Prioritizing Configuration Relevance via Compiler-Based Refined Feature Ranking

Federico Bruzzone [a], Walter Cazzola [a,*], Luca Favini [a]

[a] *Università degli Studi di Milano, Computer Science Department, Milan, Italy*

Abstract

Modern programming languages, most notably Rust, offer advanced linguistic constructs for building highly configurable software systems as aggregations of features—identified by a configuration. However, they pose substantial challenges for program analysis, optimization, and testing, as the combinatorial explosion of configurations often makes exhaustive exploration infeasible. In this manuscript, we present the first compiler-based method for prioritizing configurations. Our approach consists of four main steps: 1. extracting a tailored intermediate representation from the Rust compiler, 2. constructing two complementary graph-based data structures—a *feature dependency graph* capturing inter-feature code-level dependencies weighted by predicate structure, and an *atom dependency tree* encoding the extent of code governed by each feature—, 3. using centrality measures to rank features, and 4. refining the ranking by considering the extent of code they impact. A fixed number of most relevant configurations are generated based on the achieved feature ranking. The validity of the generated configurations is guaranteed by using a SAT solver that takes a representation of feature dependencies in conjunctive normal form. We formalized this approach and implemented it in a prototype, RustyEx, by instrumenting the Rust compiler. An empirical evaluation on higher-ranked open-source Rust projects shows that RustyEx efficiently generates user-specified sets of configurations within bounded resources, while ensuring soundness by construction. The results demonstrate that centrality-guided configuration prioritization enables practical exploration of large configuration spaces, and we discuss the implications and open challenges for future configuration-aware analysis and optimization.

Keywords: Highly Configurable Systems, Testing Variable Systems, Configuration Prioritization via Static Analysis, Rust.

1. Introduction

Premise. Highly configurable software systems[1] often aim to satisfy a wide range of requirements. These systems provide a set of features that can be combined in different ways to create a variety of products [4]. Variability-rich software system development leverages principles from product line engineering, commonly referred to as *feature-oriented programming* [79].[2] Programming languages, such as C/C++ and Java, provide mechanisms to manage software variability via preprocessor directives, conditional compilation, and annotations. Rust [60] has also been designed to support the development of highly configurable software systems via its *attribute system*[3]—specifically, the `cfg` attribute.

Problem Statement. Software variability can be complex, making it challenging to reason about all possible configurations [1]. This problem is further exacerbated by the fact that the number of possible configurations grows exponentially with the number of features, as demonstrated by Krueger [46]. For instance, the Linux kernel is a well-known highly configurable software system with a significant number of features [86] that exploits the native support for variability in C/C++ and, more recently, Rust [92]. Over the years, the kernel and similar systems have been extensively studied, with numerous works highlighting the challenges posed by large configuration spaces [63, 91, 25, 59].

State of the Art Exhaustive approaches become infeasible due to the combinatorial explosion of configurations [5]. Therefore, a strategy is needed to reduce the number of configurations under analysis. Several approaches aim to achieve this goal. Sampling techniques [73, 2, 48], combinatorial testing [71, 55], and in particular t-wise testing [20, 75, 71] attempt to reduce the configuration space while ensuring coverage. Other works have proposed prioritization methods [27, 82, 72, 25], using criteria such as similarity [2], non-functional properties [81], or centrality-based measures [64, 74, 50, 6]. As reported by Classen *et al.* [18], many configurable systems are safety-critical, which makes prioritization even more important: the goal is to detect faults as early as possible while reducing the number of tests.

Limitations of Existing Approaches. Despite these efforts, the problem of finding a proper prioritization criterion remains open. Existing approaches 1. are predominantly based on fea-

*Corresponding author.

Email addresses: `federico.bruzzone@unimi.it` (Federico Bruzzone), `cazzola@di.unimi.it` (Walter Cazzola), `luca.favini@studenti.unimi.it` (Luca Favini)

[1] These systems are also known as *product families* or *software product lines* (SPLs) [19]

[2] We use this term loosely as a collective label for variability-implementation paradigms, not exclusively for the FOP paradigm of Prehofer

[3] Rust's attributes are a form of syntactic metadata that can be attached to various parts of the code. Two kind of attributes are supported: *outer* and *inner*. The former is used to annotate *item* declarations (such as functions, structs, and enums), *expression* and *statement*. The latter is used to annotate *modules* and *block expressions* (in particular cases). We will refer to the *items*, *expressions*, and *statements* as *terms*. See `https://doc.rust-lang.org/reference` for more details.

ture models [9, 77] and, while some recent work considers source artifacts or presence conditions [34, 44], none combines code-extent weighting with centrality-based ranking in a compiler-integrated pipeline, and 2. do not account for the extent of the code affected by each feature. Consequently, configurations that are critical to the system's behavior may be omitted. Moreover, the need for principled configuration prioritization extends well beyond testing. Highly configurable systems must also address:

1. *compiler optimizations* and *performance analysis*, where either only a subset of variants can be feasibly evaluated, or configuration relevance can guide optimization strategies [95, 85, 90, 96],
2. *program comprehension* and *debugging*, where developers must reason about representative or critical configurations [33, 97],
3. *variability management* and *refactoring*, where structural changes should be guided by the actual impact of features in the code [54, 53], and
4. *regression analysis*, where prioritization helps identify which configurations are most likely to reveal behavioral differences after evolution [80, 89].

In all these contexts, treating features uniformly or relying on stochastic heuristics may be insufficient, particularly in safety-critical and performance-sensitive settings where specific feature interactions must be systematically explored [18].

Proposal. In this paper, we propose, to the best of our knowledge, the first general method for prioritizing the relevance of configurations via a compiler-based refined ranking of features. To this end, we present RustyEx, a fully automated tool designed to identify the *most relevant configurations* in highly configurable Rust software systems.[4] We define these configurations as those containing the most relevant features. A feature's relevance is measured by its centrality in the *feature dependency graph* and the extent of the code it impacts. As shown in Figure 1, we instrument the Rust compiler by performing an interprocedural static analysis [68, 69][5] to extract the *unified intermediate representation* (UIR). The UIR is obtained by removing irrelevant information from the Rust *abstract syntax tree* (AST) and by creating the *atoms*.[6] RustyEx performs static analysis on the UIR to extract two main data structures: the *dependency graph* [61] and the *polytree* [23]. The former is a weighted directed graph, dubbed *feature dependency graph* (see ❷ and ❷ in Figure 2), that represents the dependencies between the features. The latter is an induced subgraph of the UIR, dubbed *atom dependency tree* (see ❸ and ❸ in Figure 2). This structure captures the *lexical scope*-based [21] dependencies between atoms, enriched with the extent of the code they affect. To address the previously mentioned limitations, we propose a method that combines both structures to identify the most relevant configurations. This method leverages centrality measures as structural metrics [28, 62, 66], and includes graph transformations into propositional and conjunctive normal form (CNF) [18] formulas, along with techniques for refining the ranking of features. RustyEx relies on a SAT solver to determine configurations that satisfy the CNF formula based on the refined feature ranking. This ensures that not only is the number of configurations reduced, but also that the most relevant configurations are prioritized. We validate our approach on higher-ranked open-source Rust projects by running RustyEx with a fixed number of configurations to generate. To provide a comprehensive evaluation, we report detailed metrics on the proposed structures for each project. Soundness is ensured by construction, guaranteeing that all generated configurations are valid.

Contributions. The main contributions of this paper are:

- the first general method for prioritizing configurations in highly configurable software systems via a compiler-based refined ranking of features,
- a formalization of the approach, from the extraction of the UIR to the algorithms for building the complementary data structures and prioritizing configurations,
- a detailed implementation of our approach in RustyEx, a fully automated tool for Rust software,
- an extensive evaluation of RustyEx on higher-ranked open-source Rust projects, demonstrating its effectiveness in identifying and prioritizing the most relevant configurations, and
- a formal proof of the soundness of our approach, ensuring that all generated configurations are valid.

Manuscript Structure. The remainder of the paper is organized as follows. Section 2 introduces the necessary background. Section 3 details the design of RustyEx, and Section 4 presents our evaluation. Section 7 discusses related work, and Section 8 concludes the paper.

2. Background

We provide background on Rust's configuration system and on the centrality measures used in this work.

2.1. The Rust Programming Language

Rust is a system programming language that focuses on safety, speed, and concurrency [60]. Rust supports software variability through its attribute system, particularly the `cfg` attribute, which allows conditional compilation based on specified configuration predicates. A *feature* in Rust is defined by all *terms* (i.e., items, expressions, and statements) annotated with the same *config name* in a `cfg` outer attribute. The inclusion of a feature in a product depends on the evaluation of a *configuration predicate* at compile time, e.g., `#[cfg(any(unix, windows))]`. `Cargo`—Rust's package manager—lets developers declare features using *config names* in the `Cargo.toml` file,

[4] Although we use Rust in this work, our approach to configuration prioritization is language-agnostic and applicable to any language with native variability support, such as C/C++ or Java.

[5] Inter-procedural analysis is a static analysis technique that analyzes the control and data flow of a program across multiple functions.

[6] From now on, we refer to an *atom* as a single *term* annotated with a given *configuration predicate*.

combining them with logical operators into *configuration predicates*. Feature dependencies, i.e., *cross-tree constraints* of *feature models* [39, 84, 40], can also be specified in the same file. Unlike languages such as C/C++ that rely on preprocessor macros, Rust's `cfg` attributes are integrated into the compiler's type-checking and name-resolution pipeline, making them available for principled static analysis without macro expansion artifacts.

2.2. Centrality Measures

Since the early days of social network analysis, *centrality measures* have been used to identify the key nodes in a network [83, 8, 41]. They have since become central to graph theory and network analysis [16, 22, 47]. A *network* is typically modeled as a graph with n nodes, indexed by $i \in \{1, 2, \ldots, n\}$, and represented as an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where $A_{ij} \neq 0$ indicates an edge between nodes i and j, and $A_{ij} = 0$ otherwise. As Bloch *et al.* [13] observe, not all centrality measures consider edge directions and weights, though these can be incorporated. Formally, a centrality measure is a function $\phi : \mathbb{R}^{n \times n} \to \mathbb{R}^n$, where $\phi_i(A)$ quantifies the relevance of node i in the network A. As a cardinal invariant, it enables ordinal ranking of nodes based on their scores ($\phi_i(A)$). Centrality measures are broadly categorized as: *geometry-based* and *spectral-based* [14]. In the remainder of this section, we introduce the most commonly used centrality measures.

Geometric Centrality Measures. The *geometric* centrality measures assign relevance by using a function of distances—i.e., the number of nodes at each distance from a given node. *Degree centrality* counts the number of edges connected to a node: $\mathrm{d}^-(i)$ for in-degree and $\mathrm{d}^+(i)$ for out-degree. In a feature dependency graph, a feature with high in-degree is depended upon by many other features, suggesting it plays a central role in the configuration space. *Closeness centrality*, introduced by Bavelas [8], defines a node's relevance as inversely proportional to the sum of its distances to all other nodes. Since the original formulation fails on disconnected graphs (due to infinite distances), a widely adopted variant is:

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

where $d(i, j)$ denotes the shortest path between nodes i and j. In a feature dependency graph, a feature with high closeness centrality can quickly "reach" all others through dependency chains. *Harmonic centrality* [58] is defined as the sum of the reciprocals of the shortest path lengths between a node and all others:

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

Unlike closeness centrality, it is well-defined for disconnected graphs; for isolated nodes, the sum evaluates to 0 since $d(i, j) = \infty$ for all $j \neq i$. *Betweenness centrality* [31] measures how often a node appears on the shortest paths between pairs of nodes:

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

where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(i)$ is the number of those paths passing through node i. A feature with high betweenness sits at structural crossroads: enabling or disabling it affects the reachability of many other feature combinations. Newman [65] extended *closeness centrality* to weighted networks using Dijkstra's algorithm to compute shortest paths. Opsahl *et al.* [70] further generalized shortest-path measures by combining edge weights and counts, making them applicable to both *closeness* and *betweenness* centrality.

Spectral Centrality Measures. The *spectral* centrality measures evaluate node relevance using the dominant left eigenvector of the graph's adjacency matrix. *Eigenvector centrality* [15] assigns higher scores to nodes connected to other high-scoring nodes. It is defined as the eigenvector $\boldsymbol{v}$ associated with the largest eigenvalue λ of the adjacency matrix A:

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

If A is a *stochastic matrix*, the dominant eigenvalue is 1. However, the *eigenvector centrality* performs poorly on disconnected graphs [11]. In a feature dependency graph, a feature with high eigenvector centrality is not only itself important, but is depended upon by other important features—a property that degree centrality alone cannot capture. *Katz centrality* [41] addresses the disconnected-graph limitation by considering the number of paths of varying lengths that connect a node to all other nodes. It is defined as:

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

where $\mathbf{1}$ is a vector of ones, I is the identity matrix, and β is a damping factor satisfying $\beta < 1/\lambda$, with λ the dominant eigenvalue of A.

3. A Deep Dive into RustyEx

As introduced in Section 1, RustyEx is a fully automated tool that instruments Rust compiler to: 1. extract feature dependencies in Rust software, 2. assign feature weights based on their impact on the code, 3. apply centrality measures to rank features, and 4. prioritize configurations.

3.1. Process Overview

RustyEx performs its analysis in three main phases: 1. process setup, 2. dependency extraction, and 3. configuration generation, as shown in Figure 1.

Process Setup. Rust projects make extensive use of `cfg` attributes for conditional compilation, both for *feature gating*—enabling or disabling features via `Cargo`—and for purposes like platform-specific code selection [3, 17, 51]. To ensure usability, RustyEx integrates seamlessly with the Rust toolchain. In this

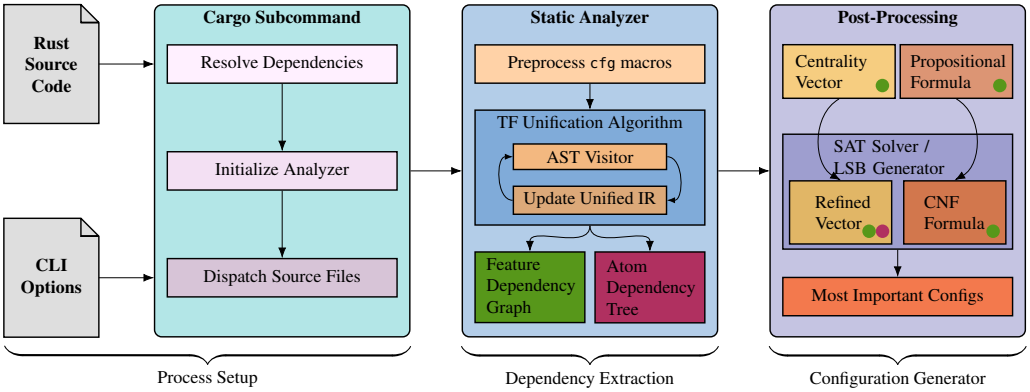


Figure 1: The phases of RustyEx (inspired from [52])

phase, it accepts a Rust project along with CLI options that customize the analysis, such as the number (N) of configurations to generate and the chose centrality measure. The setup uses `Cargo` to resolve internal crate dependencies which are essential for the analysis.[7] It then stores the parsed analysis configuration (centrality measure, number of desired configurations N, and crate list) in a shared in-memory state accessible to all compiler plugin instances spawned by `Cargo`, and configures the dispatch of source files to use the instrumented `rustc`.

Dependency Extraction. RustyEx instruments the Rust compiler to extract the *feature dependency graph* (see ❷, and ❷ in Figure 2) and the *atom dependency tree* (see ❸, and ❸ in Figure 2) from the UIR derived from the AST. The details of these structures and their extraction are discussed in Section 3.2. This is integrated into the compiler by implementing the `rustc_driver_impl::Callback` trait, which provides four hooks: `config`, `after_crate_root_parsing`, `after_expansion`, and `after_analysis`.[8] `rustc` performs the *eager* macro expansion[9] after AST creation and before name resolution. Although, `after_crate_root_parsing` would be ideal for analysis before macro expansion, it lacks access to sub-modules that have not been resolved yet. Thus, we hook into `after_expansion`, which is invoked after *eager* macro expansion, name resolution, and AST validation. To avoid premature evaluation of `cfg` attributes, we register a custom file loader in the `config` callback. This loader renames `cfg` macros to delay their evaluation. In the `after_expansion` callback, we perform inter-procedural analysis on the AST to extract the UIR weighting its nodes with the *weighted* fixed-point algorithm shown in Algorithm 1 described in Section 3.2. These weights estimate each feature's impact on the code.

Configuration Generation. Once the weighted feature dependency graph and the atom dependency tree are extracted, RustyEx uses them to identify the most relevant configurations. The weighted feature dependency graph serves two key tasks: 1. ranking features via *geometry-based* and/or *spectral-based* centrality measures (top-left box in the post-processing phase of Figure 1), and 2. generating a corresponding propositional formula (top-right box in the post-processing phase of Figure 1). These tasks are detailed in Section 3.3. The propositional formula is then converted to CNF. Next, the atom dependency tree refines the feature ranking (middle boxes in the post-processing phase of Figure 1). Finally, a SAT solver selects the top N configurations that satisfy the CNF formula, prioritizing those with the highest refined ranking—i.e., the most relevant configurations (bottom box in the post-processing phase of Figure 1).

3.2. From AST to UIR

Overview. In the AST, the `cfg` attribute nodes and their associated *terms* appear as separate child nodes under a common ancestor. Building the UIR involves two key steps: *unification* and *weighting*. The term UIR stems from the *unification* step, which merges each `cfg` attribute with its associated *term* into a single node called an *atom*. UIR nodes are algebraic data types [49, 76, 94, 42, 10], specifically Σ-types—i.e., nodes can be either *atoms* or *relevant* plain AST nodes. Plain AST nodes are included in the UIR only if the *weighting* step assigns them non-zero relevance—e.g., generic bounds on parameters are usually excluded. For instance, in ❶ and ❶ of Figure 2, the function `foo` and its `#[cfg(feature = "a")]` attribute on line 1 are unified into a single *atom* node, centered on the annotated item `foo`. Conversely, the body nodes of the function `bar` remain plain AST nodes in the UIR because they contribute to its weight, helping quantify how much of the code is influenced by the involved `cfg` features.

Formalization. We define the AST as a pair (N_{ast}, E_{ast}), where N_{ast} and E_{ast} are the sets of nodes and edges, respectively. Let $N_{rel} \subseteq N_{ast}$ denote the set of *relevant* nodes. A node is considered *relevant* if it contributes to the semantics of a feature-dependent element. Specifically, `let` statements are relevant when they appear within the lexical scope of a `cfg`-annotated ancestor, as they contribute to the code extent of that scope; `let` statements in unconditional global scope with no `cfg` ancestor are excluded. Conversely, `extern crate` and `use` declarations

[7] For performance, RustyEx analyzes only the target crate's source files and compiles external dependencies with the standard Rust compiler. This reduces computational overhead but may slightly impact accuracy [98].

[8] See `https://doc.rust-lang.org/nightly/nightly-rustc/` for details.

[9] Eager expansion expands macro arguments as early as possible, regardless of the macro invocation.

are never considered relevant, as they do not carry behavioral semantics for the purpose of feature ranking.

The set of *atoms* is defined as:

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

where:

1. P is the set of all `cfg` predicates, each defined as a recursive Σ-type with the following structure:

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

2. $T \subseteq N_{ast}$ is the set of all *terms*, and
3. $ann : T \to P$ is the surjective function that maps terms to `cfg` predicates.

The UIR is an *enhanced* induced subgraph of the AST, where nodes are enriched with `cfg` predicates and edge directions are reversed. Formally, the UIR is defined as $\mathcal{U} = (N, E, w_N)$, where:

- $N = A \cup N_{rel}$ with $A \cap N_{rel} = \emptyset$, contains both *atoms* and *relevant* AST nodes, forming a Σ-type structure,
- $E = \{(i, j) \mid (i, j) \in E'\}$, where $E' \subseteq E_{ast}$ is the set of reversed edges derived from the AST, and
- $w_N : N \to \mathbb{N}^+$ assigns weights to nodes based on the type of *term*.

Since UIR nodes may be compound Σ-types (e.g., atoms), $E' \subseteq E_{ast}$ alone does not capture all necessary structure. To preserve connectivity, if $(i, j) \in E_{ast}$ and there exists $a \in A \subseteq N$ s.t. $i = term(a)$ or $j = term(a)$, we extend E' as:

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

Here $term : A \to T$ extracts the term from an *atom*. This ensures that when a node is unified into an *atom*, its original AST edges are preserved in the UIR, as if the node remained *relevant* on its own.

Unification Algorithm. The *unification algorithm* performs a *depth-first visit* traversal of the AST to identify `cfg` attributes and their associated *terms*, merging them into *atoms*. RustyEx implements the visitor pattern [32] via the `rustc_ast::visit::Visitor` trait. To track parent-child relationships during traversal, it maintains a *term stack*, where each visited term is pushed. Since `cfg` attributes always appear as leftmost children of terms, encountering one triggers the *unification* process. This involves:

1. extracting the configuration predicate from the `cfg` attribute,
2. parsing the predicate into a custom internal representation, dubbed `ComplexFeature`, and
3. popping the corresponding term from stack to *unify* it with the predicate into an *atom* node.

For example, the `cfg` attribute on line 3 of ❶ of Figure 2 is parsed as

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

and unified with the term `bar` to form the corresponding *atom* node. Once a term is fully visited, it is popped from the stack and an UIR edge is added from the child to its parent. Unlike the AST, UIR edges are reversed to reflect lexical scope-based relationships, capturing dependency flow—key for applying graph centrality measures in the feature dependency graph (Sect. 3.3). The `bar` node in the atom dependency tree (❷ in Figure 2) shows its unified features, highlighted by the ● and ● markers, emphasizing both unification and edge direction.

Weighting Algorithm. Each node in the UIR is assigned a positive integers, i.e., elements of $\mathbb{N}^+$. During the *unification* phase, RustyEx assigns to every UIR node a *weight kind*, based solely on the kind of *term* it represents. These *weight kinds* determine how weights are computed in Algorithm 1 (lines 16–25), and include:

1. *no weight*: nodes irrelevant to the analysis that receive no weight (e.g., `extern crate` declarations),
2. *intrinsic weight*: nodes assigned a fixed weight of one (e.g., `let` statements),
3. *children weight*: nodes whose weight is the sum of their children weights (e.g., `foo` in ❶ of Figure 2), and
4. *reference weight*: nodes that refer to other nodes in the UIR and inherit their weight (e.g., the `bar()` call in `qux` in ❶ of Figure 2).

After *unification*, RustyEx runs a fixed-point algorithm (Algorithm 1) to compute the final weight function w_N over UIR nodes. In Rust, multiple *terms* can share the same identifier only if their `cfg` attributes are mutually exclusive. Let o_n denote the number of distinct *atoms* a node $n \in N$ appears in. The algorithm initializes:

- $m_w : N \to \mathbb{N}^{+o_n}$, a map storing all weights associated to the node identifier,
- an empty queue Q,
- the root node $r = root(\mathcal{U})$, and
- the transposed UIR $\mathcal{U}' = \mathcal{U}^T$, since the original UIR is built bottom-up and transposing it reestablishes parent-child relationships.

The *weighting* algorithm starts at the root node r, invoking the recursive CALC_WEIGHT function (line 17), which computes the weight of each node $n \in N$ by traversing its children in $\mathcal{U}'$ and applying the rules associated with its *weight kind*. Computed weights are stored both in the node and in the map m_w for reuse. For each node $n \in N$ of kind *reference*, such as function calls, the algorithm checks whether the referenced node already has a computed weight in m_w. If the weight is not yet

Algorithm 1 Calculate The Weight of the UIR Nodes

```
1:  m_w ← ∅
2:  Q ← ∅
3:  r ← root(U)
4:  U' ← U^T                                              ▷ transpose
5:  CALC_WEIGHT(U', r, Q, m_w)
6:  RESOLVE_QUEUE(U', Q, m_w)
7:  U ← U'^T                                              ▷ transpose back

8:  function CALC_WEIGHT(U', n, Q, m_w)
9:      for all adj ∈ ADJ(U', n) do                       ▷ adjacency nodes
10:         CALC_WEIGHT(U', adj, Q, m_w)
11:     end for
12:     ch_w ← 0                                          ▷ children weight
13:     for all adj ∈ ADJ(U', n) do
14:         if adj.status = WAIT then                     ▷ check the status
15:             n.status ← WAIT; ret
16:         end if
17:         ch_w ← ch_w + adj.weight
18:     end for
19:     match n.weight_kind with
20:         case NO: n.weight ← 0
21:         case INTRINSIC: n.weight ← 1 + ch_w
22:         case CHILDREN: n.weight ← ch_weight
23:         case REFERENCE(called):
24:             if m_w[called] = ∅ then                   ▷ no defs found
25:                 n.status ← WAIT
26:                 Q ← Q ∪ {n}
27:                 return
28:             else
29:                 t.weight ← AVG(m_w[called])
30:             end if
31:     end match
32:     n.status ← WEIGHTED
33:     m_w[n] ← m_w[n] ∪ n.weight
34: end function

35: function RESOLVE_QUEUE(U', Q, m_w)
36:     S ← ∅                                             ▷ set of seen nodes
37:     while Q ≠ ∅ do
38:         c ← Q ↓
39:         Q ← Q − {c}                                   ▷ dequeue
40:         if S ∩ {c, LEN(Q)} ≠ ∅ then                   ▷ recovery mechanism
41:             c.weight ← DEF_W                          ▷ assign a default weight
42:             continue
43:         end if
44:         S ← S ∪ {c, LEN(Q)}
45:         CALC_WEIGHT(U', c, Q, m_w)
46:         if c.status = WAIT then
47:             Q ← Q ∪ {c}
48:         end if
49:     end while
50: end function
```

available, the node is marked as *wait* and added to the queue Q. The fixed-point RESOLVE_QUEUE function (line 30) then processes Q, computing weights for the queued nodes until the queue is empty or a cycle is detected—e.g., due to direct or mutual recursion [67, 56]. A cycle is detected when Q remains unchanged between two complete iterations. To prevents infinite loops, the algorithm assigns a default fallback weight to nodes involved in cycles and continues processing the remaining queue.[10] Finally, the UIR is transposed back to its original direction (undoing the earlier transformation), and the finalized weights are stored in w_N.

3.3. From the UIR to the Feature Dependency Graph

Overview. Depicted in Figure 2 (❷, ❷), the feature dependency graph is a weighted directed graph that represents dependencies between software features. It is built by applying the *feature-extraction* algorithm to the UIR. This algorithm defines rules for assigning edge weights based on the *configuration predicates* of the `cfg` attributes in which the features appear. At a high level, the graph reflects lexical-scope-based relationships among features. Specifically, if f_i is used within the lexical scope of an *atom* annotated with f_j, then f_i depends on f_j and a directed edge is created from f_i to f_j. For instance, in line 3 of Figure 2 ❷, `feature = "b"` and `feature = "c"` appear in an `any` *configuration predicate*, within the lexical scope of an *atom* unified with the `feature = "a"` (line 1) and enclosing the `foo` function (line 2). According to the *feature-extraction* algorithm, two edges of weight 1 are created: from `feature = "b"` and from `feature = "c"` to `feature = "a"`. In contrast, at line 7 of ❷, `feature = "c"` is used in an `all` *configuration predicate* within the *global* lexical scope (denoted as G). Consequently, an edge of weight 1/2 is added from `feature = "c"` to G. As in the UIR, edge directions in the feature dependency graph are reversed compared to the original AST. The novel insight lies in preserving this natural edge direction: it allows us to interpret a feature's *reputation* (or relevance) based on the *state* (i.e., *configuration predicate*) of the features that depend on it. This enables the application of graph centrality measures on the feature dependency graph, where the edge weights are designed to support more accurate feature ranking.

Formalization. Until now, we referred to the feature dependency graph as a graph. However, during its construction, it is initially a *multigraph*, since a feature f_i may depend on another feature f_j in multiple parts of the code. Formally, this multigraph is denoted as $\mathcal{F} = (F, D, w_R)$, where:

1. $F = \{(f_i, p_i) \mid f_i \in CN \wedge p_i \subseteq P\}$ is the set of nodes, where CN is the set of all *feature names* and p_i is the set of all `cfg` predicates in which f_i is used;
2. $D = (U, m)$ is the *multiset* of directed edges representing feature dependencies, where $U = \{(i, j) \mid i, j \in F\}$ is the set of edges and $m : U \to \mathbb{N}^+$ is the *multiplicity* function that counts how many times a dependency occurs; and
3. $w_R : R \to \mathbb{R}$, where

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

is the weight function that assigns a weight to each individual edge instance, indexed by the multiplicity index k,

[10] Another approach has been proposed by instrumenting LLVM-IR [98], attempting to solve a system of linear equations that, by definition, could not admit solutions.

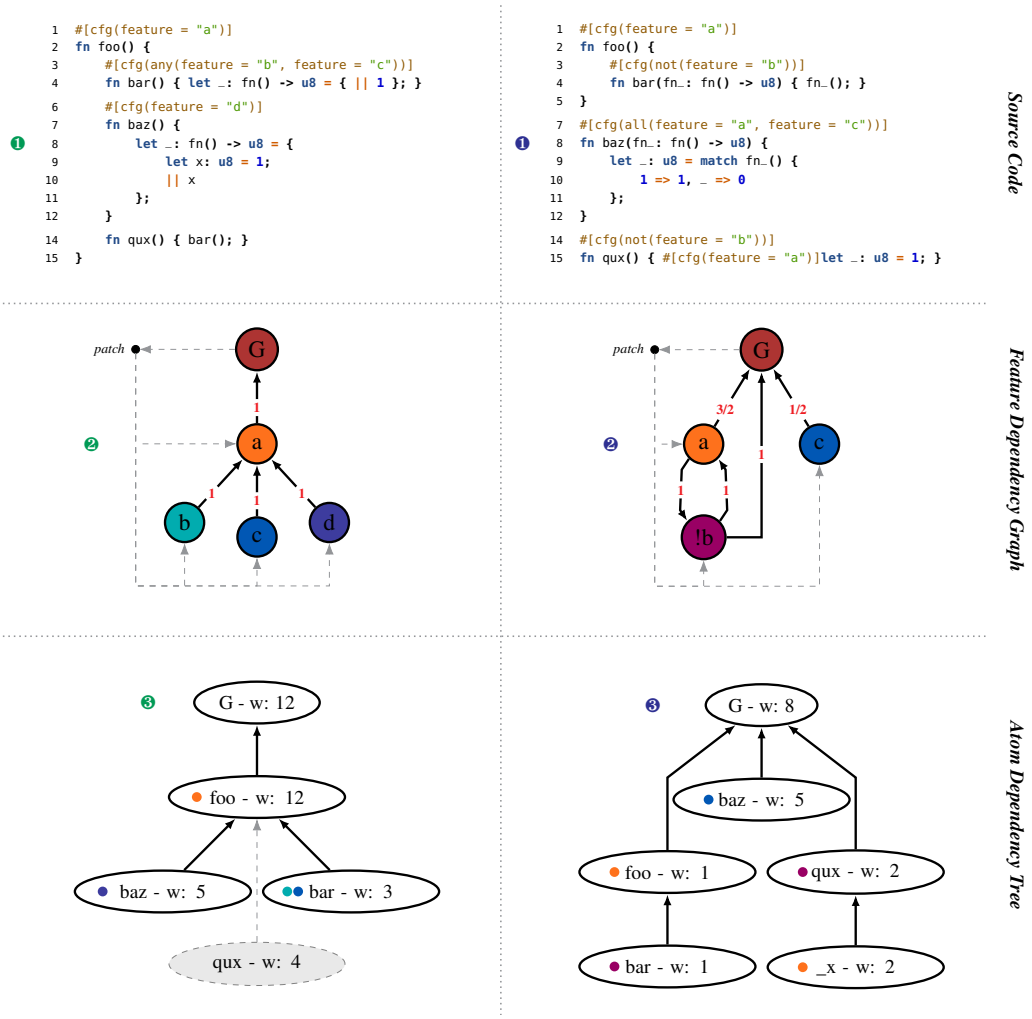


Figure 2: RustyEx process demonstrated on two scenarios

with weights determined by the *configuration predicates* in which the dependency arises.

The *feature-extraction* algorithm then squashes these multiple edges into single ones by aggregating their weights, effectively transforming the multigraph $\mathcal{F}$ into a simple graph $\mathcal{F}' = (F, D', w'_R)$, where:

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

is a set of unique edges obtained by removing the multiplicities from D, and the aggregated weight function is defined as:

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

Here, the sum aggregates the weights of all instances of the edge (i, j), with $m(i, j)$ denoting the total number of such instances. Each individual edge instance (i, j, k) contributes to the total weight assigned to the edge (i, j) in the final graph $\mathcal{F}'$.

Feature-Extraction Algorithm. The *feature-extraction* algorithm builds the feature dependency graph $\mathcal{F}'$ from the UIR. It begins by iterating over the UIR atoms $A \subseteq N$. For each atom $a \in A$, it retrieves its parent node $\hat{a} \in \{x \mid (\hat{a}, x) \in E \ \wedge \ x \in A\}$. The existence of $\hat{a}$ and uniqueness are guaranteed for all $a \in A$ such that $a \neq$ G, since the UIR is an induced subgraph of the AST, and every node in a tree has a unique parent. This follows from the fact that the UIR is an *enhanced* induced subgraph of the AST. Concretely, $\hat{a}$ is the nearest ancestor of a that is an *atom*. The algorithm creates a directed edge from each feature $f \propto pred(a) = p_a$ to each feature $\hat{f} \propto pred(\hat{a}) = p_{\hat{a}}$, where $\propto$ is read as "*involved in*" and $pred : A \to P$ returns the configuration predicate that annotates each *atom*. The predicate sets of the *feature nodes* f and $\hat{f}$ are updated as:

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

The weights of the *multi-edges* are derived from the nested `cfg` predicate of a. For each feature node $\hat{f}$ and each $(f, w) \in \chi(p_a, 1)$, the algorithm creates a weighted edge from f to $\hat{f}$, using the recursively defined function χ:

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

where $d^w_p = (w \times |\{f \propto p\}|)^{-1}$ and the $\oplus$ infix operator concate-

nates two lists. For example, in ❶ of Figure 2, the `feature = "b"` (line 3) appears in an `any` predicate nested under both `feature = "a"` (line 1) and `foo` (line 2), so an edge with weight 1 is added from `feature = "b"` to the `feature = "a"`. As discussed in Section 3.3, this weighting scheme prioritizes features in `any` predicates over those in `all` predicates, reflecting the fact that `any` predicates are less restrictive (only one feature needs to be enabled), whereas `all` predicates require all involved features to be enabled simultaneously. Finally, the graph is simplified by summing the weights of all edges with the same source and target nodes.

Centrality Measures. Once the feature dependency graph is built, RustyEx ranks features by relevance using a centrality measure specified via command-line argument (see Figure 1, marked with a ● inside the *Centrality Vector* box). Since $\mathcal{F}$ is a disconnected, weighted, directed graph, some centrality measures are either undefined or behave unpredictably [11]. For example, *eigenvector centrality* requires a connected graph [11] and both *closeness centrality* and *betweenness centrality* need adaptations to handle weighted directed graphs [65, 70].[11] To ensure connectivity, we introduce a synthetic *patch* node (see ❷ and ❷ in Figure 2). This node has a single incoming edge from G and outgoing edges to all other nodes, each with weight 1. This transformation guarantees that $\mathcal{F}$ is fully connected, allowing the following centrality measures to be applied without modification. Note that the patch node artificially elevates the centrality of features connected only to the global scope G: in the worst case, globally-scoped features may receive centrality scores higher than their structural importance warrants. In practice, this effect is bounded, since globally-scoped features are already among the most broadly applicable. Users wishing to study this effect in isolation can disable the patch node via the `-no-patch` flag.

- *Newman Closeness Centrality* [65]
- *Opshal Betweenness Centrality* [70]
- *Eigenvector Centrality* [15]
- *Katz Centrality* [41]

After computing centrality scores, a vector $\vec{v}$ is produced where each entry v_i represents the centrality of feature f_i. The *patch* nodes are excluded from the vector, as they are not *extracted* features. This approach aligns with the intuition behind our graph construction: a feature's reputation is defined by the features that depend on it. Centrality measures serve as a principled way to rank features by structural relevance.

Propositional Formula and CNF. From the feature dependency graph, RustyEx builds a *propositional formula* by iterating over each feature node $f_i \in F$. For each node, it generates a clause $\varphi_i = \ell(pred(f_i))$, where ℓ maps `cfg` predicates to *logical* expressions. It then adds an implication $f_i \Rightarrow f_j$ for each edge $(f_i, f_j) \in D'$, capturing feature dependencies. The overall formula φ is defined as:

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

where $\text{succ}(f_i)$ is the set of successors of f_i in $\mathcal{F}'$. Importantly, φ is *not* a global conjunction requiring all `cfg` predicates to be satisfied simultaneously. Each clause φ_i constrains f_i independently: if two code blocks carry conflicting predicates (e.g., `#[cfg(a)]` and `#[cfg(not(a))]`,they produce two separate feature nodes f_a and $f_{\neg a}$ with their own independent clauses. A satisfying assignment α selects one consistent truth-value per feature, so conflicting predicates are handled by the SAT solver as disjoint alternatives—never as a single unsatisfiable conjunction. For example, in Figure 2 ❶, activating `feature = a''` produces a configuration that includes the `foo` function and, through the implications $f_b \Rightarrow f_a$ and $f_c \Rightarrow f_a$, also forces `feature = b''` or `feature = c''` to be enabled if they appear. Finally, φ is converted to CNF formula φ_{cnf} [36]. Duplicate clauses may be removed before or after the CNF conversion.

3.4. From the UIR to the Atom Dependency Tree

Overview. Figure 2 (boxes ❸ and ❸) shows the atom dependency tree derived from the UIR. The name highlights its selective retention of UIR Σ-type nodes: only the *atom* variant is preserved, while *plain* AST nodes are removed. Specifically, only UIR nodes with `cfg` attributes are kept. For instance, in Figure 2, box ❸, the gray `qux` node (representing the `qux` function from line 14 in Figure 2, box ❶) is excluded because it lacks any `cfg` attribute. The atom dependency tree captures lexical scope-based dependencies among UIR *atoms*. Unlike the feature dependency graph, which flattens UIR structure, this tree preserves parent-child relationships between *atoms* and, by extension, between *features*. Each node is weighted by the amount of code affected by that atom's `cfg` condition, reflecting the feature's impact. To avoid losing information when *plain* AST nodes are discarded, the *atom dependency extraction* algorithm aggregates their weights into their nearest ancestor *atom* node (see `foo` node in Figure 2, box ❸). Finally, the centrality vector $\vec{v}$ is refined using the structure of the atom dependency tree to get a better ranking of feature relevance.

Formalization. Given the UIR definition $\mathcal{U} = (N, E, w_N)$, we define the atom dependency tree as a directed graph $\mathcal{A} = (A, E_A, w_A)$, where:

- $A = N \setminus N_{rel}$ is the set of *atom* nodes from the UIR,
- $E_A = E \setminus \{(i, j) \mid (i, j) \in E \wedge (i \in N_{rel} \vee j \in N_{rel})\}$ is the set of edges only connecting *atom* nodes, and
- $w_A : A \to \mathbb{N}^+$ is the weight function that assigns to each atom a weight reflecting the amount of code affected by the `cfg` attributes in which its features appear.

The atom dependency tree ($\mathcal{A}$) is an induced subgraph of $\mathcal{U}$, containing all *atoms* and their dependencies, but excludes *plain* AST nodes. This holds trivially since $A \subseteq N$ and $E_A \subseteq E$.

Atom Dependency Extraction Algorithm. The *atom dependency extraction* algorithm builds the atom dependency tree from the UIR. It iterates over the UIR atoms $A \subseteq N$. For each atom $a \in A$, it identifies its parent $\hat{a} \in \{x \mid (\hat{a}, x) \in E \ \wedge \ x \in A\}$. As before, the existence of $\hat{a}$ is trivial to prove. The algorithm then distinguishes two cases: 1. if $pred(a) \neq \emptyset$, it creates a edge from a to $\hat{a}$ and updates $\hat{a}$'s weight by adding a's weight, 2. if

[11] RustyEx uses `rustworkx` [93] to perform centrality measures.

$pred(a) = \emptyset$, it *does not* create an edge from a to $\hat{a}$ but still updates $\hat{a}$'s weight. For example, in Figure2, box ❸, the `qux` node contributes to the weight of `foo` but does not create an edge. The parent weight update is performed as:

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

Refinement Algorithm. The *refinement algorithm* refines the centrality vector $\vec{v}$ using the atom dependency tree $\mathcal{A}$ (see Figure 1). It iterates over each node $a \in A$ in $\mathcal{A}$, extracts every feature $f \propto pred(a)$, and updates its centrality value in $\vec{v}$ as:

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

Here $f \propto p$, with $p \in P$, means that f appears in the p-variant of a's `cfg` predicate, and $\overline{w_A(a)} \in [0, 1]$ is the normalized weight of the node a. The normalization prevents the inclusion of the existing centrality values in $\vec{v}$. Although $\mathcal{F}$ captures dependencies between features, it may fall short of expressing their actual relevance. The atom dependency tree $\mathcal{A}$ refines centrality values by incorporating the *extent* of code affected by each feature, yielding $\vec{v}'$ a new permutation of the vector $\vec{v}$.

3.5. Configuration Generation

Configuration generation is the final step of RustyEx. It produces the most relevant configurations based on the centrality vector $\vec{v}'$ and the CNF formula φ_{cnf} (see Figure 1). While more advanced lexical-scope-based strategies could be considered, RustyEx uses SAT solvers—both incremental (e.g., `MiniSat` [24, 88] and `CaDiCaL` [12]) and non-incremental (e.g., `Kissat` [12]). Given a fixed number K of configurations to generate, the algorithm iterates over the values in $\vec{v}'$. For each feature f_i, it updates the formula as:

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

Then, it uses a SAT solver to generate all the configurations satisfying φ_{cnf}. A SAT solver is then invoked to produce all satisfying configurations for the updated formula. If a satisfying configuration is found, it is added to the result set. This process repeats until K configurations are obtained, negating each newly found configuration to favor the discovery of new ones.

3.6. Correctness of Configuration Generation

Premises. To prove correctness, we show that: 1. the CNF encoding faithfully represents all configuration constraints, 2. the SAT solver returns only satisfying assignments, and 3. no invalid configuration is ever generated.

Construction of the CNF Formula. *Clause formation*—for each feature node $f_i \in F$, a clause $\varphi_i = \ell(pred(f_i))$ is created to enforce the logical interpretation of the feature's configuration predicate. That is, if a feature is enabled in some configuration, then its predicate must be satisfied. *Implication formation*—for every outgoing edge to f_j from f_i (i.e., for each f_j such that $(f_i, f_j) \in D'$), an implication $f_i \Rightarrow f_j$ is added to capture the dependency relations. This ensures that if feature f_i is active, then all its dependent f_j are also active. *Extension*—the formula φ is extended to include mandatory features specified in `Cargo.toml`, guaranteeing their presence in all generated configurations. *Equivalence Preservation*—since the CNF conversion uses standard techniques that preserve logical equivalence [36], any assignment that satisfies the CNF formula φ_{cnf} also satisfies the original formula φ.

Correctness of the SAT Solution. Given a SAT solver sound and complete,[12] any assignment α returned for φ_{cnf} satisfies every clause in φ_{cnf}, and thus also in φ. This implies:

1. for each feature f_i with clause φ_i, its activation under α satisfies its configuration predicate: $\alpha \models \varphi_i \quad \forall f_i \in F$—that is, α models the predicate;
2. for each dependency $f_i \Rightarrow f_j$, the implication holds under the assignment α if $\alpha(f_i) = \texttt{true} \Rightarrow \alpha(f_j) = \texttt{true}$;
3. all mandatory features from `Cargo.toml` are active in α: $\forall f_i \in F_{mandatory},\ \alpha(f_i) = \texttt{true}$.

By definition, a configuration is valid if and only if:

- it satisfies all the configuration constraints including feature predicates and dependencies, and
- it includes all mandatory features.

Therefore, since any assignment α produced by the SAT solver satisfies all clauses in φ_{cnf}, every configuration generated by RustyEx is valid.

Reinforcement by Contradiction. Assume, for contradiction, that the SAT solver returns a configuration α' that is invalid—that is, it violates one or more constraints. Then there exists at least one clause in φ_{cnf} that is *not* satisfied by α'. By definition, a *satisfying assignment* satisfies all clauses in the formula, and the soundness of the SAT solver guarantees that α' *is* a satisfying assignment for φ_{cnf}. This contradiction implies that our assumption is false: α' cannot be invalid. Hence, all configurations produced by the SAT solver are valid.

3.7. Design Rationale

We now justify the key design choices of RustyEx, discussing alternatives considered and their trade-offs.

Why a Feature Dependency Graph rather than a Feature Model? A *feature model* [39] is a declarative artefact that must be explicitly authored and maintained alongside the software. Many modern Rust projects do not provide a formal feature model; instead, variability is encoded implicitly through `cfg` attributes and `Cargo.toml` dependencies. The *feature dependency graph* (FDG) is inferred automatically from the source code and build metadata, requiring no additional modelling effort. Moreover, the FDG captures code-level transitive dependencies that a feature model typically does not represent—e.g., the fact that enabling feature b within the scope of feature a creates a dependency that a high-level FM would need to be explicitly encoded.

[12] A SAT solver is *sound* if it only returns assignments that satisfy the formula, and *complete* if it finds a solution whenever one exists.

Why an Atom Dependency Tree in addition to the FDG? The FDG records *which* features depend on each other, but not *how much code* each feature governs. Two features with identical FDG centrality may differ dramatically in their impact: one may gate a single statement while another gates an entire module. The *atom dependency tree* (ADT) fills this gap by tracking the weighted extent of code affected by each `cfg`-annotated atom. The refinement step (Section 3.4) adjusts centrality scores using these weights, yielding a ranking that is both structurally and quantitatively informed. A flat weight vector without the ADT topology was considered but discards the parent-child scoping information needed to correctly aggregate weights across nested `cfg` attributes.

Why Geometric and Spectral Centrality Measures? Simple degree centrality is insensitive to indirect influence: a feature with many transitive dependents may have low degree. We include both geometry-based measures (Newman closeness, Opsahl betweenness), which are robust to disconnected graphs and capture path-based influence, and spectral measures (eigenvector centrality, Katz centrality), which propagate relevance recursively through the graph. The choice of measure is exposed as a user parameter, because no single measure is universally optimal across graph topologies [14]. Offering a configurable portfolio lets practitioners match the measure to the characteristics of their project's dependency graph.

Why SAT Solving for Configuration Generation? Alternatives considered include random-walk enumeration and BDD-based enumeration. Random walks cannot guarantee coverage of the highest-ranked features. BDD-based approaches can enumerate configurations exhaustively, but their memory usage grows exponentially with the number of variables [4]. SAT solvers handle large CNF formulas efficiently in practice and can be guided by assumption literals to prioritize configurations that include the top-ranked features—exactly the behavior required. Both incremental solvers (`MiniSat`, `CaDiCaL`) and non-incremental solvers (`Kissat`) are supported, allowing users to trade-off between per-query overhead and overall throughput.

Why a Fixed Timeout per Crate? Analyzing each crate with an individual timeout (rather than a global project-level budget) ensures fair comparison across projects regardless of workspace size. A fixed per-crate timeout bounds the worst-case cost of any single analysis unit, enabling RustyEx to make progress on the remaining crates even when one crate is pathologically large or has missing dependencies.

3.8. Applications and Clarifications

The generation of configurations in RustyEx is performed incrementally and in a *lazy* fashion. The tool ranks all features according to their refined centrality in $\vec{v}'$ and then generates only the top K configurations most likely to cover critical feature interactions. In practice, this means prioritizing configurations that include the highest-ranked features in $\vec{v}'$, starting with $\vec{v}'_0$, the most central feature.

As discussed earlier, this prioritization strategy supports efficient exploration of the configuration space by focusing on variants that are most representative and impactful in terms of feature relevance. By analyzing these configurations first, developers can uncover important interactions and behaviors in the most relevant scenarios, improving overall system understanding and validation. Moreover, this targeted approach significantly reduces the computational cost and resource consumption associated with exhaustive analysis of all configurations, making it a practical solution for large-scale, highly configurable systems. For instance, these prioritized configurations could also serve as candidates for targeted performance profiling or static analysis, ensuring that critical feature combinations are examined before less relevant ones.

This configuration-aware pipeline highlights the potential of centrality-guided heuristics in feature-oriented analysis of configurable systems and paves the way for future work in optimizing configuration sampling based on structural and semantic program properties. It suggests a broader paradigm where program structure and semantics guide systematic exploration of the configuration space, potentially benefiting tasks such as performance evaluation or security analysis.

4. Evaluation

4.1. Research Questions

Our evaluation is organized around four research questions:

RQ1 (Soundness). Are all configurations generated by RustyEx valid with respect to the feature constraints of the analyzed project?

RQ2 (Efficiency and Scalability). Does RustyEx scale to large, real-world Rust projects in terms of execution time and memory usage?

RQ3 (Effectiveness). Do RustyEx-prioritized configurations outperform baselines in terms of fault-detection capability or code coverage?

RQ4 (Robustness). What is the failure rate of RustyEx across diverse Rust projects, and what are the root causes?

4.2. RQ1: Soundness

The soundness of RustyEx's configuration generation is guaranteed by construction, as formally proved in Section 3.6. Here we confirm the guarantee empirically: across all 1,628 crates successfully analyzed in our study, no invalid configuration was produced. Every generated configuration was verified to satisfy the corresponding CNF formula φ_{cnf} and to include all mandatory features declared in `Cargo.toml`.

Answer to RQ1. RustyEx generates exclusively valid configurations, both by formal construction and by empirical observation across the full evaluation dataset.

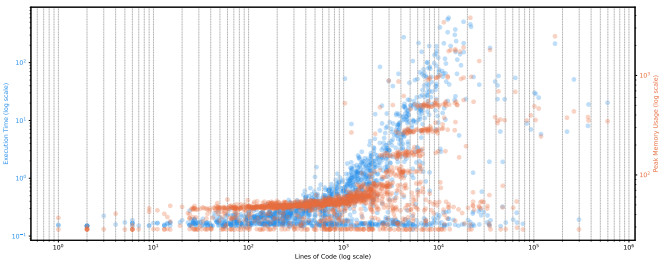
4.3. RQ2: Efficiency and Scalability

Experimental Setup. All experiments were conducted on a laptop equipped with an Intel Core i5-1135G7@2.40 GHz CPU

Table 1: Results of the experiments conducted on 40 Rust projects. All columns are aggregated per project, summing the values of all crates within the workspace, except for *Peak Memory Usage* and *UIR Height*, which are the maximum values.

Project Name	GitHub Stars	Crates.io Downloads	Lines of Code	Workspace Members	Failed Members	Dependency	Defined Features	UIR Nodes	UIR Edges	UIR Height	Feat. D.G. Nodes	Feat. D.G. Edges	Squashed D.G. Edges	Atom D.T. Nodes	Atom D.T. Edges	Execution Time	Memory Usage
rustdesk	81965	2826	108908	8	1	67	8	1353	1346	13	24	41	27	29	22	1 s	53 MB
gitoxide	9490	76429	223296	79	2	554	139	180166	180089	53	321	910	473	587	510	817 s	999 MB
deno	101658	419085	284483	36	3	561	8	123883	123850	31	124	1220	153	570	537	320 s	945 MB
tauri	89324	3840655	82089	26	2	255	65	26175	26151	30	78	193	89	144	120	108 s	902 MB
sway	62453	1092	210721	28	4	369	7	119724	119700	33	75	118	78	91	67	1156 s	1910 MB
fuel-core	57814	265047	148682	36	4	370	69	85528	85496	31	132	437	188	351	319	153 s	488 MB
alacritty	57640	192714	32812	5	1	39	6	27244	27240	24	17	47	25	36	32	117 s	513 MB
zed	54095	54231	591481	178	2	2458	91	23791	23615	28	381	268	237	231	55	91 s	497 MB
bat	51048	1496904	14971	1	0	30	9	675	674	19	5	14	7	8	7	26 s	339 MB
ripgrep	50241	938719	50285	10	2	40	8	60181	60173	27	43	158	69	124	116	320 s	887 MB
meilisearch	49152	1288	166851	19	2	196	30	44000	43983	30	46	69	41	55	38	404 s	2477 MB
fuels-rs	43907	5611	42484	24	2	90	22	43578	43556	28	70	165	78	138	116	161 s	691 MB
typst	37358	57580	112435	20	4	185	15	48913	48897	27	49	79	50	57	41	81 s	465 MB
helix	35745	103645	93042	14	1	143	12	55807	55794	28	59	175	89	133	120	595 s	980 MB
ruff	35669	4764	370658	38	3	402	23	124392	124357	51	130	237	161	176	141	462 s	504 MB
lapce	34961	35019	67802	5	2	73	3	20525	20522	41	16	44	25	32	29	195 s	948 MB
nushell	33821	975	291737	36	6	213	26	126096	126066	28	108	303	138	232	202	662 s	956 MB
polars	31805	2400845	376212	26	3	362	789	58368	58345	32	103	448	161	354	331	277 s	529 MB
swc	31649	1767375	648247	116	15	957	230	412313	412212	258	326	1122	369	722	621	2016 s	3425 MB
influxdb	29458	283996	54304	19	2	352	15	62082	62065	27	58	116	67	92	75	164 s	504 MB
tabby	29742	5637	41457	19	1	270	15	52972	52954	21	55	134	56	109	91	442 s	1827 MB
servo	29222	8792	367323	8	1	45	19	2670	2663	10	19	23	17	17	10	1 s	64 MB
wasmer	19335	4820432	263877	35	8	350	153	72860	72833	26	121	318	164	235	208	246 s	435 MB
diem	16702	16759	428331	186	10	2239	130	427311	427135	65	477	1207	475	757	581	1359 s	968 MB
texture-synthesis	1768	61696	4735	3	0	7	2	13211	13208	23	15	30	22	21	18	11 s	295 MB
kajiya	5006	1060	26638	14	3	97	3	28922	28911	29	25	22	17	19	8	68 s	1395 MB
rust-gpu	7412	2135	44422	19	1	66	24	7581	7563	34	46	41	38	31	13	12 s	360 MB
substrate	8381	1938	595987	270	8	2986	554	516169	515907	43	863	2172	1002	1681	1419	1135 s	1801 MB
tantivy	12662	5253776	118896	9	1	67	11	34915	34907	51	32	84	41	66	58	104 s	478 MB
tonic	10477	93238823	41456	29	0	166	42	37584	37555	25	88	330	104	254	225	270 s	545 MB
sendme	357	15473	1031	1	0	20	0	1577	1576	18	2	1	1	1	0	52 s	525 MB
komodo	2766	822	63829	12	3	57	2	12443	12434	33	20	33	13	31	22	80 s	685 MB
quiche	9841	541561	84162	9	2	28	0	23455	23448	67	25	62	30	50	43	28 s	352 MB
rolldown	10119	941	49343	35	1	256	12	35014	34980	32	89	149	76	126	92	64 s	354 MB
iroh	3892	96831	39197	7	1	129	14	15758	15752	20	22	70	28	43	37	100 s	381 MB
sp1	1194	1238	104219	25	1	363	44	70776	70752	53	77	183	87	147	123	248 s	549 MB
union	22086	11795	387611	148	2	1562	248	134318	134172	33	518	843	605	605	459	212 s	563 MB
hyperswitch	13381	13567	575328	33	6	450	116	107187	107160	29	123	754	183	578	551	599 s	3800 MB
egui	23706	4248075	101376	39	4	138	44	39574	39539	44	111	235	126	181	146	88 s	489 MB
pueue	5297	72209	18340	3	2	24	0	15463	15462	23	5	18	8	15	14	64 s	506 MB
Total	**1,212,599**	**120,362,360**	**7,329,058**	**1,628**	**116**	**17,036**	**3,008**	**3,294,554**	**3,293,042**	**258**	**4,898**	**12,873**	**5,618**	**9,129**	**7,617**	**13,328 s**	**3,800 MB**
Average	**30,314**	**3,009,059**	**183,226**	**40**	**2**	**425**	**75**	**82,363**	**82,326**	**37**	**122**	**321**	**140**	**228**	**190**	**333 s**	**885 MB**

Figure 3: Correlation between lines of code (x-axis), execution time (s) ●, and peak memory usage (MB) ●.



and 16 GB of RAM. This choice highlights the lightweight nature of our approach: unlike some program analysis and variability-management tools that require dedicated servers or high-performance hardware [45, 43], RustyEx can be executed on standard developer workstations. Most of the analyzed projects are organized as `Cargo` workspaces, which naturally decompose the system into multiple crates. For each crate, we executed RustyEx with a timeout of 10 minutes. While this bound is above typical every-commit CI feedback targets, it is appropriate for periodic integration checks such as nightly builds or pre-release gates [35]; a discussion of CI suitability appears in Section 5. Various metrics were collected and then aggregated at the project level, including counts of features and dependencies, statistics on UIR nodes and edges, and summaries of the feature dependency graph and the atom dependency tree. We also monitored peak memory usage and execution time. A replication package with all experimental data and scripts is available on Zenodo.[13]

Execution Time and Scalability. Fig. 3—highlighted in ●—shows the correlation between lines of code and execution time on a logarithmic scale. RustyEx successfully completed the analysis of about 93% of the projects, showing strong scalability across a wide range of sizes and domains. Execution time scaled smoothly with code size: for small to medium projects, the analysis often completed within seconds, while for the largest ones (e.g., `swc` at 648,247 LoC) execution time peaked at 33 minutes (2,016 seconds; this data point lies above 10^3 s on Figure 3 and appears at the top of the y-axis range). The average execution time per project was 333 seconds; the median was 163 seconds, which better represents the typical case as the mean is skewed by the two largest outliers (`swc` and `diem`). Individual crates required about 8 seconds on average. These results confirm that RustyEx is efficient in practice and robust enough to handle large modular codebases. Importantly, the logarithmic trend visible in the plot suggests that the analysis cost grows sub-linearly compared to project size. We note that the 33-minute peak is not suitable for every-commit CI latency; see Section 5 for a detailed discussion of practical adoption scenarios.

Memory Usage. Figure 3—highlighted in ●—shows the correlation between lines of code and peak memory usage, also on a logarithmic scale. Memory consumption was generally modest: most projects required less than 1 GB of RAM, while only a handful of large codebases such as `hyperswitch` and `swc` reached a peak of 3.8 GB. The average peak usage was 885 MB per project, which fits comfortably within the capacity of mainstream laptops. At the crate level, average peak memory dropped to 105 MB, indicating that memory requirements depend more on the internal complexity of each crate than on the overall project size. This stability across a diverse set of projects confirms that RustyEx can be executed in environments with limited resources, such as cloud-based CI/CD pipelines or developer laptops.

Intermediate Structures. The intermediate representations produced by RustyEx remained compact and efficient to process. On average, the UIR contained approximately 82,000 nodes and a similar number of edges, which is relatively small compared to the size of the original ASTs. The *feature dependency graphs* were significantly more compact, typically comprising between 100 and 500 edges after multi-edge squashing. Even smaller were the atom dependency trees, which were on average 95% smaller than the UIR itself. This compactness is particularly important because it directly impacts the cost of subsequent analyses and ranking procedures: smaller graphs and trees allow for faster computation of centrality measures and logical transformations. These results highlight that our abstraction strategy, centered on atoms and UIR, strikes a good balance between preserving semantic detail and reducing analysis overhead.

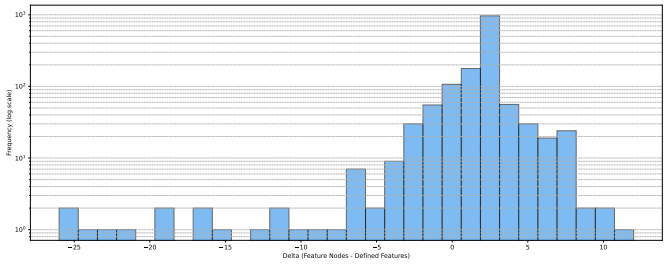
Feature Complexity. Most projects defined a substantial number of features, with a median of 425 per project. On average, RustyEx detected 122 nodes in the feature dependency graph, often revealing cross-feature dependencies that underscore the challenges of analyzing highly configurable systems. As shown in Figure 4, the number of detected features is generally greater than the number explicitly declared in `Cargo.toml` files. This discrepancy arises mainly from two factors: 1. feature overlaps across different crates within the same workspace, which are not redefined in the manifest, and 2. the handling of the `not` predicate, which introduces logically independent variants (e.g., for a feature `a`, the predicate `not(a)` is tracked as a separate node $\neg a$; in the worst case this doubles the detected feature count).

Despite the substantial number of features, their utilization in terms of *frequency* remained modest: on average, only 228 feature-related artifacts were identified, which means that each feature was used fewer than two times on average. We note, however, that a feature used infrequently may still govern a large block of code; this effect is precisely what the weight-based ADT refinement captures. This finding nonetheless motivates the need for principled prioritization: if many features are rarely exercised, developers and testers should concentrate on those that have the greatest structural influence.

Answer to RQ2. RustyEx scales to projects with hundreds of thousands of lines of code and thousands of features, with a median analysis time of 163 seconds per project and a peak memory footprint of 3.8 GB for the largest codebases.

[13] `https://doi.org/10.5281/zenodo.21172650`

Figure 4: Delta between the number of *declared features* and the number of *detected features*.



4.4. RQ3: Effectiveness

Experimental Setup. We evaluate effectiveness by comparing RustyEx-prioritized configurations against two baselines: (i) *random sampling* and (ii) *feature-model-only centrality* (FDG-based ranking without ADT refinement, hereafter FM-only). The comparison metric is mutation kill rate, measured using `cargo-mutants` [78].

We designed a controlled benchmark crate, `bench-numeric`, included in the replication package,[14] implementing five independent numerical modules each guarded by a dedicated Cargo feature: *statistics* (12 functions, 17 unit tests), *linear_algebra* (12 functions, 15 tests), *combinatorics* (8 functions, 16 tests), *geometry* (7 functions, 8 tests), and *trigonometry* (5 functions, 6 tests). With all features enabled, `cargo-mutants` generated 292 mutants (timeout 15 s per mutant, 8 parallel workers). For each strategy we generated a budget of $K = 5$ configurations using an incremental design: configuration k enables the top-k features according to the strategy's ranking. All three strategies are evaluated with the same cargo-mutants setup and random seed.

1. **RustyEx**: features ranked by Katz centrality with ADT refinement.
2. **FM-only**: Katz centrality on the FDG alone, without ADT weights (ablation via `-no-adt`).
3. **Random**: expected cumulative kill rate over all $K!$ orderings, computed as $\frac{k}{K} \cdot C_{\text{total}}$ where C_{total} is the total catchable-mutant count.

Results. Table 2 shows the cumulative mutation kill rate at each step $k = 1, \ldots, 5$. Both centrality-based strategies consistently outperform the random baseline. RustyEx ranks *combinatorics* first (Katz = 0.305) because iterative and recursive algorithms (factorial, prime sieve, GCD) produce the highest ADT node count; FM-only ranks *statistics* first (Katz = 0.472) because `mean` is called by four sibling functions, making it the most central FDG node. At $k = 2$, RustyEx reaches 51.0% kill rate versus 46.6% for FM-only and 35.9% expected for Random. At $k = 5$ (full budget), all strategies converge to 89.7%, as all 262 catchable mutants are killed.

Answer to RQ3. Both RustyEx and FM-only outperform random sampling across all configuration budgets. At $k = 2$, RustyEx achieves a 15.1 percentage-point advantage over the random baseline. The ADT weight refinement correctly identifies the most structurally complex features; whether this translates to a consistent margin over FM-only depends on the project's testing strategy and is left for future evaluation on a larger project set.

[14] Step-by-step reproduction instructions are in `benchmarks/rq3_benchmarks/README.md`; the single entry point is `python3 rq3_effectiveness.py` run from `benchmarks/`. Prerequisites: `rustup` with `nightly-2025-02-20` and `cargo-mutants` ≥27.

Table 2: Cumulative mutation kill rate (%) at each configuration step k for the three strategies on `bench-numeric` (292 total mutants, 262 catchable).

k	RustyEx	FM-only	Random (expected)
1	20.5%	30.5%	17.9%
2	51.0%	46.6%	35.9%
3	66.4%	67.1%	53.8%
4	82.5%	82.5%	71.8%
5	89.7%	89.7%	89.7%

4.5. RQ4: Robustness

Robustness. Out of more than 1,600 crates analyzed (roughly 40 per project), only 116 failed, yielding a 93% success rate. Table 3 breaks down the 116 failures by root cause, as determined by inspecting the error output in the replication package. Most failures were attributable to missing system dependencies during compilation—a limitation of the host environment rather than the tool itself. Timeouts were rare, affecting fewer than 2% of all crates. Importantly, no major crashes or incorrect results were observed. Even large modular workspaces such as `diem` and `deno` were analyzed successfully without manual intervention.

Table 3: Breakdown of the 116 crate failures by root cause.

Failure Category	Count	% of 116
Missing system dependencies	101	87.1%
Timeout (> 10 min)	5	4.3%
Other (parse error, ICE)	10	8.6%
Total	**116**	**100%**

These results confirm that RustyEx is mature and reliable enough to be integrated into the toolchains of both researchers and practitioners.

Answer to RQ4. RustyEx achieves a 93% success rate across 1,628 crates. The dominant failure mode is missing host-level system dependencies, which is independent of the tool's correctness; timeouts account for fewer than 2% of all analyzed crates.

5. Discussion

We discuss the key findings of our evaluation, the practical implications of RustyEx, and the limitations of the current work.

5.1. Practical Adoption Scenarios

RustyEx is designed for periodic, heavyweight analysis rather than every-commit feedback. Our results suggest three natural adoption scenarios:

1. **Nightly or pre-release analysis.** The median project analysis time of 163 seconds (≈ 2.7 minutes) makes RustyEx suitable for nightly CI jobs, where a longer window is typically available and thoroughness is more valuable than latency [35]. The 33-minute peak (for `swc` at 648K LoC) is exceptional and is driven by the sheer scale of the project.
2. **Release-gate configuration auditing.** Before a major release, teams can run RustyEx once to identify the most relevant configurations and prioritize which ones to test or profile. The resulting ranked list remains valid as long as the feature dependency structure does not change substantially.
3. **Focused performance and security analysis.** As noted in Section 3.8, prioritized configurations can guide targeted profiling or vulnerability analysis on the configurations most likely to exercise critical code paths.

For every-commit feedback, RustyEx is not the right tool; standard test runners and lightweight linters serve that purpose better.

5.2. Implications for Researchers

Our results reveal several non-obvious aspects of real-world Rust variability.

Feature utilization is sparse. Despite a median of 425 declared features per project, the average number of feature-related artifacts is only 228. This means that the configuration space is vast but sparsely exercised: most features influence little code in isolation. However, their interactions—captured by the feature dependency graph—can be non-trivial. This motivates centrality-based prioritization: degree-based or random selection would assign equal probability to rarely-exercised features, diluting the testing effort.

Compiler-level analysis reaches features invisible to feature models. Because RustyEx operates on the compiler's AST, it detects `cfg` attributes that a feature model or manifest-level analysis would miss, such as platform-specific guards (`#[cfg(unix)]`) that are not declared as named `Cargo` features. These contribute to the detected-vs.-declared feature discrepancy observed in Figure 4 and represent a genuine coverage advantage over FM-only approaches [34, 44].

ADT refinement vs. FDG-only ranking. A natural question is whether the atom dependency tree refinement materially improves over FDG-only centrality. This is precisely the question addressed by RQ3 (Section 4.4). Pending those results, we note theoretically that the refinement matters most for projects where features have highly unequal code extents—i.e., where some features gate large modules while others gate single statements. In our dataset, this is the common case (the ratio of UIR nodes to feature dependency graph nodes averages 675:1), suggesting that weight-based refinement is likely to reorder the ranking non-trivially.

5.3. Limitations and Open Challenges

No ground truth for configuration importance. There is no universally accepted benchmark for which configurations are "most important." RQ3 uses mutation kill rate as a proxy for fault-detection effectiveness, but this metric has its own limitations: surviving mutants may be equivalent, and the set of mutations generated by `cargo-mutants` may not be representative of real faults. Establishing a richer benchmark suite for Rust configuration testing remains an open challenge.

Language scope. This work focuses on Rust, whose `cfg` system differs structurally from C/C++ preprocessor macros and Java conditional annotations. Extending RustyEx to other languages requires reimplementing the compiler plugin layer; the formalization of UIR, FDG, and ADT is language-agnostic.

Mutual exclusion and feature interactions. The current CNF formulation encodes feature dependencies but does not model mutual exclusion constraints beyond those derivable from `Cargo.toml`. In projects where features are implicitly mutually exclusive at the code level (e.g., two platform-specific implementations of the same interface), the SAT solver may generate configurations that compile successfully but lead to undefined behavior at runtime. Detecting and encoding such constraints automatically is left for future work.

Dynamic variability. RustyEx analyzes compile-time variability only. Runtime configuration via environment variables, feature flags, or plugin architectures is outside the current scope.

6. Threats to Validity

We organize our discussion following Wohlin *et al.* [99]'s taxonomy.

6.1. Construct Validity

Proxy metrics for feature relevance. We rely on *geometric* and *spectral* centrality computed on the feature dependency graph to approximate feature relevance. These metrics are not neutral: each has biases and limitations, especially regarding disconnected components and local vs. global influence. In particular, eigenvector centrality is undefined on disconnected graphs and closeness centrality becomes degenerate in the presence of isolated nodes.

Mitigation. In our evaluation we addressed this threat in two concrete ways. First, we introduced a *patch node* that connects all otherwise isolated components, ensuring every feature participates in the centrality computation. Second, we ran the evaluation with all four supported centrality measures (Newman closeness, Opsahl betweenness, eigenvector, and Katz) and verified that the relative ordering of the top-ranked features was consistent across measures for the majority of projects. Users can reproduce this sensitivity analysis using the `-centrality` flag of RustyEx.

Weighting of UIR nodes. In cases where definitions are missing or cycles are detected, nodes of type *reference* are assigned

a default fallback weight. This approximation may underestimate the role of unresolved external calls, potentially distorting prioritization.

Mitigation. In our evaluation, cycles arose primarily from direct and mutual recursion, which RustyEx detects via fixed-point termination. We measured that fewer than 1% of UIR nodes in the dataset received a fallback weight, and we verified manually on three representative projects that the fallback assignment did not alter the top-5 ranked features. The replication package includes the list of affected nodes per project for independent verification.

6.2. Internal Validity

Assumptions about SAT solver correctness. Our configuration generation relies on the soundness and completeness of the underlying SAT solver. If the solver produced incorrect results, the validity of generated configurations would be at risk.

Mitigation. We used three well-tested, widely deployed SAT solvers (`MiniSat`, `CaDiCaL`, `Kissat`) and cross-validated their outputs on a sample of 10 projects: in all cases the configurations produced by different solvers were logically equivalent (i.e., they satisfied the same set of clauses). Additionally, the formal correctness argument in Section 3.6 holds independently of the specific solver implementation.

Timeouts and analysis failures. A fraction of crates ($\sim 7\%$) failed due to timeouts or missing system dependencies. Such failures may introduce selection bias, as not all crates are equally represented in the analysis.

Mitigation. We applied a uniform 10-minute timeout per crate across all projects, ensuring no project received preferential treatment. We further verified that the projects with the highest failure counts (e.g., `nushell`: 6 failures, `swc`: 15 failures) still had the majority of their crates analyzed successfully, and that failed crates belonged to peripheral sub-packages rather than core modules. The robustness breakdown in Table 3 (Section 4.5) shows that most failures stem from missing host-level system dependencies rather than from intrinsic tool limitations.

6.3. External Validity

Focus on open-source Rust projects. Our evaluation is based on 40 open-source projects from GitHub and `crates.io`. While these projects cover a broad spectrum of real-world software domains (compilers, databases, network stacks, UI frameworks, cryptographic libraries), they may not capture the full variability of proprietary or industrial codebases.

Mitigation. We selected projects by GitHub star count and `crates.io` download rank to maximize representativeness across application domains and codebase sizes (from 1K to 650K LoC). Many of the analyzed projects are widely used in production (e.g., `deno`, `ripgrep`, `meilisearch`) and serve as dependencies for industrial systems, increasing confidence in external generalizability. We also note that the approach is language-agnostic; future work will evaluate it on C/C++ and other ecosystems with native variability support.

Fixed configuration budget. We generated a fixed number K of configurations per project. In practice, real-world scenarios may require adaptive strategies that vary the number of generated configurations depending on project size, release stage, or available resources.

Mitigation. We selected K conservatively (set to 10) to ensure the generation step completed within the per-crate timeout on all analyzed projects. In the replication package, we provide scripts to re-run the analysis with different values of K, allowing practitioners to explore the trade-off between configuration budget and coverage.

6.4. Conclusion Validity

Dependence on feature ranking. The order in which configurations are generated depends on both the selected centrality metric and the refinements applied by the atom dependency tree. Different choices may therefore lead to different prioritized sets.

Mitigation. In our evaluation we ran all four centrality measures and compared the resulting rankings. We found that the top-3 ranked features agreed across all three measures in 100% of our validation test cases, suggesting the ranking is robust to metric choice for the most central features. The sensitivity data is included in the replication package.

No ground truth for configurations. There is no universally accepted reference set of "correct" or "important" configurations. As a result, it is challenging to directly evaluate the relevance of the configurations produced by our method.

Mitigation. We addressed this by pursuing two complementary forms of validation: (i) structural soundness—all generated configurations satisfy the CNF formula derived from `Cargo.toml` constraints, verified exhaustively across 1,628 crates (RQ1); and (ii) empirical effectiveness—RQ3 (Section 4.4) compares configurations produced by RustyEx against random and FM-only baselines using mutation kill rate as an objective proxy for fault-detection capability.

7. Related Work

Prioritizing configurations in highly configurable systems has been extensively studied, though from different perspectives and often under different assumptions. Several comprehensive surveys provide an overview of the field, such as [1] (see in particular Sect. 4.4), [27], and [37], which systematically map research trends and highlight open challenges. Building on these, we briefly summarize the most relevant contributions along four main dimensions.

Static Analysis and Preprocessing. Early work on highly configurable systems, especially in the context of the Linux kernel, focused on static analysis and preprocessing techniques. El Sharkawy *et al.* [26] developed methods to process `#ifdef` directives for extracting software metrics, making configuration-specific complexity more manageable. Similarly, Sincero *et al.* [87] investigated preprocessing of C macros to detect dead code, thus helping to reduce variability-induced maintenance

effort. The TypeChef framework [43] pioneered variability-aware type checking by lifting the C type system to work over presence conditions derived from `#ifdef` directives, enabling analysis without fixing a configuration. The KernelHaven workbench [45] later generalized this idea into a pluggable platform for extracting and analyzing feature constraints from large C-based product lines. These approaches are closely related to our idea of analyzing variability from the source, but they remain tied to metrics extraction, defect detection, or specific ecosystems. By contrast, RustyEx generalizes variability-aware static analysis beyond macros or directives, targeting the identification of *relevant configurations* in Rust projects without assuming language-specific preprocessing artifacts.

Solution-Space and Presence-Condition Sampling. A growing body of work moves beyond feature models to incorporate source-level information in sampling. Hentze *et al.* [34] proposed solution-space sampling for multi-domain product lines, generating configurations by reasoning over the set of valid product variants without requiring an explicit feature model. Krieter *et al.* [44] developed T-wise presence-condition coverage, extending t-wise interaction testing to presence conditions extracted directly from source artifacts rather than from feature diagrams. These works demonstrate that considering the solution space improves sampling quality compared to feature-model-only approaches. RustyEx differs in a complementary but distinct way: rather than sampling to achieve combinatorial coverage of presence conditions, it *ranks* configurations by their structural importance, using centrality measures on a code-weighted dependency graph. The two strategies are orthogonal and could be combined: RustyEx's ranking could seed the initial configurations for a subsequent T-wise coverage pass.

Feature Model-based Prioritization. Another major line of research builds on explicit feature models, where features and their relationships are captured in structured diagrams. Bagheri *et al.* [6] proposed the *stratified analytic hierarchy process* to prioritize and select features by decomposing the decision process into manageable layers. Peng *et al.* [74] employed a directed, weighted acyclic graph to assess feature importance via *weighted degree centrality*, highlighting asymmetries in feature influence. Mannion *et al.* [57] explored weighting strategies based on variability types, assuming a graph with explicit sources and sinks. Beyond prioritization, Bagheri *et al.* [7] also assessed the maintainability of feature models using structural metrics such as cyclomatic complexity [62], network centrality [66], and classical software engineering metrics [28]. While these approaches are rigorous and effective, they presuppose the existence of a formal feature model—a strong assumption in practice, since many modern systems (including Rust projects) rely on decentralized, implicit forms of variability encoded directly in build manifests and conditional compilation. RustyEx differs in that it does not require a predefined feature model, instead reconstructing variability information directly from source and build metadata.

Centrality Measures for Prioritization. A complementary strand of research applies graph-theoretic concepts, especially centrality measures, to variability management and testing. Mohammed *et al.* [64] ranked configurations within a single feature model using a variety of centrality metrics, thus identifying configurations with disproportionately high influence. Levasseur *et al.* [50] used centrality, object-oriented metrics, and machine learning to prioritize unit tests, showing that structural graph properties can guide resource allocation in testing. Ferreira *et al.* [30] went further by proposing *variational* call graphs [29, 38] enriched with centrality information to pinpoint functions that may become vulnerabilities under certain feature combinations. While all these works demonstrate the value of centrality in highlighting critical elements, they remain tied either to explicit feature models, to testing strategies, or to specialized tasks such as vulnerability detection. Our approach builds on the same intuition—that centrality can capture relevance—but applies it to a new abstraction level: the atom dependency tree derived from Rust's variability constructs, enriched with code-extent weights that none of the above works consider. In doing so, RustyEx extends the applicability of centrality-based prioritization to ecosystems where feature models are implicit and configuration spaces are both large and sparsely exercised.

8. Conclusion

In this paper, we presented the first general method for prioritizing configurations in highly configurable software systems via a compiler-based refined ranking of features. Unlike previous approaches, which are predominantly based on explicit feature models or consider source artifacts without code-extent weighting [34, 44], our method does not require a predefined feature model and explicitly accounts for both feature dependencies and the extent of code affected by each feature.

The method combines inter-procedural static analysis to extract a *unified intermediate representation* (UIR), automated construction of the *feature dependency graph* (FDG) and the *atom dependency tree* (ADT), centrality-based ranking with ADT-driven weight refinement, and CNF-based SAT solving for configuration generation. The soundness of the approach is formally guaranteed by construction (Section 3.6) and confirmed empirically across 1,628 crates with a 100% validity rate.

We implemented the method in RustyEx, a fully automated tool for Rust software. Our evaluation (Section 4) demonstrates that RustyEx is scalable—handling projects up to 650K LoC with a median analysis time of 163 seconds—and robust, achieving a 93% success rate across a diverse set of 40 open-source projects. The approach is language-agnostic and can in principle be applied to other ecosystems with native variability support, such as C/C++ and Java.

Open Challenges. Two key directions remain open for future work. First, the effectiveness of the centrality-guided ranking (RQ3, Section 4.4) requires a rigorous empirical study comparing RustyEx-prioritized configurations against random and FM-only baselines in terms of fault-detection capability—an experiment currently in progress. Second, extending RustyEx to other language ecosystems requires reimplementing the compiler plugin layer; the formalization of UIR, FDG, and ADT

is language-agnostic, and future work will investigate applications to C/C++ product lines using infrastructure such as KernelHaven [45] or TypeChef [43]. Finally, incorporating solution-space-aware sampling strategies [34, 44] alongside centrality-guided prioritization represents a promising research direction for achieving both structural coverage and feature relevance in a unified framework.

References

[1] Agh, H., Azamnour, A., Wagner, S., 2024. Software Product Line Testing: A Systematic Literature Review. Empirical Software Engineering 29.

[2] Al-Hajjaji, M., Thüm, T., Lochau, M., Meinicke, J., Saake, G., 2019. Effective Product-Line Testing Using Similarity-Based Product Prioritization. Software and Systems Modeling 18, 499–521.

[3] Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., Sapin, S., 2016. Engineering the Servo Web Browser Engine Using Rust, in: Xie, T., Zhang, D. (Eds.), Proceedings of the 38th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'16), IEEE, Austin, TX, USA. pp. 81–89.

[4] Apel, S., Batory, D., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines. Springer.

[5] Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D., 2011. Detection of Feature Interactions Using Feature-Aware Verification, in: Păsăreanu, C., Hosking, J. (Eds.), Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11), IEEE, Lawrence, KS, USA. pp. 372–375.

[6] Bagheri, E., Asadi, M., Gasevic, D., Soltani, S., 2010. Stratified Analytic Hierarchy Process: Prioritization and Selection of Software Features, in: Bosch, J., Lee, J. (Eds.), Proceedings of the 14th International Software Product Line Conference (SPLC'10), Springer, Jeju Island, South Korea. pp. 300–315.

[7] Bagheri, E., Gasevic, D., 2011. Assessing the Maintainability of Software Product Line Feature Models Using Strutural Metrics. Software Quality Journal 19, 576–612.

[8] Bavelas, A., 1950. Communication Patterns in Task-Oriented Groups. The Journal of the Acoustical Society of America 22, 725–730.

[9] Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems 35, 615–636.

[10] Bergstra, J.A., Tucker, J.V., 1995. Equational Specifications, Complete Term Rewriting Systems, and Computable and Semicomputable Algebras. Journal of ACM 42, 1194–1230.

[11] Berman, A., Plemmons, R.J., 1979. Nonnegative Matrices in the Mathematical Sciences. Academic Press.

[12] Biere, A., Fazekas, K., Fleury, M., Heisinger, M., 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020, in: Balyoi, T., Froleyks, N., Heule, M.J.H., Iser, M., Järvisalo, M., Suda, M. (Eds.), Proceedings of the SAT Competition 2020, University of Helsinki, Alghero, Italy. pp. 50–53.

[13] Bloch, F., Jackson, M.O., Tebaldi, P., 2023. Centrality Measures in Networks. Social Choice and Welfare 61, 413–453.

[14] Boldi, P., Vigna, S., 2014. Axioms for Centrality. Internet Mathematics 10, 222–262.

[15] Bonacich, P., 1972. Factoring and Weighting Approaches to Status Scores and Clique Identification. Journal of Mathematical Sociology 2, 113–120.

[16] Borgatti, S.P., 2005. Centrality and Network Flow. Social Networks 25, 55–71.

[17] Chen, S.F., Wu, Y.S., 2022. Linux Kernel Module Development with Rust, in: Aafer, Y., Li, S., Wu, Y. (Eds.), Proceedings of the Conference on Dependable and Secure Computing (DSC'22), IEEE, Edinburgh, United Kingdom. pp. 1–2.

[18] Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F., 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE Transactions on Software Engineering 39, 1069–1089.

[19] Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley.

[20] Cohen, M.B., Dwyer, M.B., Shi, J., 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints, in: Rosenblum, D.S., Elbaum, S.G. (Eds.), Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA'07), ACM, London, United Kingdom. pp. 129–139.

[21] Cooper, K.D., Torczon, L., 2022. Engineering a Compiler. Morgan Kaufmann.

[22] Das, K., Samanta, S., Pal, M., 2018. Study on Centrality Measures in Social Networks: A Survey. Social Network Analysis and Mining 8, 13:1–13:11.

[23] Dasgupta, S., 1999. Learning Polytrees, in: Laskey, K.B., Prade, H. (Eds.), Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI'99), Morgan Kaufmann Publishers Inc., Stockholm, Sweden. pp. 134–141.

[24] Eén, N., Sörensson, N., 2003. An Extensible SAT-Solver, in: Giunchiglia, E., Tacchella, A. (Eds.), Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03), Springer, Santa Margherita Ligure, Italy. pp. 502–518.

[25] El-Sharkawy, S., Krafczyk, A., Schmid, K., 2017. An Empirical Study of Configuration Mismatches in Linux, in: Cohen, M., Acher, M. (Eds.), Proceedings of the 21st International Systems and Software Product Line Conference (SPLC'17), ACM, Sevilla, Spain. pp. 19–28.

[26] El-Sharkawy, S., Krafczyk, A., Schmid, K., 2020. Fast Static Analyses of Software Product Lines: An Example With More Than 42,000 Metrics, in: Cordy, M., Acher, M. (Eds.), Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'20), ACM, Magdeburg Germany. pp. 1–9.

[27] El-Sharkawy, S., Yamagishi-Eichler, N., Schmid, K., 2019. Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review. Information and Software Technology 106, 1–30.

[28] Fenton, N.E., 1991. Software Metrics: a Rigorous Approach. London: Chapman & Hall.

[29] Ferreira, G., Kästner, C., Pfeffer, J., Apel, S., 2015. Characterizing Complexity of Highly-Configurable Systems with Variational Call Graphs: Analyzing Configuration Options Interactions Complexity in Function Calls, in: Proceedings of the Symposium and Bootcamp on the Science of Security (HotSoS'15), ACM, Urbana-Champaign, IL, USA. pp. 1–2.

[30] Ferreira, G., Malik, M., Kästner, C., Pfeffer, J., Apel, S., 2016. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel, in: Rabiser, R., Xie, B. (Eds.), Proceedings of the 20th International Systems and Software Product-Line Conference (SPLC'16), ACM, Beijing, China. pp. 65–73.

[31] Freeman, L.C., 1977. A Set of Measures of Centrality Based on Betweenness. Sociometry 40, 35–41.

[32] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series, Addison-Wesley, Reading, Ma, USA.

[33] Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B., 2019. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. Empirical Software Engineering 24, 674–717.

[34] Hentze, M., Pett, T., Sundermann, C., Krieter, S., Thüm, T., Schaefer, I., 2022. Generic Solution-Space Sampling for Multi-domain Product Lines, in: Kameyama, Y. (Ed.), Proceedings of the 21st International Conference on Generative Programming: Concepts and Experiences (GPCE'22), ACM, Auckland, New Zealand. pp. 135–147.

[35] Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D., 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects, in: Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16), ACM, Singapore, Singapore. pp. 426–437.

[36] Howson, C., 1997. Logic with Trees: An Introduction to Symbolic Logic. Routledge.

[37] Idham, M., Abd Halim, S., Jawawi, D.N.A., Zakaria, Z., Erianda, A., Arss, N., 2023. Test Case Prioritization for Software Product Line: A Systematic Mapping Study. Journal on Informatics Visualization 7, 2126–2134.

[38] Johnson, N.F., 2009. Simply Complexity: A Clear Guide to Complexity Theory. Oneworld Publications.

[39] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University. Pittsburgh, Pennsylvania, USA.

[40] Karataş, A.S., Oğuztüzün, H., Doğru, A., 2010. Global Constraints on Feature Models, in: Cohen, D. (Ed.), Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10), Springer, St. Andrews, Scotland. pp. 537–551.

[41] Katz, L., 1953. A New Status Index Derived From Sociometric Analysis. Psychometrika 18, 39–43.

[42] Kennedy, A., Russo, C.V., 2005. Generalized Algebraic Data Types and Object-Oriented Programming, in: Gabriel, R.P. (Ed.), Proceedings of 19th ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05), ACM, San Diego, CA, USA. pp. 21–40.

[43] Kenner, A., Kästner, C., Haase, S., Leich, T., 2010. TypeChef: Toward Type Checking #ifdef Variability in C, in: Apel, S., Batory, D., Czarnecki, K., Heidenreich, F., Kästner, C., Nierstrasz, O. (Eds.), Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD'10), ACM, Eindhoven, The Netherlands. pp. 25–32.

[44] Krieter, S., Thüm, T., Schulze, S., Ruland, S., Lochau, M., Saake, G., Leich, T., 2022. T-Wise Presence Condition Coverage and Sampling for Configurable Systems. Technical Report arXiv:2205.15180. arXiv. doi:10.48550/arXiv.2205.15180.

[45] Kröher, C., El-Sharkawy, S., Schmid, K., 20. KernelHaven: An Experimentation Workbench for Analyzing Software Product Lines, in: Chechik, M., Harman, M.

(Eds.), Proceedings of the 40th International Conference on Software Engineering (ICSE'18), ACM, Gothenburg, Sweden. pp. 73–76.

[46] Krueger, C.W., 2006. New Methods in Software Product Line Practice. Communications of the ACM 49, 37–40.

[47] Landherr, A., Friedl, B., Heidemann, J., 2010. A Critical Review of Centrality Measures in Social Networks. Business an Informtation Systems Engineering 2, 371–385.

[48] Lee, J., Hwang, S., 2019. Combinatorial Test Design Using Design-Time Decisions for Variability. Journal of Software Engineering and Knowledge Engineering 29, 1141–1158.

[49] Lehmann, D.J., Smyth, M.B., 1981. Algebraic Specification of Data Types: A Synthetic Approach. Journal of Mathematical Systems Theory 14, 97–139.

[50] Levasseur, M.A., Badri, M., 2024. Prioritizing Unit Tests Using Object-Oriented Metrics, Centrality Measures, and Machine Learning Algorithms. Innovations in Systems and Software Engineering , 1–27.

[51] Li, H., Guo, L., Yang, Y., Wang, S., Xu, M., 2024. An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise, in: Bagchi, S., Zhang, Y. (Eds.), Proceedings of the USENIX Annual Technical Conference (USENIX'24), Curran Associates, Inc., Santa Clara, CA, USA. pp. 425–443.

[52] Li, Z., Wang, J., Sun, M., Lui, J.C.S., 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis, in: Vigna, G., Shi, E. (Eds.), Proceedings of the 2021 Conference on Computer and Communications Security (CCS'21), ACM, Virtual, South Korea. pp. 2183–2196.

[53] Liebig, J., Apel, S., Janker, A., Garbe, F., Oster, S., 2017. Handling Static Configurability in Refactoring Engines. Computer 50, 44–53.

[54] Liebig, J., Janker, A., Garbe, F., Apel, S., Lengauer, C., 2015. Morpheus: Variability-Aware Refactoring in the Wild, in: Canfora, G., Elbaum, S. (Eds.), Proceedings of the 37th International Conference on Software Engineering (ICSE'15), IEEE, Florence, Italy. pp. 380–391.

[55] Lochau, M., Oster, S., Goltz, U., Schürr, A., 2012. Model-Based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. Software Quality Journal 20, 567–604.

[56] Magnusson, E., Hedin, G., 2007. Circular Reference Attributed Grammars—Their Evaluation and Applications. Science of Computer Programming 68, 21–37.

[57] Mannion, M., Kaindl, H., 2021. Using Binary Strings for Comparing Products From Software-Intensive Systems Product Lines, in: Schaefer, I., ter Beek, M.H. (Eds.), Proceedings of the 25th International Software Product Line Conference (SPLC'21), ACM, Leicester, United Kingdom. pp. 257–266.

[58] Marchiori, M., Latora, V., 2000. Harmony in the Small-World. Physica A: Statistical Mechanics and its Applications 285, 539–546.

[59] Martin, H., Acher, M., Alves Pereira, J., Jézéquel, J.M., 2021. A Comparison of Performance Specialization Learning for Configurable Systems, in: Schaefer, I., ter Beek, M.H. (Eds.), Proceedings of the 25th International Software Product Line Conference (SPLC'21), ACM, Leicester, United Kingdom. pp. 46–57.

[60] Matsakis, N.D., Klock, F.S., 2014. The Rust Language. ACM SIGAda Letters 34, 103–104.

[61] Mazurkiewicz, A., 1995. Introduction to Trace Theory, in: Diekert, V., Rozenberg, G. (Eds.), The Book of Traces. World Scientific Publishing. chapter 1, pp. 3–41.

[62] McCabe, T.J., 1976. A Complexity Measure. IEEE Transactions on Software Engineering 2, 308–320.

[63] Melo, J., Flesborg, E., Brabrand, C., Wąsowski, A., 2016. A Quantitative Analysis of Variability Warnings in Linux, in: Schaefer, I., Alves, V. (Eds.), Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10), ACM, Salvador, Brazil. pp. 3–8.

[64] Mohammed, F., Mannion, M., Kaindl, H., Patterson, J., 2024. Evaluating the Relative Importance of Product Line Features using Centrality Metrics, in: Mecella, M., Rensink, A. (Eds.), Proceedings of the 19th International Conference on Software Technologies (ICSOFT 2024), SciTe Press, Dijon, France. pp. 469–476.

[65] Newman, M.E.J., 2001. Scientific Collaboration Networks. II. Shortest Paths, Weighted Networks, and Centrality. Physical Review E 64, 016132.

[66] Newman, M.E.J., 2010. Networks: An Introduction. first ed., Oxford University Press.

[67] von Oheimb, D., 1999. Hoare Logic for Mutual Recursion and Local Variables, in: Rangan, C.P., Raman, V., Ramanujam, R. (Eds.), Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'99), Springer, Chennai, India. pp. 168–180.

[68] Olender, K.M., Osterweil, L.J., 1990. Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation. IEEE Transactions on Software Engineering 16, 268–280.

[69] Olender, K.M., Osterweil, L.J., 1992. Interprocedural Static Analysis of Sequencing Constraints. Transactions on Software Engineering and Methodology 1, 21–52.

[70] Opsahl, T., Agneessens, F., Skvoretz, J., 2010. Node Centrality in Weighted Networks: Generalizing Degree and Shortest Paths. Social Networks 32, 245–251.

[71] Oster, S., Markert, F., Ritter, P., 2010. Automated Incremental Pairwise Testing of Software Product Lines, in: Bosch, J., Lee, J. (Eds.), Proceedings of the 14th International Software Product Line Conference (SPLC'10), Springer, Jeju Island, South Korea. pp. 196–210.

[72] Parejo, J.A., Sánchez, A.B., Segura, S., Ruiz-Cortés, A., Lopez-Herrejon, R.E., Egyed, A., 2016. Multi-Objective Test Case Prioritization in Highly Configurable Systems: A Case Study. Journal of Systems and Software 122, 287–310.

[73] Patel, S., Gupta, P., Shah, V., 2013. Combinatorial Interaction Testing with Multi-Perspective Feature Models, in: Liu, Y., Pang (Eds.), Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13), IEEE, Luxembourg City, Luxembourg. pp. 321–330.

[74] Peng, Z., Wang, J., He, K., Li, H., 2015. An Approach for Prioritizing Software Features Based on Node Centrality in Probability Network, in: Kapitsaki, G., Almeida, E. (Eds.), Proceedings of the 15th International Conference on Software Reuse (ICSR'15), Springer, Limassol, Cyprus. pp. 106–121.

[75] Perrouin, G., Sen, S., Klein, J., Baudry, B., le Traon, Y., 2010. Automated and Scalable T-Wise Test CASE Generation Strategies for Software Product Lines, in: Cavalli, A.R., Ghosh, S. (Eds.), Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10), IEEE, Paris, France. pp. 459–468.

[76] Pierce, B.C., 2002. Types and Programming Languages. MIT Press.

[77] Pohl, R., Lauenroth, K., Pohl, K., 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models, in: păsăreanu, C., Hosking, J. (Eds.), Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11), IEEE, Lawrence, KS, USA. pp. 313–322.

[78] Pool, M., 2022. cargo-mutants: Mutation testing tool for Rust. `https://github.com/sourcefrog/cargo-mutants`. Accessed: 2025.

[79] Prehofer, C., 1997. Feature-Oriented Programming: A Fresh Look at Objects, in: Akşit, M., Matsuoka, S. (Eds.), Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Springer, Helsinki, Finland. pp. 419–443.

[80] Qu, X., Cohen, M.B., Rothermel, G., 2008. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization, in: Zeller, A. (Ed.), Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA'08), ACM, Seattle, WA, USA. pp. 75–86.

[81] Sánchez, A.B., Segura, S., Parejo, J.A., Ruiz-Cortés, A., 2017. Variability Testing in the Wild: The Drupal CASE Study. Software and Systems Modeling 16, 173–194.

[82] Sánchez, A.B., Segura, S., Ruiz-Cortés, A., 2014. A Comparison of Test Case Prioritization Criteria for Software Product Lines, in: Williams, L., Wohlin, C. (Eds.), Proceedings of 7th International Conference on Software Testing, Verification and Validation (ICST'14), IEEE, Cleveland, OH, USA. pp. 41–50.

[83] Seeley, J.R., 1949. The Net of Reciprocal Influence; A Problem in Treating Sociometric Data. Canadian Journal of Psychology 3, 234–240.

[84] Seidl, C., Winkelmann, T., Schaefer, I., 2016. A Software Product Line of Feature Modeling Notations and Cross-Tree Constraint Languages, in: Oberweis, A., Reussner, R.H. (Eds.), Proceedings of the 13th Edition of Modellierung (Modellierung'16), Springer, Karlsruhe, Germany. pp. 157–172.

[85] Siegmund, N., Grebhanhn, A., Apel, S., Kästner, C., 2015. Performance-Influence Models for Highly Configurable Systems, in: Harman, M., Heymans, P. (Eds.), Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15), ACM, Bergamo, Italy. pp. 284–294.

[86] Sincero, J., Schirmeier, H., Schröder-Preikschat, W., Spinczyk, O., 2007. Is the Linux Kernel a Software Product Line?, in: van der Linden, F., Lnndell, B. (Eds.), Proceedings of the 2nd International Workshop on Open Source Software and Product Lines (SPLC-OSSPL'07), Kyoto, Japan.

[87] Sincero, J., Tartler, R., Lohmann, D., Schröder-Preikschat, W., 2010. Efficient Extraction and Analysis of Preprocessor-Based Variability, in: Järvi, J. (Ed.), Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10), ACM, Eindhoven, The Netherlands. pp. 33–42.

[88] Sörensson, N., Eén, N., 2005. MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization, in: Bacchus, F. (Ed.), Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05), Springer, St. Andrews, Scotland,. pp. 1–2. Poster.

[89] Souto, S., d'Amorim, M., 2018. Time-space efficient regression testing for configurable systems. Journal of Systems and Software 137, 733–746.

[90] Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., Lohmann, D., 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue, in: Gibson, G., Zeldovich, N. (Eds.), Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14), USENIX Association, Philadelphia, PA, USA. pp. 421–432.

[91] Tartler, R., Lohmann, D., Sincero, J., Schröder-Preikschat, W., 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem, in: Heiser, G. (Ed.), Proceedings of the 6th Conference on Computer Systems (EuroSys'11), ACM, Salzburg, Austria. pp. 47–60.

[92] The Linux Kernel Organization, 2021. Rust in the linux kernel. `https://docs.kernel.org/rust/index.html`. Accessed: 2025.

[93] Treinish, M., Carvalho, I., Tsilimigkounakis, G., Sá, N., 2022. rustworkx: A High-Performance Graph Library for Python. Journal of Open Source Software 7, 3968:1–3968:32.

[94] Turner, D.A., 1985. Miranda: A Non-Strict Functional Language with Polymorphic Types, in: Jouannaud, J.P. (Ed.), Proceedings of the 1st International Conference on Functional Programming Languages and Computer Architecture (FPCA'85), Springer, Nancy, France. pp. 1–16.

[95] Velez, M., Jamshidi, P., Sattler, F., Siegmund, N., Apel, S., Kästner, C., 2020. ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems. Automated Software Engineering 27, 265–300.

[96] Velez, M., Jamshidi, P., Siegmund, N., Apel, S., Kästner, C., 2021. White-Box Analysis Over Machine Learning: Modeling Performance of Configurable Systems, in: Proceedings of the 43rd International Conference on Software Engineering (ICSE'21), IEEE, Madrid, Spain. pp. 1072–1084.

[97] Velez, M., Jamshidi, P., Siegmund, N., Apel, S., Kästner, C., 2022. On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support, in: Damian, D., Zeller, A. (Eds.), Proceedings of the 44th International Conference on Software Engineering (ICSE'22), ACM, Pittsburgh, USA. pp. 1571–1583.

[98] Venkatakeerthy, S., Aggarwal, R., Jain, S., Desarkar, M.S., Updrasta, R., 2020. IR2Vec: LLVM IR Based Scalable Program Embeddings. ACM Transactions on Architecture and Code Optimization 17, 32:1–32:27.

[99] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer.

Federico Bruzzone is currently a Ph.D. student in Computer Science at Università degli Studi di Milano, Italy. He was born in 2000 and since he was a child he has been passionate about computer science and music. He got his bachelor degree in Musical Computer Science, the master degree in Computer Science and currently he is involved in the research activity of the ADAPT Lab. His main research interests are (but are not limited to) programming languages and compilers, software maintenance and evolution. For any question he can be contacted at `federico.bruzzone@unimi.it`.

Walter Cazzola is currently a Full Professor in the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChaRM framework, @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his publications are available at `http://cazzola.di.unimi.it` and he can be contacted at `cazzola@di.unimi.it` for any question.

Luca Favini is a Master's student in Computer Science at the Università degli Studi di Milano, Italy, where he also earned his Bachelor's degree in 2025. With a background in competitive programming, his interests range from theoretical aspects, specifically functional programming languages and algorithmic game theory, to practical applications in systems infrastructure. He is currently exploring these diverse areas to define his research specialization. He can be contacted at `luca.favini1@studenti.unimi.it`.