UIR 边。与抽象语法树不同，UIR 边的方向是反转的，以反映基于词法作用域的关系并捕获依赖流——这对于在特征依赖图中应用图中心性度量至关重要（第 3.3 节）。原子依赖树中的 `bar` 节点（图 2 中的 ❷ ）展示了其统一后的特征，这些特征以 and 标记突出显示，强调了统一和边方向。

赋权算法。UIR 中的每个节点都被赋予一个正整数，即 $\mathbb{N}^{+}$中的元素。在统一阶段，RustyEx 根据每个 UIR 节点所表示的项的类型，为其赋予一个权重类型。这些权重类型决定了算法 1（第 16–25 行）中权重的计算方式，包括：

1. 无权重：与分析无关、因此不赋予权重的节点（例如 **`extern crate`**声明），2. 内在权重：赋予固定权重 1 的节点（例如 **`let`** 语句），3. 子节点权重：权重为其子节点权重之和的节点（例如 `foo` 图 2 的❶中的节点），以及4. 引用权重：引用 UIR 中其他节点并继承其权重的节点（例如 `bar()` 调用，在 `qux` 图 2 的❶ 中）。

完成统一后，RustyEx 运行不动点算法（算法 1），计算 UIR 节点的最终权重函数 w_N 。在 Rust 中，多个项 只有在其 `cfg` 属性互斥时，才能共享同一个标识符。令 o_n 表示节点 $n \in N$ 出现的不同原子 数量。该算法初始化如下：

算法 1 计算 UIR 节点的权重

```
1: m_w ← ∅
2: Q ← ∅
3: r ← root(U)
4: U' ← U^T                                              ▷ 转置
5: 计算_权重(U', r, Q, m_w)
6: 解析_队列(U', Q, m_w)
7: U ← U'^T                                              ▷ 转置回去

8: 函数 计算_权重(U', n, Q, m_w)
9:     对于所有 adj ∈ ADJ(U', n) 执行                     ▷ 邻接节点
10:        计算_权重(U', adj, Q, m_w)
11:    结束循环
12:    ch_w ← 0                                          ▷ 子节点权重
13:    对所有 adj ∈ 邻接(U', n) 执行
14:        如果 adj.status = 等待 则                      ▷检查状态
15:            n.status ← 等待; 返回
16:        结束如果
17:        ch_w ← ch_w + adj.weight
18:    结束循环
19:    匹配 n.weight_kind 与
20:        情况 否:  n.weight ← 0
21:        分支 内在:  n.weight ← 1 + ch_w
22:        分支 子节点:  n.weight ← ch_weight
23:        分支 引用(called):
24:            如果 m_w[called] = ∅ 则                    ▷ 未找到定义
25:                n.status ← 等待
26:                Q ← Q ∪{n}
27:                返回
28:            else
29:                t.weight ← 平均值(m_w[called])
30:            结束 if
31:    结束 match
32:    n.status ← 加权
33:    m_w[n] ← m_w[n] ∪ n.weight
34: 结束函数

35: 函数resolve_queue(U', Q, m_w)
36:    S ← ∅                                             ▷已访问节点集合
37:    当 Q ≠ ∅ 时执行
38:        c ← Q ↓
39:        Q ← Q − {c}                                   ▷ 出队
40:        如果 S ∩ {c, LEN(Q)} ≠ ∅ 则                   ▷ 恢复机制
41:            c.weight ← DEF_W                          ▷分配一个默认权重
42:            继续
43:        结束条件判断
44:        S ← S ∪ {c, len(Q)}
45:        calc_权重(U', c, Q, m_w)
46:        if c.status = 等待 then
47:            Q ← Q ∪{c}
48:        结束 if
49:    结束 while
50: 结束函数
```

- m_w：$N \to \mathbb{N}^{+^{o_n}}$，一个用于存储与节点标识符关联的所有权重的映射，一个空队列 Q、
- 根节点 $r = root$（$\mathcal{U}$）以及转置后的 UIR $\mathcal{U}' = \mathcal{U}^T$，因为原始 UIR 是自底向上构建的，
- 对其进行转置可恢复父子关系。
-

加权 算法从根节点 r 开始，调用递归的 calc_权重函数（第 17 行），该函数通过遍历 $\mathcal{U}'$ 中的子节点，并应用与其权重类型关联的规则，计算每个节点 $n \in N$ 的权重。计算出的权重会同时存储在节点和映射 m_w 中，以供重用。对于每个类型为 引用的节点 $n \in N$，例如函数调用，算法会检查被引用的节点是否已在 m_w中计算出权重。如果权重尚不可用，则将该节点标记为等待 并加入队列 Q。随后，定点 resolve_queue 函数（第 30 行）会处理 Q，为队列中的节点计算权重，直到队列为空或检测到循环，例如由直接递归或相互递归 [67, 55]导致的循环。当 Q 在两次完整迭代之间保持不变时，即检测到循环。为防止无限循环，算法会为涉及循环的节点指定默认回退权重，并继续处理队列中的其余节点。[9] 最后，将 UIR 转置回原始方向（撤销之前的变换），并将最终权重存储在 w_N中。

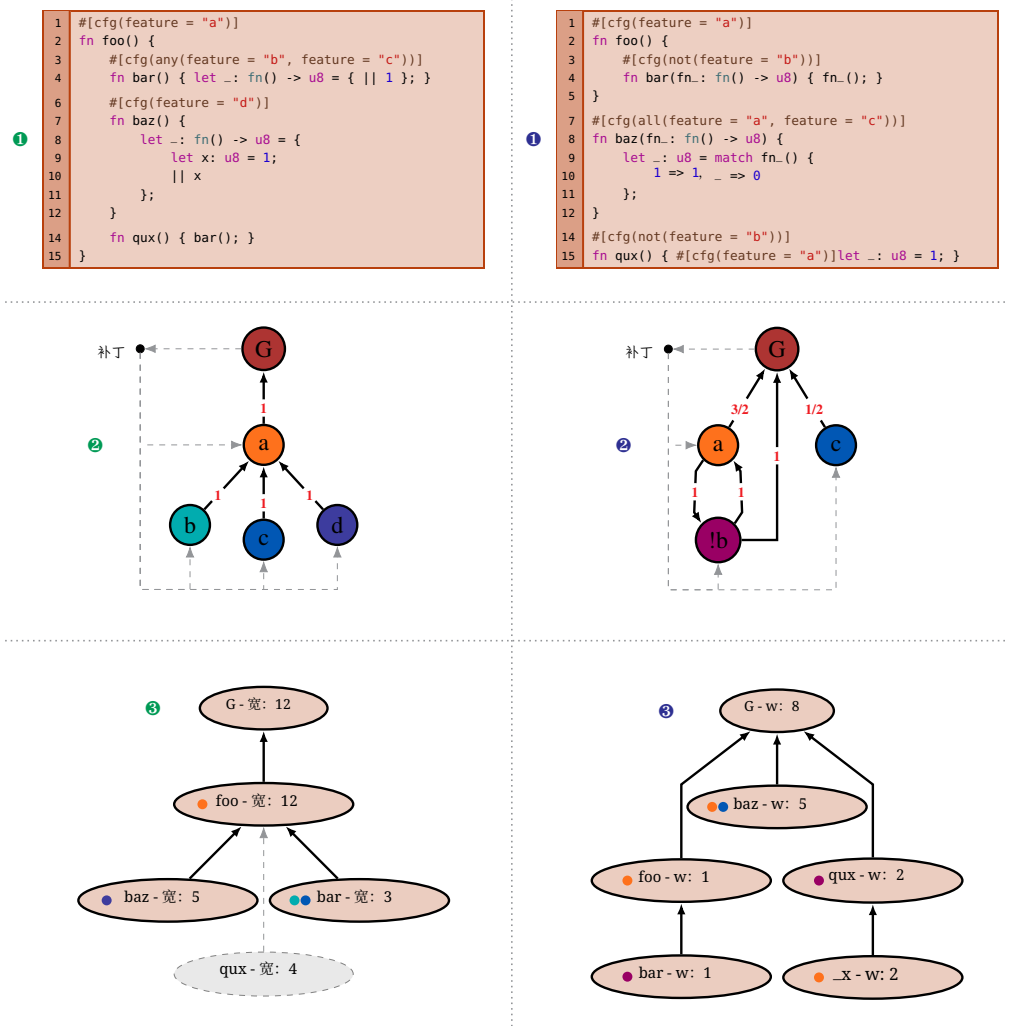
3.3 从 UIR 到特征依赖图

概述。如图 2（❷、❷）所示，特征依赖图是一种加权有向图，用于表示软件特征之间的依赖关系。该图通过对 UIR 应用特征提取算法构建。该算法根据特征所在配置属性的配置谓词，定义了分配边权重的规则。从高层次来看，该图反映了特征之间基于词法作用域的关系。具体而言，如果 f_i 在标注为 f_j 的原子词法作用域内使用，那么 f_i 依赖于 f_j，并会创建一条从 f_i 指向 f_j 的有向边。例如，在图 2 ❷ 的第 3 行，`feature = "b"` 和 `feature = "c"` 出现在一个 `any` 配置谓词中，该谓词位于与 `feature = "a"` （第 1 行）统一的原子的词法作用域内，并包围着 `foo` 函数（第 2 行）。根据特征提取算法，会创建两条权重为 1 的边：从 `feature = "b"` 和从 `feature = "c"` 指向 `feature = "a"`。相较之下，在 ❷ 的第 7 行，`feature = "c"` 出现在全局词法作用域（记为 G）内的一个 `all` 配置谓词中。因此，会添加一条从 `feature = "c"` 指向 G、权重为 $1/2$ 的边。与 UIR 一样，特征依赖图中的边方向与原始抽象语法树中的边方向相反。其创新之处在于保留了这种自然的边方向：这使我们能够根据依赖于某个特征的其他特征的状态（即配置谓词），来解读该特征的声誉（或相关性）。这样便可以在特征依赖图上应用图中心性度量，而边权重的设计旨在支持更准确的特征排序。

形式化。到目前为止，我们将特征依赖图称为图。然而，在构建过程中，它最初是一个多重图，因为某个特征 f_i 可能会在代码的多个部分依赖另一个特征 f_j。形式上，这个多重图表示为$\mathcal{F} = (F, D, w_R)$，其中：

[9]另一种方法是通过检测 LLVM-IR [97], ，尝试求解一个根据定义无解的线性方程组。

图 2 RustyEx 在两个场景中的演示过程



1. $F = \{(f_i, p_i) \mid f_i \in CN \wedge p_i \subseteq P\}$ 是节点集合，其中 CN 是所有特征名称的集合，p_i 是所有使用 f_i 的 `cfg` 前驱节点谓词的集合；2. $D = (U, m)$ 是表示特征依赖关系的有向边多重集，其中 $U = \{(i, j) \mid i, j \in F\}$ 是边集合，$m : U \to \mathbb{N}^+$ 是用于统计依赖关系出现次数的重数函数；以及 3. $w_R : R \to \mathbb{R}$，其中

$$R = \{((i, j), k) \mid (i, j) \in U,\ 1 \le k \le m((i, j))\},$$

是一个权重函数，用于为每个单独的边实例分配权重，并由重数索引 k 标识；权重由产生依赖关系时的配置谓词 决定。

随后，特征提取算法通过聚合权重，将这些多条边合并为单条边，从而将多重图 $\mathcal{F}$ 有效转换为简单图 $\mathcal{F}' = (F,\ D',\ w'_R)$，其中：

$$D' = \{(i, j) \mid (i, j) \in U\}$$

是从 D 中移除重复项后得到的唯一边集合，聚合权重函数定义如下：

$$w'_R((i,j)) = \sum_{k=1}^{m((i,j))} w_R((i,j),k),\ \forall (i,j) \in D'.$$

此处，总和汇总了边 (i,j) 所有实例的权重，其中 $m(i,j)$ 表示此类实例的总数。每个边实例 (i,j,k) 都会为最终图 $\mathcal{F}'$ 中的边 (i,j) 贡献相应的权重。

特征提取算法。特征提取算法根据 UIR 构建特征依赖图 F′。它首先遍历 UIR 原子 $A \subseteq N$。对于每个原子 $a \in A$，算法会获取其父节点 $\hat{a} \in \{x \mid (\hat{a}, x) \in E \wedge x \in A\}$。对于所有满足 $a \neq$ G 的 $a \in A$，$\hat{a}$ 的存在性和唯一性都有保证，因为 UIR 是抽象语法树的诱导子图，而树中的每个节点都有唯一的父节点。这是因为 UIR 是抽象语法树的增强诱导子图。具体而言，$\hat{a}$ 是 a 最近的原子祖先。该算法会从每个特征 $f \propto pred(a) = p_a$ 向每个特征 $\hat{}$ $f \propto pred(\hat{a}) = p_{\hat{a}}$ 创建一条有向边，其中 $\propto$ 表示“涉及”，而 $pred: A \to P$ 则返回标注每个原子的配置谓词。特征节点 f 和 f 的 $\hat{}$ 谓词集合更新如下：

$$(f,p) \leftarrow (f, p \wedge pred(a)) \text{ and } (\hat{f}, p') \leftarrow (\hat{f}, p' \wedge pred(\hat{a})).$$

多重边的权重根据 a 的嵌套 `cfg` 谓词得出。对于每个 $\hat{}$ 特征节点 f，以及每个 $(f,\ w) \in \chi(p_a,\ 1)$，算法会使用递归定义的函数 χ，从 f 到 $\hat{}f$ 创建一条加权边：

$$\chi(p_a, w) = \begin{cases} [(f,w)], & \text{if } p_a \equiv \texttt{single}(f) \vee \texttt{not}(f) \\ \chi(p_1,1) \oplus \chi(p_2,1), & \text{if } p_a \equiv \texttt{any}(p_1,p_2) \\ \chi(p_1, d^w_{p_1}) \oplus \chi(p_2, d^w_{p_2}), & \text{if } p_a \equiv \texttt{all}(p_1,p_2) \end{cases}$$

其中 $d^w_p = (w \times |\{f \propto p\}|)^{-1}$，$\oplus$ 中缀运算符用于连接两个列表。例如，在❶ 中，图 2 的 `feature = "b"` （第 3 行）出现在一个 `any` 谓词中，该谓词嵌套在`feature = "a"` （第 1 行）和 `foo` （第 2 行）之下，因此会从 `feature= "b"` 到 `feature = "a"`添加一条权重为 1 的边。如第 3.3 节所述，这种加权方案优先考虑 `any` 谓词中的特征，而非 `all` 谓词中的特征，这反映出 `any` 谓词限制较少（只需启用一个特征），而 `all` 谓词则要求同时启用其中涉及的所有特征。最后，通过将源节点和目标节点相同的所有边的权重相加，简化该图。

中心性度量。构建特征依赖图后，RustyEx 会根据通过命令行参数指定的中心性度量，按相关性对特征进行排序（见图 1，中心性向量框内的标记）。由于$\mathcal{F}$ 是一个非连通的加权有向图，因此某些中心性度量要么未定义，要么表现不可预测 [11]。例如，特征向量中心性要求图是连通的 [11]，而接近中心性和中介中心性则需要进行调整，才能处理加权有向图 [64, 70]。[10] 为确保连通性，我们引入一个合成的补丁节点（见❷ 和❷，图 2）。该节点有一条来自G 的入边，以及指向所有其他节点的出边，每条边的权重均为1。这一变换保证$\mathcal{F}$ 完全连通，因此无需修改即可应用以下中心性度量：

[10]RustyEx 使用 `rustworkx` [92] 来执行中心性度量。

- *Newman* 接近中心性 [64]
- *Opshal* 中介中心性 [70]特征向量
- 中心性 [15]*Katz* 中心性 [43]

计算中心性分数后，会生成一个向量 $\vec{v}$，其中每个条目 v_i 表示特征 f_i的中心性。补丁节点 不会纳入该向量，因为它们并非提取出的 特征。这种方法符合我们图构建背后的直觉：特征的声誉由依赖于它的特征决定。中心性度量为按照结构相关性对特征进行排序提供了一种有原则的方法。

命题公式与合取范式。 RustyEx 根据特征依赖图，通过遍历每个特征节点 $f_i \in F$来构建一个命题公式 。对于每个节点，它会生成一个子句 $\varphi_i = \ell(pred(f_i))$，其中 ℓ 将 `cfg` 谓词映射为逻辑 表达式。然后，它为每条边$(f_i, f_j) \in D'$添加一个蕴含式 $f_i \Rightarrow f_j$，以表示特征依赖关系。整体公式 φ 定义如下：

$$\varphi = \bigwedge_{f_i \in F} \left(\varphi_i \wedge \bigwedge_{f_j \in \text{succ}(f_i)} (f_i \Rightarrow f_j) \right)$$

其中，succ(f_i)是$\mathcal{F}'$中 f_i 的后继节点集合。最后，将 φ 转换为CNF公式φ_{cnf} [38]。可以在合取范式转换之前或之后移除重复子句。

3.4 从 UIR 到原子依赖树

概述。图 2（框 ❸ 和 ❸）展示了从 UIR 得出的原子依赖树。这个名称突出了该树对 UIR Σ 类型节点的选择性保留：仅保留原子变体，而移除普通抽象语法树节点。具体而言，只保留具有 `cfg` 属性的 UIR 节点。例如，在图 2 的框 ❸ 中，灰色 `qux` 节点（表示图 2 框 ❶ 中第 14 行的 `qux`函数）因不具备任何 `cfg` 属性而被排除。原子依赖树捕捉了 UIR 原子之间基于词法作用域的依赖。与将 UIR 结构扁平化的特征依赖图不同，这棵树保留了原子之间的父子关系，并由此保留特征之间的父子关系。每个节点的权重取决于该原子的 `cfg` 条件所影响的代码量，反映了特征的影响。为避免丢失被舍弃的普通抽象语法树节点所包含的信息，原子依赖提取算法会将它们的权重汇总到最近的祖先原子节点上（见图 2 框 ❸ 中的 `foo`节点）。最后，利用原子依赖树的结构对中心性向量 $\vec{v}$ 进行优化，从而更准确地排列特征相关性。

形式化。根据 UIR 的定义 $\mathcal{U} = (N, E, w_N)$，我们将原子依赖树定义为有向图 $\mathcal{A} = (A, E_A, w_A)$，其中：

- $A = N \setminus N_{rel}$ 是 UIR 中原子节点的集合，$E_A = E \setminus \{(i, j) \mid (i, j) \in E \wedge ($
- $i \in N_{rel} \vee j \in N_{rel})\}$ 是仅连接原子节点的边的集合，而w_A：$A \to \mathbb{N}^+$ 是权重函数，为每个原子分配一个权重，反映其特征所出现的 `cfg`属性会影响的代码量。

原子依赖树（$\mathcal{A}$）是$\mathcal{U}$的一个诱导子图，包含所有原子及其依赖关系，但不包含普通AST 节点。这显然成立，因为 $A \subseteq N$ 和 $E_A \subseteq E$。

原子依赖提取算法。 该原子依赖提取 算法根据 UIR 构建原子依赖树。它遍历 UIR 原子 $A \subseteq N$。对于每个原子 $a \in A$，它会识别其父节点$\hat{a} \in \{x \mid (\hat{a}, x) \in E \wedge x \in A\}$。和之前一样，

证明 $\hat{a}$ 的存在性很简单。然后，该算法区分两种情况：1. 如果$pred(a) \neq \emptyset$，则创建一条从 a 到$\hat{a}$ 的边，并将 a的权重加到$\hat{a}$的权重上；2. 如果$pred(a) = \emptyset$，则不会创建从 a 到$\hat{a}$ 的边，但仍会更新$\hat{a}$的权重。例如，在图 2 的方框 ❸ 中， `qux` 节点会计入 `foo` 的权重，但不会创建边。父节点权重更新的执行方式如下：

$$w_A(\hat{a}) \leftarrow w_A(\hat{a}) + w_N(a).$$

细化算法。细化算法使用原子依赖树 $\mathcal{A}$ 细化中心性向量 $\vec{v}$ （见图 1）。该算法遍历$\mathcal{A}$中的每个节点 $a \in A$，提取其中的每个特征 $f \propto pred(a)$，并更新该特征在 $\vec{v}$ 中的中心性值，如下所示：

$$\vec{v}_f \leftarrow \vec{v}_f + \begin{cases} \overline{w_A(a)}, & \text{if } f \propto \texttt{single}(f) \ \vee \ f \propto \texttt{not}(f) \\ \overline{w_A(a)}, & \text{if } f \propto \texttt{any}(p_1, p_2) \\ \dfrac{\overline{w_A(a)}}{|\{f \propto pred(a)\}|}, & \text{if } f \propto \texttt{all}(p_1, p_2) \end{cases}$$

这里，$f \propto p$（其中 $p \in P$）表示 f 出现在 a的 `cfg` 谓词的 p变体中，而$w_A(a) \in [0, 1]$是节点 a的归一化权重。归一化可防止将现有中心性值计入 $\vec{v}$。尽管$\mathcal{F}$ 捕获了特征之间的依赖关系，但可能无法充分表达它们的实际相关性。原子依赖树$\mathcal{A}$通过纳入每个特征所影响的代码范围 来细化中心性值，从而得到向量 $\vec{v}$的新排列$\vec{v}'$ 。

3.5 配置生成

配置生成 是 RustyEx 的最后一步。它根据中心性向量 $\vec{v}'$ 和 CNF 公式 φ_{cnf} 生成最相关的配置（见图 1）。虽然可以考虑更先进的基于词法作用域的策略，但 RustyEx 使用 SAT 求解器——包括增量式（例如 `MiniSat` [26, 87] 和 `CaDiCaL` [12]）和非增量式（例如 `Kissat` [12]）。给定要生成的配置数量 K，算法会遍历 $\vec{v}'$中的值。对于每个特征 f_i，它会按如下方式更新公式：

$$\varphi_{cnf} \leftarrow \varphi_{cnf} \wedge f_i$$

然后，它使用 SAT 求解器生成所有满足 φ_{cnf} 的配置。接着调用 SAT 求解器，为更新后的公式生成所有满足条件的配置。如果找到满足条件的配置，就将其添加到结果集中。此过程会重复进行，直到获得 K 个配置，并对每个新找到的配置取反，以便发现更多新的配置。

3.6 应用与说明

RustyEx 中配置的生成采用增量式和惰性方式。该工具会根据 $\vec{v}'$ 中的精细化中心性对所有特征进行排序，然后仅生成最有可能覆盖关键特征交互的前 K 个配置。实际上，这意味着优先选择包含 $\vec{v}'$中排名最高特征的配置，从 $\vec{v}'_0$开始，也就是中心性最高的特征。

如前所述，这种优先级排序策略通过聚焦于在特征相关性方面最具代表性且影响最大的变体，支持高效探索配置空间。通过优先分析这些配置，开发人员可以发现

最相关场景中的重要交互和行为，从而加深对整个系统的理解并改进验证。此外，这种有针对性的方法显著降低了对所有配置进行穷尽分析所需的计算成本和资源消耗，因此适用于大型、高度可配置系统。例如，这些优先级较高的配置还可作为有针对性的性能剖析或静态分析的候选对象，确保先检查关键特征组合，再检查相关性较低的组合。

这条配置感知型流水线凸显了中心性引导启发式在可配置系统面向特性的分析中的潜力，并为未来基于程序结构和语义属性优化配置采样的研究铺平了道路。它提出了一种更广泛的范式：利用程序结构和语义来指导对配置空间的系统性探索，从而有望惠及性能评估或安全分析等任务。

4 评估

我们从两个方面评估所提出的方法：1.通过证明生成配置均有效，来验证RustyEx ；以及 2. 评估RustyEx在一组40 个真实世界中排名靠前的开源 Rust 项目上的性能。

4.1 可靠性

前提。 为证明正确性，我们说明： 1. 合取范式编码忠实地表示了所有配置约束， 2. SAT 求解器仅返回满足赋值，并且 3. 绝不会生成无效配置。

CNF公式的构造。 子句构造—对于每个特征节点 $f_i \in F$，都会创建一个子句 $\varphi_i = \ell(pred(f_i))$，以强制实现对该特征配置谓词的逻辑解释。也就是说，如果某个特征在某项配置中启用，则其谓词必须得到满足。 蕴含关系构造—对于从 f_i指向 f_j 的每条出边（即，对于每个满足 $(f_i, f_j) \in D'$的 f_j ），都会添加一个蕴含式 $f_i \Rightarrow f_j$ ，以表示依赖关系。这确保了如果特征 f_i 处于激活状态，那么它所有依赖的特征 f_j 也都处于激活状态。 扩展—扩展公式 φ ，将 `Cargo.toml`中指定的必选特征纳入其中，从而确保它们出现在所有生成配置中。 等价性保持 — 由于合取范式转换采用了保持逻辑等价的标准技术，任何满足 CNF 公式 φ_{cnf} 的赋值也满足原始公式 φ。

SAT 解的正确性。若 SAT 求解器可靠且完备，[11] 它返回的任意赋值 α 都满足 φ_{cnf} 中的每个子句，因此也满足 φ中的每个子句。这意味着：

1. 对于每个带有子句 φ_i的特征 f_i，其在 α 中的激活状态满足其配置谓词：$\alpha \models \varphi_i \quad \forall f_i \in F$也就是说， α 满足该谓词；2. 对于每个依赖关系 $f_i \Rightarrow f_j$，若 $\alpha(f_i) =$ `true` $\Rightarrow \alpha(f_j) =$ `true`，则该蕴含在赋值 α 下成立；3. 来自 `Cargo.toml` 的所有必选特征在 α中均处于激活状态： $\forall f_i \in F_{mandatory}, \alpha(f_i) =$ `true`。

[11]若 SAT 求解器只返回满足公式的赋值，则称其可靠；若只要存在解，它就能找到该解，则称其完备。

23:16 基于编译器的精细特征排序，优先考虑配置相关性

ng the values of all crates within the

Atom e	D.T. Edges	Execution Time	Memory Usage
	22	1 s	53 MB
	510	817 s	999 MB
	537	320 s	945 MB
	120	108 s	902 MB
	67	1156 s	1910 MB
	319	153 s	488 MB
	32	117 s	513 MB
	55	91 s	497 MB
	7	26 s	339 MB
	116	320 s	887 MB
	38	404 s	2477 MB
	116	161 s	691 MB
	41	81 s	465 MB
	120	595 s	980 MB
	141	462 s	504 MB
	29	195 s	948 MB
	202	662 s	956 MB
	331	277 s	529 MB
	621	2016 s	3425 MB
	75	164 s	504 MB
	91	442 s	1827 MB
	10	1 s	64 MB
	208	246 s	435 MB
	581	1359 s	968 MB
	18	11 s	295 MB
	8	68 s	1395 MB
	13	12 s	360 MB
	1419	1135 s	1801 MB
	58	104 s	478 MB
	225	270 s	545 MB
	0	52 s	525 MB
	22	80 s	685 MB
	43	28 s	352 MB
	92	64 s	354 MB
	37	100 s	381 MB
	123	248 s	549 MB
	459	212 s	563 MB
	551	599 s	3800 MB
	146	88 s	489 MB
	14	64 s	506 MB
	7617	**13328 s**	**3800 MB**
	190	**333 s**	**885 MB**

根据定义，当且仅当满足以下条件时，配置才有效：

- 满足所有配置约束（包括特征谓词和依赖关系），并且包含所有必选特征。

因此，由于 SAT 求解器生成的任意赋值 α 都满足 φ_{cnf}中的所有子句，RustyEx 生成的每个配置都是有效的。

反证强化。 假设 SAT 求解器返回的配置 α' 无效，以此导出矛盾——也就是说，它违反了一条或多条约束。那么， φ_{cnf} 中至少存在一个子句未 被 α'满足。根据定义，满足赋值 满足公式中的所有子句，而 SAT 求解器的可靠性保证 α' 是 φ_{cnf}的满足赋值。由此矛盾可知，我们的假设不成立： α' 不可能无效。因此，SAT 求解器生成的所有配置都是有效的。

4.2 性能评估

实验设置。 所有实验均在一台配备英特尔酷睿 i5-1135G7@2.40 GHz 中央处理器和 16 GB 内存的普通笔记本电脑上进行。这一选择凸显了我们方法的轻量特性：与许多通常需要专用服务器或高性能硬件的程序分析和可变性管理工具不同，RustyEx 可在标准开发者工作站上运行。大多数分析的项目都组织为 `Cargo` 工作空间，自然地分解为多个 crate。对于每个 crate，我们运行 RustyEx，并设置 10 分钟的超时时间，这对于集成到持续集成（CI）流水线而言是一个切合实际的上限。我们收集了各种度量指标，并在项目层面进行汇总，包括特征和依赖关系的数量、有关 UIR 节点和边的统计数据，以及特征依赖图和原子依赖树的摘要。我们还监测了峰值内存使用量和执行时间。总体而言，这些选择表明，RustyEx 可以无缝融入日常开发工作流，而快速反馈和适度的资源需求正是此类工作流的关键。包含所有实验数据和脚本的复现包可在 Zenodo 上获取：

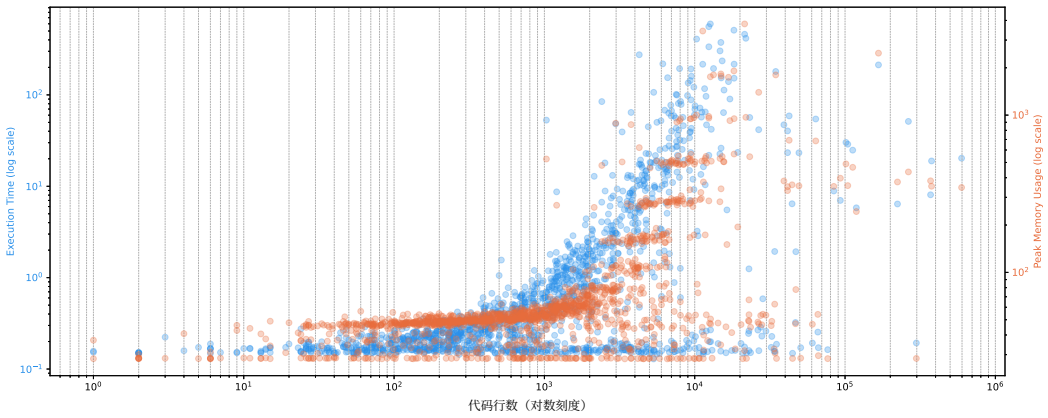
`https://doi.org/10.5281/zenodo.17691776`。

执行时间与可扩展性。 图 3（在 中突出显示）展示了代码行数与对数刻度下执行时间之间的相关性。RustyEx 成功完成了约 93% 项目的分析，在各种规模和领域中展现出良好的可扩展性。执行时间随代码规模平稳增长：对于中小型项目，分析通常可在几秒内完成；而对于最大的项目（例如 `hyperswitch` 和 `swc`），执行时间峰值达到 33 分钟。每个项目的平均执行时间为 333 秒，而单个 crate 平均约需 8 秒。这些结果证实，我们的工具不仅在实践中效率很高，而且足够稳健，能够处理包含数十万行代码和数千个特征的大型模块化代码库。值得注意的是，图中可见的对数趋势表明，相较于项目规模，分析成本呈次线性增长，这说明 RustyEx 适合长期扩展。

内存使用量。 图 3——其中突出显示了——展示了代码行数与峰值内存使用量之间的相关性，采用的同样是对数刻度。内存消耗总体较低：大多数项目所需的内存不到 1 GB，只有少数大型代码库（如 `hyperswitch` 和 `swc` ）的峰值达到 3.8 GB。每个项目的平均峰值内存使用量为 885 MB

23:18 基于编译器的精细特征排序：优先考虑配置相关性

图 3 代码行数（x 轴）与执行时间（秒）之间的相关性 以及峰值内存使用量（MB） .



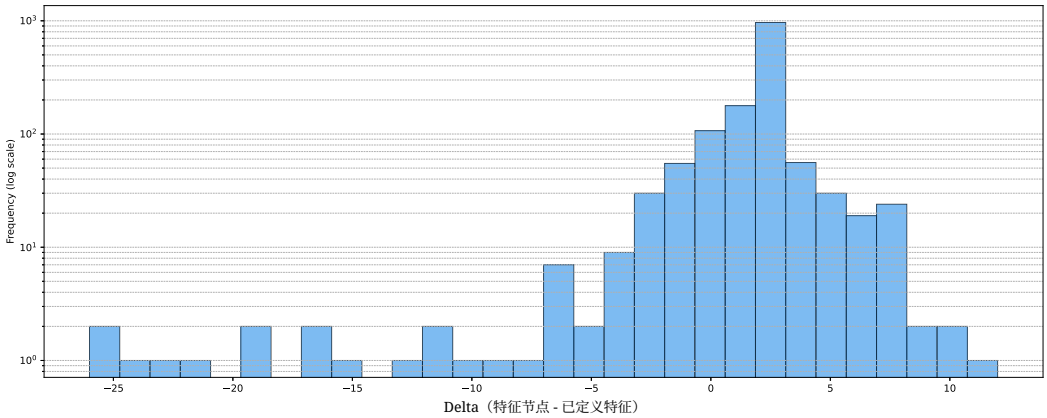
，这完全在主流笔记本电脑的承受范围内。在 crate 层面，平均峰值内存使用量降至 105 MB，这表明内存需求更多取决于各 crate 的内部复杂性，而非项目的整体规模。在各种不同项目中都表现出的这种稳定性，证实了 RustyEx 可以在资源有限的环境中运行，例如基于云的 CI/CD 流水线或开发者笔记本电脑。

中间结构。 RustyEx 生成的中间表示保持紧凑，且易于处理。平均而言，UIR 包含约 82,000 个节点和数量相近的边，与原始抽象语法树的规模相比相对较小。特征依赖图 则紧凑得多，在多边压缩后通常包含 100 至 500 条边。原子依赖树更小，平均比 UIR 小 95%。这种紧凑性尤为重要，因为它直接影响后续分析和排序过程的成本：更小的图和树能够更快地计算中心性度量并执行逻辑变换。这些结果表明，我们以原子和 UIR 为核心的抽象策略，在保留语义细节与降低分析开销之间取得了良好平衡。

特征复杂度。大多数项目都定义了相当多的特征，每个项目的中位数为425 个。平均而言，RustyEx 检测到122 个特征依赖图中的节点，通常揭示出跨特征依赖，这凸显了分析高度可配置系统所面临的挑战。如图 4 所示，检测到的特征数量通常多于 `Cargo.toml` 文件中明确声明的数量。这种差异主要源于两个因素：1. 同一工作空间中不同 crate 之间的特征重叠，这些特征不会在清单文件中重新定义；以及2. 对 `not`谓词的处理，该处理会引入逻辑上相互独立的变体，并且在最坏情况下会使检测到的特征数量翻倍。

尽管特征数量众多，但其使用情况仍不甚普遍：平均仅识别出 228 个特征相关制品，这意味着每个特征平均使用次数不到两次。这一发现表明，虽然项目在特征层面提供了丰富的配置空间，但许多特征对实际代码库的影响有限。因此，有必要采用有原则的优先级划分：如果大多数特征很少使用，开发人员和测试人员就应集中关注那些对系统影响最大的特征。

图 4 已声明特征的数量与检测到的特征数量之间的 Delta。



稳健性。 最后，我们评估了 RustyEx 的稳健性。在分析的 1,600 多个 crate 中（每个项目约 40 个），只有 116 个失败，成功率达到 93%。大多数失败是由于编译期间缺少系统依赖所致，与分析本身无关。超时情况很少见，受影响的 crate 不到 2%。重要的是，没有发现重大崩溃或错误结果。即使是 `diem` 和 `deno` 这样的大型模块化工作空间，也无需人工干预即可成功完成分析。这些结果证实，RustyEx 已足够成熟可靠，可以集成到研究人员和实践者的工具链中。

5 对有效性的威胁

我们按照 Wohlin 等人 [99]的分类法组织讨论。

5.1 构念效度

特征相关性的代理指标。 我们依赖在特征依赖图上计算的几何 和谱 中心性来近似衡量特征相关性。这些度量指标并非中立：它们各自存在偏差和局限，尤其是在不连通分量以及局部影响力与全局影响力方面。

缓解措施。 RustyEx 允许用户选择最适合其项目领域的中心性度量，从而降低系统性误读的风险。为避免排除孤立特征，该工具引入一个补丁节点 连接原本不连通的分量，确保所有特征都纳入排序过程。

UIR 节点的加权。 在定义缺失或检测到循环的情况下，类型为`引用` 的节点会被赋予默认权重。这种近似可能低估未解析的外部调用所起的作用，从而可能扭曲优先级划分。

缓解措施。 RustyEx 实现了一种恢复机制，为节点分配回退权重，以防止无限循环并限制失真。这确保未解析的依赖关系不会抵消其他相关节点的影响。

5.2 内部效度

关于 SAT 求解器正确性的假设。 我们的配置生成依赖于底层 SAT 求解器的可靠性和完备性。如果求解器产生了错误的

结果，生成配置的有效性就会受到威胁。

缓解措施。 为最大限度地降低这一威胁，我们采用广泛使用且经过充分测试的 SAT 求解器，从而降低出错的可能性，并增强方法的稳健性。**超时与分析失败**。 有一部分 crate（∼ 7%）因超时或缺少系统依赖而失败。这类失败可能会引入选择偏差，因为并非所有 crate 在分析中的代表性都相同。

缓解措施。 我们为每个 crate 设定固定的 10 分钟超时时间，以确保数据集内的处理公平。失败的 crate 将被跳过；但由于大多数失败是由缺少依赖关系而非工具本身的内在局限造成的，因此评估的整体有效性仍得以保留。

5.3 外部有效性

聚焦于开源 Rust 项目。 我们的评估基于来自 GitHub 和 `crates.io` 的 40 个开源项目。尽管这些项目涵盖了广泛的真实软件，但它们未必能体现专有或工业代码库的全部可变性。

缓解措施。 许多受分析的项目已在生产环境中得到广泛使用，并作为工业系统的依赖项。这增强了我们对结果能够推广至纯学术或业余软件之外的信心。此外，由于我们的方法与语言无关，未来的工作将探索其在其他生态系统中的应用，例如 C/C++。

固定配置预算。 我们为每个项目生成固定数量的配置，即 K 。在实际应用中，真实场景可能需要采用自适应策略，根据项目规模、发布阶段或可用资源调整生成配置的数量。

缓解措施。 RustyEx 支持参数化配置策略，用户可以根据自身需求和约束进行调整 K 。这种灵活性确保该工具能够适应动态的开发和测试场景。

5.4 结论有效性

对特征排序的依赖。 配置的生成顺序取决于所选的中心性指标以及对原子依赖树所做的细化。因此，不同的选择可能会产生不同的优先配置集合。

缓解措施。 RustyEx 支持切换中心性度量以及禁用细化。这使用户能够开展敏感性分析，并评估在不同假设下排序如何变化。

配置缺乏真实标准。 目前尚无普遍认可的“正确”或“重要”配置参考集合。因此，直接评估我们的方法所生成配置的相关性颇具挑战。

缓解措施。 为了解决这一问题，RustyEx 会执行结构一致性检查，并确保符合 `Cargo.toml` 约束，例如必需特性和有效的谓词逻辑。此外，它还会验证所有生成配置是否满足推导出的 CNF公式。这些保障措施确保生成配置既有效，又能代表关键执行路径。

6 相关工作

高度可配置系统中的配置优先级排序已得到广泛研究，但研究视角各不相同，所基于的假设也往往不同。多篇全面的

综述概述了这一领域，例如 [1] （特别参见第 4.4 节）、 [29], 和 [39]„ 它们系统地梳理了研究趋势并指出了开放性挑战。在此基础上，我们从三个主要维度简要总结最相关的研究成果。

静态分析与预处理。 早期关于高度可配置系统的研究，尤其是在 Linux 内核的背景下，主要聚焦于静态分析和预处理技术。El Sharkawy 等人 [28] 开发了处理 `#ifdef` 指令以提取软件度量的方法，从而更易于管理配置特定复杂度。类似地，Sincero等人 [86] 研究了 C 宏的预处理，以检测死代码，进而帮助减少变异性引起的维护工作量。这些方法与我们从源代码分析可变性的思路密切相关，但仍局限于特定生态系统中的度量提取或缺陷检测。相比之下，RustyEx 将变异性感知静态分析的概念推广到宏或指令之外，旨在识别 Rust 项目中的相关配置，而不假定存在特定语言的预处理产物。

基于特征模型的优先级排序。 另一项重要的研究方向以显式特征模型为基础，在这类模型中，特征及其关系通过结构化图示呈现。Bagheri 等人 [6] 提出了分层分析层次过程，通过将决策过程分解为易于管理的层次，对特征进行优先级排序和选择。Peng 等人 [74]采用有向加权无环图，通过加权度中心性评估特征重要性，并揭示特征影响力的不对称性。Mannion等人 [56]探索了基于变异性类型的加权策略，并假定图中存在显式的源节点与汇节点。除了优先级划分之外，Bagheri等人 [7] 还使用圈复杂度 [61], 网络中心性 [65],以及经典软件工程度量 [30]等结构度量评估特征模型的可维护性。尽管这些方法严谨且有效，但它们预设了形式化特征模型的存在——这一假设在实践中要求很高，因为许多现代系统（包括 Rust 项目）依赖于分散且隐式的可变性形式，这些形式直接编码在构建清单和条件编译中。RustyEx 在这一点上有所不同：它不要求预定义特征模型，而是直接从源代码与构建元数据中重建可变性信息。

用于优先级排序的中心性度量。 另一类互补研究将图论概念，尤其是中心性度量，应用于可变性管理和测试。Mohammed 等人 [63] 使用多种中心性指标对单个特征模型中的配置进行排序，从而识别出影响力远高于其他配置的配置。Levasseur 等人 [49] 使用中心性、面向对象度量和机器学习对单元测试进行优先级排序，表明图的结构属性能够指导测试中的资源分配。Ferreira 等人 [32] 更进一步，提出了融入中心性信息的变异调用图 [31, 40], 以确定在某些特征组合下可能出现漏洞的函数。尽管这些研究都证明了中心性在突出关键元素方面的价值，但它们仍局限于显式特征模型、测试策略，或漏洞检测等专门任务。我们的方法沿用了相同的思路——中心性能够体现相关性——但将其应用于一个新的抽象层次：从 Rust 的变异性构造中派生出的原子依赖树。由此，RustyEx 将基于中心性的优先级排序拓展到这样的生态系统：其中的特征模型是隐式的，配置空间既庞大，又很少得到充分探索。

7 结论

本文提出了首个通用方法，通过基于编译器的特征精化排序，对高度可配置软件系统中的配置进行优先级排序。与以往方法不同，我们的方法不依赖预先存在的特征模型，并且会明确考虑特征依赖以及每个特征影响的代码范围。

该方法结合过程间静态分析来提取统一中间表示 (UIR)、构建特征依赖图 和原子依赖树、基于中心性的排序，以及基于合取范式的 SAT 求解。通过这种组合，可以识别并生成最相关的配置，同时减少需要考虑的配置总数。

为了展示该方法的实用性，我们使用 RustyEx 实现了它。RustyEx 是一款面向 Rust 软件的全自动工具。对知名开源 Rust 项目进行的大规模评估表明，RustyEx 具有良好的可扩展性、稳健性和效率：它能够处理包含数千个特征的大型代码库，内存和运行时间开销适中，并且在多个 crate 中都取得了较高的成功率。该方法与编程语言无关，可应用于其他具有原生可变性支持的生态系统，例如 C/C++ 和 Java。

通过明确优先考虑最相关的特征和配置，我们的方法及其 RustyEx 实现支持高效测试、编译器优化、程序理解、可变性管理和回归分析。生成的配置有效、可靠，并能代表关键执行路径，为随机策略或均匀策略提供了一种有原则的替代方案。

总体而言，这项工作首次提出了一种通用、形式上健全且实用的配置优先级排序方法，可提升高度可配置系统的效率和故障检测能力，并为未来将其扩展到其他语言和领域奠定基础。

参考文献 1 Halimeh Agh、Aidin Azamnour 和 Sefan Wagner。《软件产品线测试：系统性文献综述》。《实证软件工程》，29(146)，2024年9月。2 Mustafa Al-Hajjaji、Thomas Thüm、Malte Lochau、Jens Meinicke 和 Gunter Saake。《利用基于相似度的产品优先级排序进行有效的产品线测试》。《Software and Systems Modeling》，18：499–521，2019年2月。3 Brian Anderson、Lars Bergstrom、Manish Goregaokar、Josh Matthews、Keegan McAllister、Jack Moffitt 和 Simon Sapin。《使用 Rust 构建 Servo 网络浏览器引擎》。载于 Tao Xie 和 Dongmei Zhang（编），《第38届国际软件工程会议论文集：软件工程实践》（ICSE-SEIP'16），第81–89页，美国得克萨斯州奥斯汀，2016年5月。IEEE。4 Sven Apel、Don Batory、Christian Kästner 和 Gunter Saake。《面向特征的软件产品线》。施普林格，2013年4月。5 Sven Apel、Hendrik Speidel、Philipp Wendler、Alexander von Rhein 和 Dirk Beyer。《利用特征感知验证检测特征交互》。载于 Corina Păsăreanu 和 John Hosking（编），《第26届国际自动化软件工程会议论文集》（ASE'11），第372–375页，美国堪萨斯州劳伦斯，2011年11月。IEEE。6 Ebrahim Bagheri、Mohsen Asadi、Dragan Gasevic 和 Samaneh Soltani。《分层分析层次过程：软件特征的优先级排序与选择》。载于 Jan Bosch 和 Jaejoon Lee（编），《第14届国际软件产品线会议论文集》（SPLC'10），《计算机科学讲义》6287，第300–315页，韩国济州岛，2010年9月。施普林格。7 Ebrahim Bagheri 和 Dragan Gasevic。《使用结构度量评估软件产品线特征模型的可维护性》。《Software Quality Journal》，19(3)：576–612，2011年1月。8 Alex Bavelas。《任务导向群体中的通信模式》。《美国声学学会期刊》，22(6)：725–730，1950年11月。9 David Benavides、Sergio Segura 和 Antonio Ruiz-Cortés。《20年后的特征模型自动化分析：文献综述》。《信息系统》，35(6)：615–636，2010年9月。10 Jan A. Bergstra 和 John V. Tucker。《等式规范、完备项重写系统以及可计算与半可计算代数》。《ACM期刊》，42(6)：1194–1230，1995年11月。11 Abraham Berman 和 Robert J. Plemmons。《数学科学中的非负矩阵》。Academic Press，1979年1月。12 Armin Biere、Katalin Fazekas、Mathias Fleury 和 Maximilian Heisinger。《CaDiCaL、Kissat、Paracooba、Plingeling 和 Treengeling 参加2020年SAT竞赛》。载于 Tomáš Balyoi、Nils Froleyks、Marijn J. H. Heule、Markus Iser、Matti Järvisalo 和 Martin Suda（编），《2020年SAT竞赛论文集》，第50–53页，意大利阿尔盖罗，2020年7月。赫尔辛基大学。13 Francis Bloch、Matthew O. Jackson 和 Pietro Tebaldi。《网络中的中心性度量》。《社会选择与福利》，61(2)：413–453，2023年4月。14 Paolo Boldi 和 Sebastiano Vigna。《中心性公理》。《互联网数学》，10(3–4)：222–262，2014年9月。15 Phillip Bonacich。《地位评分与派系识别的因子分解与加权方法》。《数学社会学期刊》，2(1)：113–120，1972年。16 Stephen P. Borgatti。《中心性与网络流》。《Social Networks》，25(1)：55–71，2005年1月。17 Chandrasekhar Boyapati、Robert Lee 和 Martin Rinard。《用于安全编程的所有权类型：防止数据竞争和死锁》。载于 Satoshi Matsuoka（编），《第17届ACM面向对象编程、系统、语言与应用年会论文集》（OOPSLA'02），第211–230页，美国华盛顿州西雅图，2002年11月。ACM出版社。

23:24 通过基于编译器的精细特征排序，优先考虑配置相关性

18 Shao-Fu Chen 和 Yu-Sung Wu。《使用 Rust 开发 Linux 内核模块》。载于 Yousra Aafer、Shujun Li 和 Yulei Wu（编），《可信与安全计算会议论文集》（DSC'22），第1–2页，英国爱丁堡，2022年6月。IEEE。19 David G Clarke、John M Potter 和 James Noble。灵活别名保护的所有权类型。载于 Craig Chambers（编），《第13届面向对象编程系统、语言与应用国际会议论文集》（OOPSLA'98），第48–64页，加拿大不列颠哥伦比亚省温哥华，1998年10月。ACM。20 Andreas Classen、Maxime Cordy、Pierre-Yves Schobbens、Patrick Heymans、Axel Legay 和 Jean-François Raskin。特征转换系统：验证可变性密集型系统的基础及其在 LTL 模型检测中的应用。《IEEE 软件工程汇刊》，39(8):1069–1089，2013年8月。21 Paul Clements 和 Linda Northrop。《软件产品线：实践与模式》。Addison-Wesley，2001年8月。22 Myra B. Cohen、Matthew B. Dwyer 和 Jiangfan Shi。存在约束时对高度可配置系统进行交互测试。载于 David S. Rosenblum 和 Sebastian G. Elbaum（编），《第16届软件测试与分析国际研讨会论文集》（ISSTA'07），第129–139页，英国伦敦，2007年7月。ACM。23 Keith D. Cooper 和 Linda Torczon。《编译器工程》。Morgan Kaufmann，2022年11月。24 Kousik Das、Sovan Samanta 和 Madhumangal Pal。社会网络中心性度量研究综述。《社交网络分析与挖掘》，8:13:1–13:11，2018年2月。25 Sanjoy Dasgupta。学习多树。载于 Kathryn B. Laskey 和 Henri Prade（编），《第15届人工智能不确定性会议论文集》（UAI'99），第134–141页，瑞典斯德哥尔摩，1999年7月。Morgan Kaufmann 出版社。26 Niklas Eén 和 Niklas Sörensson。可扩展的 SAT 求解器。载于 Enrico Giunchiglia 和 Armando Tacchella（编），《第6届可满足性测试理论与应用国际会议论文集》（SAT'03），LNCS 2919，第502–518页，意大利圣玛格丽塔利古雷，2003年5月。施普林格。27 Sascha El-Sharkawy、Adam Krafczyk 和 Klaus Schmid。Linux 中配置不匹配的实证研究。载于 Myra Cohen 和 Mathieu Acher（编），《第21届系统与软件产品线国际会议论文集》（SPLC'17），第19–28页，西班牙塞维利亚，2017年9月。ACM。28 Sascha El-Sharkawy、Adam Krafczyk 和 Klaus Schmid。软件产品线的快速静态分析：一个包含超过42,000个度量指标的示例。载于 Maxime Cordy 和 Mathieu Acher（编），《第14届软件密集型系统可变性建模国际工作会议论文集》（VaMoS'20），第1–9页，德国马格德堡，2020年2月。ACM。29 Sascha El-Sharkawy、Nozomi Yamagishi-Eichler 和 Klaus Schmid。软件产品线中可变性及其实现的分析度量指标：系统性文献综述。《信息与软件技术》，106:1–30，2019年2月。30 Norman E. Fenton。《软件度量：严谨方法》。伦敦：Chapman & Hall，1991年。31 Gabriel Ferreira、Christian Kästner、Jürgen Pfeffer 和 Sven Apel。利用变异性调用图刻画高度可配置系统的复杂性：分析配置选项交互对函数调用复杂性的影响。载于《安全科学研讨会与训练营论文集》（HotSoS'15），第1–2页，美国伊利诺伊州厄巴纳-香槟，2015年4月。ACM。32 Gabriel Ferreira、Momin Malik、Christian Kästner、Jürgen Pfeffer 和 Sven Apel。#ifdef 条件编译指令会影响漏洞出现吗？Linux 内核实证研究。载于 Rick Rabiser 和 Bing Xie（编），《第20届系统与软件产品线国际会议论文集》（SPLC'16），第65–73页，中国北京，2016年9月。ACM。33 Linton C. Freeman。基于中介性的中心性度量集。《社会计量学》，40(1):35–41，1977年3月。

34 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides。《设计模式：可复用面向对象软件的要素》。专业计算丛书。Addison-Wesley，美国马萨诸塞州雷丁，1995年。35 Jean-Yves Girard。线性逻辑。《理论计算机科学》，50(1):1–101，1987年。36 Jean-Yves Girard、Yves Lafont 和 Laurent Regnier。《线性逻辑进展》。剑桥大学出版社，1995年7月。37 Axel Halin、Alexandre Nuttinck、Mathieu Acher、Xavier Devroey、Gilles Perrouin 和 Benoit Baudry。全部测试，值得吗？评估 JHipster Web开发技术栈上的配置采样。《实证软件工程》，24(2):674–717，2019年7月。38 Colin Howson。《树逻辑：符号逻辑导论》。Routledge，1997年2月。39 Muhammad Idham、Shahliza Abd Halim、Dayang Norhayati Abang Jawawi、Zalmiyah Zakaria、Aldo Erianda 和 Nachnoer Arss。软件产品线测试用例优先级排序：系统性映射研究。《信息学可视化期刊》，7(3-2):2126–2134，2023年11月。40 Neil F. Johnson。《简明复杂性：复杂性理论清晰指南》。Oneworld出版社，2009年10月。41 Kyo C. Kang、Sholom G. Cohen、James A. Hess、William E. Novak 和 A. Spencer Peterson。特征导向领域分析（FODA）可行性研究。技术报告 CMU/SEI-90-TR-21，卡内基梅隆大学，美国宾夕法尼亚州匹兹堡，1990年11月。42 Ahmet Serkan Karataş、Halit Oğuztüzün 和 Ali Doğru。特征模型上的全局约束。载于 David Cohen（编），《第16届约束编程原理与实践国际会议论文集》（CP'10），LNCS 6308，第537–551页，苏格兰圣安德鲁斯，2010年9月。施普林格。43 Leo Katz。由社会计量分析导出的新地位指数。《Psychometrika》，18(1):39–43，1953年3月。44 Andrew Kennedy 和 Claudio V. Russo。广义代数数据类型与面向对象编程。载于 Richard P. Gabriel（编），《第 19 届ACM面向对象编程系统、语言与应用国际会议论文集》（OOPSLA'05），第21–40页，美国加利福尼亚州圣迭戈，2005年10月。ACM。45 Charles W. Krueger。软件产品线实践的新方法。《ACM通讯》，49(12):37–40，2006年12月。46 Andrea Landherr、Bettina Friedl 和 Julia Heidemann。社会网络中心性度量的批判性综述。《Business an Informtation Systems Engineering》，2(5):371–385，2010年10月。47 Jihyun Lee 和 Sunmyung Hwang。利用设计时决策进行可变性组合测试设计。《软件工程与知识工程期刊》，29(8):1141–1158，2019年8月。48 Daniel J. Lehmann 和 Michael B. Smyth。数据类型的代数规范：一种综合方法。《数学系统理论期刊》，14(2):97–139，1981年12月。49 Marc-Antoine Levasseur 和 Mourad Badri。利用面向对象度量、中心性度量和机器学习算法对单元测试进行优先级排序。《系统与软件工程创新》，第1–27页，2024年2月。50 Hongyu Li、Liwei Guo、Yexuan Yang、Shangguag Wang 和 Mengwei Xu。Rust-for-Linux 实证研究：成功、未如所愿与妥协。载于 Saurabh Bagchi 和 Yiying Zhang（编），《USENIX 年度技术会议论文集》（USENIX'24），第425–443页，美国加利福尼亚州圣克拉拉，2024年7月。Curran Associates, Inc.。51 Zhuohua Li、Jincheng Wang、Mingshen Sun 和 John C. S. Lui。MirChecker：通过静态分析检测 Rust程序中的缺陷。载于 Giovanni Vigna 和 Elaine Shi（编），《2021年计算机与通信安全会议论文集》（CCS'21），第2183–2196页，韩国线上会议，2021年11月。ACM。

52 Jörg Liebig、Sven Apel、Andreas Janker、Florian Garbe 和 Sebastian Oster。《重构引擎中的静态可配置性处理》。《Computer》，50(7):44–53，2017年。53 Jörg Liebig、Andreas Janker、Florian Garbe、Sven Apel 和 Christian Lengauer。《Morpheus：真实环境中的变异性感知重构》。载于 Gerardo Canfora 和 Sebastian Elbaum（编），《第37届国际软件工程会议（ICSE'15）论文集》，第380–391页，意大利佛罗伦萨，2015年5月。IEEE。54 Malte Lochau、Sebastian Oster、Ursula Goltz 和 Andy Schürr。《面向软件产品线工程中特征交互覆盖的基于模型的成对测试》。《Software Quality Journal》，20(3-4):567–604，2012年9月。55 Eva Magnusson 和 Görel Hedin。《循环引用属性文法——求值与应用》。《Science of Computer Programming》，68(1):21–37，2007年8月。56 Mike Mannion 和 Hermann Kaindl。《使用二进制字符串比较软件密集型系统产品线中的产品》。载于 Ina Schaefer 和 Maurice H. ter Beek（编），《第25届国际软件产品线会议（SPLC'21）论文集》，第257–266页，英国莱斯特，2021年9月。ACM。57 Massimo Marchiori 和 Vito Latora。《小世界中的和谐》。《Physica A: Statistical Mechanics and its Applications》，285(3-4):539–546，2000年10月。58 Hugo Martin、Mathieu Acher、Juliana Alves Pereira 和 Jean-Marc Jézéquel。《可配置系统性能特化学习的比较》。载于 Ina Schaefer 和 Maurice H. ter Beek（编），《第25届国际软件产品线会议（SPLC'21）论文集》，第46–57页，英国莱斯特，2021年9月。ACM。59 Nicholas D. Matsakis 和 Felix S. Klock。《Rust语言》。《ACM SIGAda Letters》，34(3):103–104，2014年10月。60 Antoni Mazurkiewicz。《迹理论导论》。载于 Volker Diekert 和 Grzegorz Rozenberg（编），《The Book of Traces》，第1章，第3–41页。世界科学出版社，1995年3月。61 Thomas J. McCabe。《复杂度度量》。《IEEE 软件工程汇刊》，2(4):308–320，1976年12月。62 Jean Melo、Elvis Flesborg、Claus Brabrand 和 Andrzej Wąsowski。《对 Linux 中变异性警告的定量分析》。载于 Ina Schaefer 和 Vander Alves （编），《第 10 届软件密集型系统变异性建模国际研讨会（VaMoS'10）论文集》，第3–8页，巴西萨尔瓦多，2016年1月。ACM。63 Fathiya Mohammed、Mike Mannion、Hermann Kaindl 和 James Patterson。《使用中心性指标评估产品线特征的相对重要性》。载于 Massimo Mecella 和 Arend Rensink（编），《第19届国际软件技术会议（ICSOFT 2024）论文集》，第469–476页，法国第戎，2024年7月。SciTe Press。64 Mark E. J. Newman。《科学合作网络：II. 最短路径、加权网络与中心性》。《Physical Review E》，64(1):016132，2001年6月。65 Mark E. J. Newman。《网络：导论》。牛津大学出版社，第一版，2010年3月。66 Martin Odersky。《线性类型的观察者》。载于 Bernd Krieg-Brückner（编），《第4届欧洲程序设计研讨会（ESOP'92）论文集》，LNCS 582，第390–407页，法国雷恩，1992年2月。施普林格。67 David von Oheimb。《相互递归与局部变量的霍尔逻辑》。载于 C. Pandu Rangan、Venkatesh Raman 和 Ramaswamy Ramanujam（编），《第19届软件技术与理论计算机科学基础会议（FSTTCS'99）论文集》，LNCS 1738，第168–180页，印度金奈，1999年12月。施普林格。68 Kurt M. Olender 和 Leon J. Osterweil。《Cecil：用于自动静态分析生成的序列约束语言》。《IEEE 软件工程汇刊》，16(3):268–280，1990年3月。69 Kurt M. Olender 和 Leon J. Osterweil。《序列约束的过程间静态分析》。《软件工程与方法学汇刊》，1(1):21–52，1992年1月。

70 Tore Opsahl、Filip Agneessens 和 John Skvoretz。《加权网络中的节点中心性：度与最短路径的推广》。《Social Networks》，32(3):245–251，2010年7月。71 Sebastian Oster、Florian Markert 和 Philipp Ritter。《软件产品线的自动增量成对测试》。载于 Jan Bosch 和 Jaejoon Lee（编），《第14届国际软件产品线会议（SPLC'10）论文集》，计算机科学讲义6287，第196–210页，韩国济州岛，2010年9月。施普林格。72 José A. Parejo、Ana B. Sánchez、Sergio Segura、Antonio Ruiz-Cortés、Roberto E. Lopez-Herrejon 和 Alexander Egyed。《高度可配置系统中的多目标测试用例优先级排序：案例研究》。《Journal of Systems and Software》，122:287–310，2016年12月。73 Sachin Patel、Priya Gupta 和 Vipul Shah。《使用多视角特征模型进行组合交互测试》。载于 Yang Liu 和 Pang（编），《第6届国际软件测试、验证与确认会议研讨会（ICSTW'13）论文集》，第321–330页，卢森堡卢森堡市，2013年3月。IEEE。74 Zhenlian Peng、Jian Wang、Keqing He 和 Hongtao Li。《一种基于概率网络节点中心性的特征优先级划分方法》。载于 Georgia Kapitsaki 和 Eduardo Almeida（编），《第15届国际软件重用会议（ICSR'15）论文集》，LNCS 9679，第106–121页，塞浦路斯利马索尔，2015年6月。施普林格。75 Gilles Perrouin、Sagar Sen、Jacques Klein、Benoît Baudry 和 Yves le Traon。《软件产品线自动化且可扩展的T-Wise测试用例生成策略》。载于 Ana Rosa Cavalli 和 Sudipto Ghosh（编），《第3届国际软件测试、验证与确认会议（ICST'10）论文集》，第459–468页，法国巴黎，2010年4月。IEEE。76 Benjamin C. Pierce。《类型与编程语言》。麻省理工学院出版社，2002年2月。77 Richard Pohl、Kim Lauenroth 和 Klaus Pohl。《面向特征模型自动化分析操作的当代算法方法的性能比较》。载于 Corina păsăreanu 和 John Hosking（编），《第26届国际自动化软件工程会议（ASE'11）论文集》，第313–322页，美国堪萨斯州劳伦斯，2011年11月。IEEE。78 Christian Prehofer。《面向特征编程：重新审视对象》。载于 Mehmet Akşit 和 Satoshi Matsuoka（编），《第11届欧洲面向对象编程会议（ECOOP'97）论文集》，计算机科学讲义1241，第419–443页，芬兰赫尔辛基，1997年6月。施普林格。79 Xiao Qu、Myra B. Cohen 和 Gregg Rothermel。《配置感知回归测试：抽样与优先级划分的实证研究》。载于 Andreas Zeller（编），《第17届国际软件测试与分析研讨会（ISSTA'08）论文集》，第75–86页，美国华盛顿州西雅图，2008年7月。ACM。80 Ana B. Sánchez、Sergio Segura、José A. Parejo 和 Antonio Ruiz-Cortés。《真实环境中的可变性测试：Drupal案例研究》。《Software and Systems Modeling》，16(2):173–194，2017年4月。81 Ana B. Sánchez、Sergio Segura 和 Antonio Ruiz-Cortés。《软件产品线测试用例优先级排序准则的比较》。载于 Lauric Williams 和 Claes Wohlin（编），《第7届国际软件测试、验证与确认会议（ICST'14）论文集》，第41–50页，美国俄亥俄州克利夫兰，2014年3月。IEEE。82 John R. Seeley。《互惠影响网络；处理社会测量数据时的一个问题》。《Canadian Journal of Psychology》，3(4):234–240，1949年12月。83 Christoph Seidl、Tim Winkelmann 和 Ina Schaefer。《由特征建模符号与跨树约束语言构成的软件产品线》。载于 Andreas Oberweis 和 Ralf H. Reussner（编），《第13届Modellierung会议（Modellierung'16）论文集》，LNIP-254，第157–172页，德国卡尔斯鲁厄，2016年3月。施普林格。84 Norbert Siegmund、Alexander Grebhanhn、Sven Apel 和 Christian Kästner。《高度可配置系统的性能影响模型》。载于 Mark Harman 和 Patrick Heymans,

23:28 基于编译器的精细特征排名：优先考虑配置相关性

编者，第十届软件工程基础联合会议论文集 (ES- EC/FSE’15)，第 284–294 页，意大利贝加莫，2015年9月。ACM。85 Julio Sincero、Horst Schirmeier、Wolfgang Schröder-Preikschat 和 Olaf Spinczyk。《Linux 内核是软件产品线吗？》载于 Frank van der Linden 和 Björn Lnndell 编，第 2 届开源软件与产品线国际研讨会论文集 (SPLC-OSSPL’07)，日本京都，2007年9月。86 Julio Sincero、Reinhard Tartler、Daniel Lohmann 和 Wolfgang Schröder-Preikschat。《基于预处理器的可变性的高效提取与分析》。载于 Jaakko Järvi 编，第九届生成式编程与构件工程国际会议论文集 (GPCE’10)，第 33–42 页，荷兰埃因霍温，2010年10月。ACM。87 Niklas Sörensson 和 Niklas Eén。MiniSat v1.13——一种带冲突子句最小化功能的 SAT 求解器。载于 Fahiem Bacchus 编，第八届可满足性测试理论与应用国际会议论文集 (SAT’05)，LNCS 3569，第 1–2 页，苏格兰圣安德鲁斯，2005年6月。施普林格。海报。88 Sabrina Souto 和 Marcelo d’Amorim。面向可配置系统的高效回归测试。《Journal of Systems and Software》，137:733–746，2018年3月。89 Bjarne Stroustrup。《C++ 的设计与演进》。Addison-Wesley，第一版，1994年3月。90 Reinhard Tartler、Christian Dietrich、Julio Sincero、Wolfgang Schröder-Preikschat 和 Daniel Lohmann。《系统软件中可变性的静态分析：90,000 个 #ifdef 条件编译指令问题》。载于 Garth Gibson 和 Nickolai Zeldovich 编，USENIX 年度技术会议论文集 (USENIX ATC’14)，第 421–432 页，美国宾夕法尼亚州费城，2014年6月。USENIX 协会。91 Reinhard Tartler、Daniel Lohmann、Julio Sincero 和 Wolfgang Schröder-Preikschat。《编译时可配置的系统软件中的特征一致性：面对 Linux 一万特征问题》。载于 Gernot Heiser 编，第六届计算机系统会议论文集 (EuroSys’11)，第 47–60 页，奥地利萨尔茨堡，2011年4月。ACM。92 Matthew Treinish、Ivan Carvalho、Georgios Tsilimigkounakis 和 Nahum Sá。rustworkx：用于 Python 的高性能图形库。《开放源代码软件杂志》，7(79):3968:1–3968:32，2022年11月。93 David A. Turner。Miranda：一种具有多态类型的非严格函数式语言。载于 Jean-Pierre Jouannaud 编，第一届函数式编程语言与计算机体系结构国际会议论文集 (FPCA’85)，LNCS 201，第 1–16 页，法国南锡，1985年9月。施普林格。94 Miguel Velez、Pooyan Jamshidi、Florian Sattler、Norbert Siegmund、Sven Apel 和 Christian Kästner。ConfigCrusher：迈向可配置系统的白盒性能分析。《自动化软件工程》，27:265–300，2020年8月。95 Miguel Velez、Pooyan Jamshidi、Norbert Siegmund、Sven Apel 和 Christian Kästner。《机器学习之上的白盒分析：对可配置系统性能进行建模》。载于第 43 届国际软件工程会议论文集 (ICSE’21)，第 1072–1084 页，西班牙马德里，2021年5月。IEEE。96 Miguel Velez、Pooyan Jamshidi、Norbert Siegmund、Sven Apel 和 Christian Kästner。《调试可配置软件系统的性能：开发者需求与定制化工具支持》。载于 Daniela Damian 和 Andreas Zeller 编，第 44 届国际软件工程会议论文集 (ICSE’22)，第 1571–1583 页，美国匹兹堡，2022年5月。ACM。97 S. Venkatakeerthy、Rohit Aggarwal、Shalini Jain、Maunendra Sankar Desarkar 和 Ramakrishna Updrasta。IR2Vec：基于 LLVM IR 的可扩展程序嵌入。《ACM 体系结构与代码优化汇刊》，17(4):32:1–32:27，2020年12月。98 Philip Wadler。线性类型可以改变世界！载于 Manfred Broy 和 Cliff B. Jones 编，第二届编程概念与方法工作会议论文集 (IFIP’90)，第 561–582 页，以色列加利利海，1990年4月。North-Holland。

99 Claes Wohlin、Per Runeson、Martin Höst、Magnus C. Ohlsson、Björn Regnell 和 Anders Wesslén。软件工程实验。施普林格，2012。