Feature-Focused Test Generation

José Antonio Zamudio Amaya
jose.zamudio@cispa.de
CISPA Helmholtz Center for Information Security
Germany

Gaetano Sapia
gaetano.sapia@mpi-sp.org
CISPA Helmholtz Center for Information Security and MPI-SP
Germany

Alexi Turcotte
alexi.turcotte@cispa.de
CISPA Helmholtz Center for Information Security
Germany

David Benavides
benavides@us.es
Instituto de Ingeniería Informática, Universidad de Sevilla
Spain

Marcel Böhme
marcel.boehme@mpi-sp.org
CISPA Helmholtz Center for Information Security and MPI-SP
Germany

Andreas Zeller
andreas.zeller@cispa.de
CISPA Helmholtz Center for Information Security
Germany

Abstract

Specification-based test generators use input specifications to produce valid tests for complex systems, covering the entire input space of the system under test. This broad generation capability, however, can become a burden when one wants to focus test generation on a particular subset of the input space, say, a feature whose implementation has recently changed or is particularly critical.

In this paper, we propose *Feature-Focused Test Generation* (FFTG), an approach that treats specification-based test generation as a *configuration problem*. From the input specification, FFTG first extracts a *feature model* that represents alternate and optional expansions as possible feature values. After the tester selects which features to focus upon, FFTG specializes the specification accordingly, generating inputs in which only the focused features vary.

We implement FFTG in a prototype named Compás. In our evaluation, we find that Compás-generated slices (1) effectively *focus test generation* on specific features, preserving validity; (2) exercise unique code locations associated with the targeted features and thus enable *traceability* of features; and (3) enable *test suite minimization,* with a small set of slices retaining the aggregate coverage of the full suite while requiring fewer tests.

To the best of our knowledge, this is the first work to apply product line engineering techniques to specification-based testing.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Software product lines**.

Keywords

Language-Based Testing, Software Product Lines, FANDANGO

ACM Reference Format:
José Antonio Zamudio Amaya, Gaetano Sapia, Alexi Turcotte, David Benavides, Marcel Böhme, and Andreas Zeller. 2026. Feature-Focused Test Generation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837479

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837479

1 Introduction

Specification-based testing derives test inputs from an explicit specification of the system under test rather than from manually written test cases, giving test generation a systematic and repeatable basis [46, 53]. Language-based testing [52] applies this principle to the input side: the specification is a description of the input language, which is used to produce large numbers of valid, richly structured inputs for systems that process complex formats. Compared to unconstrained random generation, the key advantage is that the specification already captures much of the domain knowledge needed for effective testing: not only the syntactic structure of the input, but also optional constructs, mutually exclusive choices, and semantic constraints over combinations of choices. In modern language-based generators such as FANDANGO [61], a specification combines (1) *grammar rules* that describe the input structure; and (2) *semantic constraints* that restrict input values. The FANDANGO specification excerpt representing the PNG format in Figure 1, for example, uses production rules to expose semantic choices such as ⟨*srgb_chunk*⟩ and ⟨*iccp_chunk*⟩; the final constraint forbids the incompatible combination `srgb_chunk=perceptual` and `iccp_chunk=eci_rgb_v2`.

Consider a developer who has just added animation support to a PNG library. To trust the change, they should be able to stress-test the animation feature and its interactions with the rest of the format, such as interlacing, color profiles, and transparency. A language-based generator is of limited help: although it can be told to focus on animation by imposing extra constraints (at a throughput cost), it cannot say which of its outputs exercise the new handler or what other code they touch, and cannot tell which inputs in a long regression run are redundant. What the developer needs are three capabilities a plain specification does not provide, and which run as the guiding example throughout this paper: to *target* a chosen feature, to *trace* which code each feature exercises, and to *minimize* a campaign to the slices that matter. In this paper, we *make this variability explicit and configurable,* treating the specification as a model of *input features* rather than just a source of valid inputs.

This is the perspective of *software product line (SPL) engineering* [1, 15], which models variability as features and constraints and separates the *problem space*, the feature model of what may vary, from the *solution space*, the mechanisms that realize a chosen configuration [19]. In this testing scenario, once the input feature space is explicit, generation can be directed at specific features and

```
<start> ::= <sig> <control_chunks> <image_data> <iend>
<sig> ::= "\x89PNG\r\n\x1a\n"

<control_chunks> ::= <srgb_chunk> <iccp_chunk>

<srgb_chunk> ::= <srgb_none> | <perceptual> | <relative>
<iccp_chunk> ::= <iccp_none> | <display_p3> | <eci_rgb_v2>

<perceptual> ::= "sRGB\x00\x00"
<relative> ::= "sRGB\x00\x01"
<display_p3> ::= "iCCP:DisplayP3"
<eci_rgb_v2> ::= "iCCP:ECIRGBv2"

where not (<srgb_chunk> == <perceptual> and
           <iccp_chunk> == <eci_rgb_v2>)
```

Figure 1: FANDANGO PNG language specification (excerpt). The `where` keyword the introduces semantic constraints.

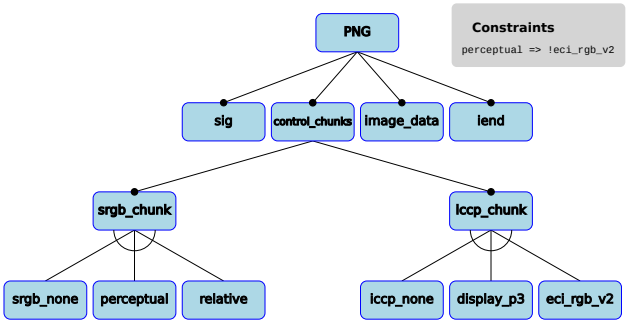

Figure 2: Feature model corresponding to the PNG specification in Figure 1. Both ⟨*srgb_chunk*⟩ and ⟨*iccp_chunk*⟩ become ALTERNATIVE groups and the `where` constraint becomes a cross-tree exclusion between `perceptual` and `eci_rgb_v2`.

their executions compared, giving the tester a black-box handle on which regions of code each semantic choice exercises.

Figure 2 represents the variability implicit in Figure 1 as an explicit feature model. This representation can be leveraged to make the specification configurable: instead of asking the generator for "another valid PNG", we can ask for a PNG with `srgb_chunk = perceptual` or `iccp_chunk = display_p3`.

Making the specification's variability explicit provides a handle to turn "re-test the whole format" into "re-test the affected feature," [22], where an undifferentiated grammar campaign offers no handle for it: a corpus without feature labels cannot say which inputs exercise the changed feature.

Furthermore, in modern practice, fuzzing complex parsers is served directly into CI/CD [13, 37, 49]. Under this regime the testing budget itself is the binding constraint: finding linearly more bugs in a fixed time has been shown to require exponentially more machines [9], so every cycle spent re-exercising unchanged parts of a format is a cycle not spent finding new defects.

This paper develops all of these ideas into *Feature-Focused Test Generation* (FFTG): a way to turn input specifications into configurable variability models and then use those models to derive feature-specific slices. FFTG spans both: the *featurizer* automatically produces a feature model from the specification, while the *configurator* and *pruner* prune the specification into a slice. Each slice encodes the selected feature values and semantics, and can be used to generate inputs that share common characteristics. By constructing one slice per feature value, we obtain a *slice suite* where each campaign targets exactly one feature value, enabling structured coverage analysis and test suite minimization. The resulting workflow (Figure 3) adds *structure* to language-based testing: tests can be organized by feature values, coverage can be related back to targeted input features, and redundant campaigns can be minimized systematically.

To the best of our knowledge, this is the first time that techniques from feature-oriented software development are brought into specification-based testing. Product-line techniques have long been used to test configurable systems; our contribution is to apply that perspective to the variability of input specifications themselves.

We implement FFTG in COMPÁS[1], an open-source prototype that operates on FANDANGO specifications. Our contributions are:

Feature-focused test generation. We bridge specification-based testing and feature-oriented software development by treating specification alternatives as configurable features. FFTG automatically extracts a feature model from an input specification and derives one slice per feature value, allowing test generation to focus directly on specific feature values while preserving validity and reusing the original specification.

Feature-to-code traceability. By comparing code coverage between slices for alternative values of the same feature, FFTG relates observed code behavior to specification-level input features. This exposes associated code locations without requiring a manual decomposition of the implementation.

Feature-based test suite minimization. The structured coverage enables greedy set-cover minimization [3, 59], identifying a small subset of slices that preserves the aggregate coverage of the full slice suite while requiring fewer tests. This turns slice-level coverage information into a practical mechanism for reducing campaigns.

We evaluate FFTG on ten real specifications across seven parser libraries: four image format parsers from Pillow (GIF, JPEG, PNG, TIFF), an audio parser from Mutagen (MP3), and archive, document, binary, configuration, and markup parsers (`zipfile`, `pypdf`, `pyelftools`, `tomllib`, `html.parser`). Across the ten subjects, the evaluation shows that the resulting slices (1) focus test generation on specific feature values while preserving the validity of all targeted inputs; (2) feature slices trigger exclusive code paths in every format, exposing sibling-distinct behavior that enables feature traceability; and (3) suites shrink to 1–2 slices while preserving the aggregate coverage of the full slice suites. Feature-Focused Test Generation thus not only enables input generation to focus on specific features but also promises advances in feature traceability and test suite reduction.

[1] A play on the Spanish *compás* (the rhythm that guides a Fandango) and the English *compass* (for navigation and targeting).

2 Background

This paper bridges specification-based testing and product line engineering. We unify these domains by lifting input specifications into feature models, enabling explicit, configurable test generation over complex input spaces.

2.1 Specification-based Testing

Testing programs that process complex inputs, such as image formats, network protocols, or structured documents, requires inputs that satisfy complex structural and semantic rules. Specification-based testing [46, 53] meets this need by deriving inputs from a formal description of the expected input rather than crafting them by hand or mutating raw bytes.

The most common structured-input specification is the context-free grammar [14]: production rules describe the valid structure of inputs and expose choices through *alternatives*, *optional* elements, and *repetitions*. Its defining property is *syntactic validity by construction*, as inputs derived from a grammar satisfy its syntactic rules and therefore pass initial parsing to exercise deeper program behavior [33]. A grammar alone, however, cannot express the *semantic* constraints that real formats impose, such as checksums, length fields, or dependencies between fields.

Language-based testing [52] closes this gap by pairing a grammar with semantic constraints, yielding a *language specification* that captures both syntax and semantics. The structural correspondence between specificaions and feature models is well established: de Jonge and Visser [21] first cast feature diagrams as grammars, and Batory [4] and Benavides et al. [7] related feature models to grammars and propositional logic. Each alternative in a grammar represents a choice among distinct options (e.g., color modes, compression methods); each optional element a presence or absence decision (e.g., metadata chunks); and each constraint a restriction on valid combinations.

Together, these constructs describe a space of possible input variations. Still, a generator uses that space only to draw valid samples, leaving the variability itself implicit rather than exposing it as a model that a tester could target or analyze. The premise of feature-focused test generation is that different values of the same feature often exercise different code in the program under test, which makes them meaningful units for targeted testing. We build on FANDANGO specifications [61], where inputs are valid by construction with respect to both syntax and semantics, so they pass a program's input-validation stage and reach the logic that specific feature combinations control. Our work lifts FANDANGO specifications into feature relations, preserving the format's semantic validity, and exploits it to make input variability explicit and configurable for test generation (Figure 2).

2.2 Feature Models and Software Product Lines

A software product line (SPL) is a family of related systems that share a common platform but differ in their supported *features* [1]. The construction of such families is performed by modeling variability explicitly rather than embedding it in code or build scripts, casting this separation as two spaces: the *problem space* of features that may vary, and the *solution space* of mechanisms that realize a chosen configuration [19].

The most common representation of this variability is the *feature model*, introduced by Kang et al. [32], represented in Figure 2. A feature model is a rooted tree whose nodes represent features and whose edges encode variability relations. A feature may be *mandatory* (always present when its parent is selected), *optional* (may or may not be present), or part of an *alternative* group (exactly one child selected) or an *or* group (at least one child selected) [5, 48]. In the PNG feature model of Figure 2, for instance, ⟨*srgb_chunk*⟩ and ⟨*iccp_chunk*⟩ are alternative groups, each selecting exactly one color-management value. Beyond the tree hierarchy, *cross-tree constraints* express dependencies between features in different subtrees, typically as *requires* (selecting A forces selection of B) or *excludes* (selecting A forbids B) relations [7]. The PNG model carries one such relation: it *excludes* the combination `perceptual` and `eci_rgb_v2`, the semantic constraint of Figure 1 lifted from the grammar's `where` clause into the model.

A valid *configuration* is a selection of features that satisfies both the tree structure and all cross-tree constraints; choosing `srgb_chunk = perceptual` together with `iccp_chunk = display_p3`, for instance, is one valid PNG configuration. The set of satisfiable configurations defines the configuration space of the product line, which is typically exponential in the number of features [54]. Input specifications are no different: the full PNG specification we study (Table 1) already admits 9,216 valid configurations. This makes exhaustive enumeration infeasible and motivates sampling strategies for testing. The explicit representation of variability also enables *automated analysis*, such as checking satisfiability, detecting dead features, and validating configurations [5, 7].

3 Feature-Focused Test Generation

We present *Feature-Focused Test Generation* (FFTG), an approach for treating specification-based testing as a configuration problem. The key concept is to view an input specification as a software product line, derive a feature model from the specification, and use configurations of that model to produce *slices* that target specific features. Figure 3 illustrates the FFTG pipeline.

3.1 Fandango Specifications

FFTG builds on FANDANGO [61], the state-of-the-art language-based test generator. A FANDANGO *specification* pairs grammar rules with semantic constraints. Constraint support is essential here, since feature-value exclusions and computed fields cannot be expressed by a plain grammar. We define such a specification as $L = (G, C)$, where G is the underlying context-free grammar and C is a set of Python predicates over nonterminals that enforce semantic validity.

Production rules may use *alternatives* ($\alpha_1 \mid \alpha_2$), *repetition* (A^*, A^+, $A^?$), and *generator functions* ($A \to \beta \ := \ f(\ldots)$), where f is a Python function that computes the terminal yield from the values of other nonterminals. Figure 1 shows an excerpt of the PNG specification. The excerpt highlights two nonterminals: ⟨*srgb_chunk*⟩ and ⟨*iccp_chunk*⟩. The excerpt also shows a concrete value-level constraint:

```
not (<srgb_chunk> == <perceptual> and
     <iccp_chunk> == <eci_rgb_v2>)
```
(1)

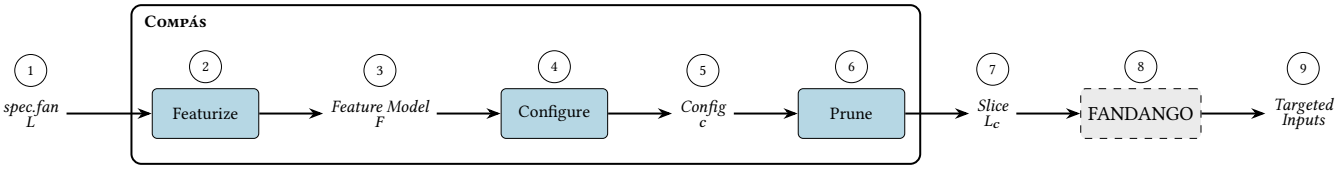

Figure 3: Overview of Feature-Focused Test Generation. From a FANDANGO specification L ①, FFTG *featurize* ② extracts a feature model F ③; *configure* ④ produces a configuration c ⑤; and *prune* ⑥ combines the configuration and other logic into a specification slice L_c ⑦. The slice is passed to FANDANGO ⑧ to generate feature-targeting inputs ⑨.

When the specification is set to `<srgb_chunk> = <perceptual>`, the conflicting `<iccp_chunk> = <eci_rgb_v2>` alternative is removed, yielding a smaller specification (*slice*, as we named it) that still generates only valid PNG files. The specification thus defines a *configuration space*: every slice corresponds to a specific combination of features from the specification. Once the desired features are selected, the resulting specification becomes a slice of the original.

3.2 Featurization

The first phase extracts a *feature model* $F = (T, C)$ from the grammar G, where T is a feature tree and C is a set of cross-tree constraints.

Definition 3.1 (Feature model). A feature model $F = (T, C)$ consists of a rooted tree T whose nodes are *features*, and a set C of cross-tree constraints. Each non-leaf node has a *relation type*: Mandatory (all children must be selected), Optional (child may or may not be selected), or Alternative (exactly one child must be selected). A *configuration* $c \subseteq T$ is a set of selected features satisfying the tree structure and all constraints in C [5].

The featurization proceeds by traversing the grammar from the start symbol S and mapping grammar constructs to feature tree nodes, producing a feature model such as the one shown in Figure 2. Following the grammar-to-feature-model correspondence established by Batory [4] and Benavides et al. [7], concatenations become hierarchical parent-child relations and alternatives become Alternative groups; we extend this mapping to FANDANGO's `where` clauses, which become cross-tree constraints. As a summary:

(1) **Alternatives** ($A \to \alpha_1 \mid \ldots \mid \alpha_n$): If $n \geq 2$ and each α_i is a terminal literal, A becomes an Alternative feature group with n child features—one per literal value. These are the primary source of configurable features.
(2) **Concatenations** ($A \to B_1\, B_2\, \ldots\, B_m$): A becomes a Mandatory feature whose children are the features derived from $B_1, \ldots, B_m$.
(3) **Optional constructs** ($A \to B^?$): B becomes an Optional child of A.
(4) **Trivial nonterminals**: Alternatives whose children are all single-character terminals (e.g., $\langle letter \rangle \to$ `"a" | "b" | ...`) are collapsed to leaf features. A configurable threshold (default: 30) separates character classes from format-semantic alternatives.
(5) **Generator nonterminals**: Rules using `:=` (generator functions) are preserved as features named after the nonterminal. If the generator expression references other nonterminals, those referenced selectors are lifted as children in the feature model; otherwise, the generator nonterminal remains a leaf. For example, a length field computed as $\langle size \rangle \to$ `:= encode(`$\langle mode \rangle$`)` from a color-mode selector $\langle mode \rangle$ makes $\langle size \rangle$ a feature and lifts $\langle mode \rangle$ as its child, keeping the computed field tied to the feature value it depends on; a generator that references no selector, such as a fixed magic number, stays a leaf.

A nonterminal's type is set by the rule that references it as a child, not by its own production, so the cases compose: if A appears as an alternative in one rule and is itself a concatenation $A \to B_1 \ldots B_m$, then A is an Alternative-group member of its parent while $B_1, \ldots, B_m$ become its mandatory children.

FANDANGO specifications may include `where` clauses expressing semantic constraints as Python predicates. These clauses are lifted into the feature model as value-level Excludes relations, such as the PNG rule (1), which states that the `perceptual` value of `srgb_chunk` cannot co-occur with the `eci_rgb_v2` value of `iccp_chunk`.

In the PNG example of Figure 2, the root `PNG` mirrors the simplified $\langle start \rangle$ rule, with four children corresponding to `sig`, `control_chunks`, `image_data`, and `iend`. The variability is concentrated under `control_chunks`, where the nonterminals $\langle srgb\_chunk \rangle$ and $\langle iccp\_chunk \rangle$ become two Alternative groups. Their children are the concrete values shown in Figure 1, such as `srgb_none`, `perceptual`, `relative`, `iccp_none`, `display_p3`, and `eci_rgb_v2`. The constraint in Figure 1 is reflected in Figure 2 as a cross-tree exclusion between `perceptual` and `eci_rgb_v2`.

3.3 Configuration

A *full configuration* c assigns each feature in T a status: *selected* or *deselected*. For Alternative groups, exactly one child must be selected. Cross-tree constraints restrict which complete configurations are valid: if $A{=}x$ excludes $B{=}y$ and c selects $A{=}x$, then c must deselect $B{=}y$.

Compás also supports *partial configurations* that leave some features undecided. Undecided features are treated as unconstrained. For the evaluation, we generate *per-value* partial configurations: one partial configuration per feature value. We call the alternative values in the same feature group *siblings*; for example, in the PNG group $\langle srgb\_chunk \rangle$ = {`srgb_none`, `perceptual`, `relative`}, the values `perceptual` and `relative` are siblings. Given a feature group $F = \{v_1, \ldots, v_n\}$, the partial configuration c_{v_i} selects v_i and deselects all siblings v_j ($j \neq i$). All other groups are left unconstrained, so the resulting slice can freely sample across those dimensions.

Definition 3.2 (Per-value partial configuration). For a feature value v_i in group F, the per-value partial configuration c_{v_i} is:

$$c_{v_i}(x) = \begin{cases} \text{selected} & \text{if } x = v_i \\ \text{deselected} & \text{if } x \in F \setminus \{v_i\} \\ \text{undecided} & \text{otherwise} \end{cases}$$

Here $F \setminus \{v_i\}$ denotes the *siblings* of v_i, the other values of the same group, and the *otherwise* case covers every feature outside F, which is left undecided and free to vary.

After constructing c_{v_i}, constraints are *propagated*: if selecting v_i excludes some value w in another group, w is automatically deselected. In PNG, for example, the partial configuration selecting `perceptual` automatically deselects the conflicting `eci_rgb_v2` value via the extracted constraint. Also, the JPEG configuration selecting the *grayscale* color mode automatically deselects the *sub_422* and *sub_420* chroma-subsampling values, since a grayscale image has no chroma to subsample, via the extracted Excludes constraints; JPEG's feature model follows the same structure as Figure 2 and is omitted for space.

The total number of per-value partial configurations equals k, the total number of feature values across all groups. This is linear in k, unlike the exponential number of full configurations ($\prod_i |F_i|$).

3.4 Pruning

Given a configuration c, the *pruner* produces a slice L_c by specializing the specification so that it generates only inputs consistent with c. At the grammar level, this corresponds to a reduced grammar $G_c \subseteq G$. The pruning operates on the specification in three passes:

(1) **Collect deselected nonterminals.** Walk the feature model: any feature whose grammar nonterminal is deselected by c, plus all its descendants, forms the set D of nonterminals to remove.
(2) **Rewrite production rules.** For each rule $A \to \alpha$:
   - If $A \in D$: drop the entire rule.
   - Otherwise: remove from α any reference to a nonterminal in D. For alternative rules, remove deselected alternatives. If the rewritten RHS is empty, drop the rule.
(3) **Filter constraints.** Drop any `where` clause that references a nonterminal in D.

Consider the configuration $c_{\text{perceptual}}$ that selects ⟨*srgb_chunk*⟩=`perceptual`. The pruner removes the alternatives `srgb_none` and `relative` from the ⟨*srgb_chunk*⟩ rule, producing:

```
<srgb_chunk> ::= <perceptual>
```

All other rules in the simplified excerpt remain unchanged except for the propagated constraint, which removes the conflicting `eci_rgb_v2` value from ⟨*iccp_chunk*⟩. The resulting slice $L_{c_{\text{perceptual}}}$ generates valid PNG files that always use the perceptual `sRGB` chunk while freely varying across the remaining semantic choices. In feature-model terminology, this pruning is a *specialization* [55]: the slice L_c admits a strict subset of the configurations of the original model, so the edit is guaranteed to only remove features, never introduce new ones.

3.5 Slicing

Given a configuration c, we define the corresponding *slice* L_c as the specialized specification obtained by pruning the original specification L with respect to c. Its generated language is restricted to inputs consistent with the selections fixed by c, while all undecided feature groups remain free to vary.

Definition 3.3 (Slice). Let L be a specification and let c be a partial or full configuration over its extracted feature model. The *slice* L_c is the specification derived from L such that every input generated by L_c satisfies the selections and deselections imposed by c.

For the evaluation, we use *per-value slicing*: we derive one slice per feature value by iterating over all k feature values:

(1) For each value v_i, construct configuration c_{v_i} (with constraint propagation).
(2) Prune the specification to obtain slice $L_{c_{v_i}}$.
(3) Generate n inputs from $L_{c_{v_i}}$ using FANDANGO.

This produces k *slices*, each containing n inputs that exercise a specific feature value while varying all other features. The total number of inputs is $k \times n$. For PNG, where the extracted model has $k = 29$ feature values, this yields 29 slices, such as one that always generates the perceptual `sRGB` chunk and another that always generates the `ECIRGBv2 iCCP` chunk.

3.6 Analysis

Feature diversity measures whether the feature values exposed by the extracted model are actually exercised at least once. This is distinct from structural grammar-coverage metrics [28]: they count derivation elements and their combinations in the grammar; feature diversity measures coverage of the extracted variability space.

Feature diversity. Formally, let V be the set of feature values extracted from the grammar, with $|V| = k$, and let T be a generated corpus. For each input $t \in T$, let $obs(t) \subseteq V$ be the set of feature values realized in t. We define *feature diversity* (Div) as

$$Div(T) = \frac{|\bigcup_{t \in T} obs(t)|}{|V|}.$$

A richer specification yields a richer extracted feature model, which in turn enables more precise slices, higher feature diversity, and finer-grained feature-to-code analysis. Conversely, if the specification omits important format options or interactions, the resulting feature model and all downstream analyses are necessarily coarser.

The two are therefore complementary rather than competing: Compás can first restrict generation to a semantically relevant subspace, after which a generator can systematically explore the remaining specification structure within that subspace.

3.7 Benefits

The extracted feature model and the resulting slices enable three downstream analyses: feature diversity, coverage analysis, and test suite minimization, and their precision depends directly on the completeness of the specification.

Coverage analysis. Because every input in a slice is generated with one feature value pinned, the code that slice exercises can be

attributed to that value. We can execute each slice's inputs against the program under test and record the set of lines each slice covers, together with per-input execution traces.

Test suite minimization. Running one slice per feature value is thorough but redundant: many slices reach overlapping code, so much of a full slice suite adds no new coverage. Minimization asks for the smallest set of slices that still reaches everything the full suite does. Unlike classical test-case minimization, every retained slice keeps its feature label, so the covering subset says not just *how many* campaigns suffice but *which features* they target, turning minimization into a statement about which parts of the format still need testing.

Computational cost. The featurizer performs a single-pass traversal of the grammar AST, collecting alternative groups and their values. The pruner rewrites the specification at the source-text level using regular expressions, avoiding full AST reconstruction.

4 Evaluation

We evaluate FFTG, as implemented in Compás, on ten subjects spanning image, audio, archive, document, executable, and configuration and markup formats, parsed by seven independent libraries, addressing four research questions:

RQ1. Does featurization preserve the validity, diversity, and aggregate coverage of the original specification while exposing configurable feature structure? We analyze whether FFTG adds semantic structure without sacrificing the practical effectiveness of the original specification.

RQ2. How effective is feature-focused generation at steering tests toward a chosen feature-specific region? We compare how quickly and how consistently a feature-focused slice reaches its corresponding feature-specific region, relative to the baseline specification.

RQ3. Do specification-level features correspond to distinct code coverage? We evaluate whether the extracted feature values actually correspond to meaningful differences in code coverage of the program under test.

RQ4. How effective is featurization for test suite minimization? We analyze whether the structure exposed by FFTG can be exploited to remove redundant slices while retaining the coverage achieved by the full slice suite.

Subjects. We selected real-world formats spanning image, audio, archive, document, executable, configuration, and markup domains and a broad range of specification variability, from richly structured formats with up to ten feature groups and thousands of valid configurations (PNG) to nearly flat ones with only two groups (MP3, ELF). The programs at test are: image decoders in Pillow [38] (GIF, JPEG, PNG, TIFF), the MP3 parser in Mutagen [39], the standard-library `zipfile` (ZIP), the document parser `pypdf` (PDF), `pyelftools` (ELF), and the standard-library `tomllib` (TOML) and `html.parser` (HTML). For each subject, we write a FANDANGO specification. FFTG automatically extracts feature groups, their feature values, cross-tree EXCLUDES constraints, computed-field constraints, and the number of valid configurations. Table 1 summarizes the targets and the extracted variability.

Table 1: Evaluation subjects and extracted variability. 'Groups' = feature groups, 'Values' = feature values, 'Excl.' = extracted cross-tree excludes, 'Comp.' = computed-field constraints (CRC, sizes, offsets), and 'Cfg.' = valid configurations.

Subj.	Parser	Groups	Values	Excl.	Comp.	Cfg.
ELF	`pyelftools`	2	11	0	1	28
GIF	`GifImage`	7	18	0	0	576
HTML	`html.parser`	3	13	0	0	48
JPEG	`JpegImage`	5	15	3	0	216
MP3	`mutagen.mp3`	2	9	0	2	20
PDF	`pypdf`	3	13	1	0	80
PNG	`PngImage`	10	29	5	2	9,216
TIFF	`TiffImage`	4	17	8	15	240
TOML	`tomllib`	3	15	0	0	60
ZIP	`zipfile`	4	8	0	13	32

Environment. Every campaign runs inside a pinned Docker image that fixes Python 3.14 and the exact parser-library versions on which the measured coverage depends. Generation is fully seeded: FANDANGO's `--random-seed` together with a fixed `PYTHONHASHSEED` makes each campaign reproduce bit-for-bit. The host was a MacBook Pro M2 Max with 64 GB of RAM running macOS 26. The replication package, including the image and all specifications, is provided in the tool repository. A full run of the evaluation takes approximately 48 hours.

Modus Operandi. For each subject, Compás automatically produces k slices, one per feature value, each pinning that value and leaving the remaining features free. Each slice is handed to FANDANGO with a per-slice budget n, the number of inputs generated from that slice; the budget controls how thoroughly each feature value is exercised and, together with k, fixes the size of the campaign at $k \times n$ inputs. To set n, we first ran a calibration campaign sweeping $n \in \{1, 5, 50, 100\}$ across all subjects, with ten repetitions each. At small budgets, the campaign is unstable due to sampling noise. The measurements converge by n=50 and do not change at n=100, so we fix n=50 as the smallest budget that yields stable results, giving up to $k \times 50$ targeted inputs per subject. Larger budgets add nothing: because each feature value is pinned as an explicit alternative, inputs beyond this point vary only payload data within an already fixed structure rather than reaching new feature combinations, so the grammars do not need more than n=50. For a fair comparison, the FANDANGO baseline receives the same input budget: from the original, unsliced specification it generates, by ordinary random generation, exactly the number of inputs the targeted suite realizes (equal input count). The full campaign is repeated ten times; we report the mean across repetitions, with standard deviation where it is non-zero. We measure:

(1) **Validity**: the fraction of generated inputs that the target parser accepts;
(2) **Feature diversity** (Div.): the fraction of extracted feature values observed in the generated corpus;
(3) **Python line coverage** (PyCov) via `coverage.py`, scoped to the parser module(s), together with the **aggregate covered number of lines** (Cov); **behavioral distinctness** (Arc), the mean pairwise arc distance between per-input execution

traces; and **generation throughput** (Tput), the sustained rate of valid inputs produced (in inputs per second);

(4) **Annotated-region reach**: the number of manually annotated feature-handling code regions triggered;
(5) **Focusing effectiveness**: for each feature value, whether the targeted slice reaches its intended region (Reach), the median inputs to the first hit (First), and the fraction of inputs that hit the region (Rate), each relative to the baseline (Table 3);
(6) **Sibling-exclusive coverage**: for each feature group, whether a slice triggers lines that no sibling triggers, with mean pairwise Jaccard distance (Dist) as a complementary separation measure (Table 4);
(7) **Traceability and minimization**: whether each annotated slice reaches its own region (Hit), the mean recall on that region (Own), whether its recall is strictly higher than every sibling's (Top) and no sibling reaches the region at all (Uniq), together with the greedy labelled covering subset and the resulting time-to-coverage speedup (Table 5).

4.1 RQ1: Preservation

Featurization must not degrade the testing properties of the original specification. Table 2 confirms that this preservation holds across all ten subjects. Every generated input is valid (100%), every feature value is observed in the targeted corpus (100% feature diversity), and aggregate line coverage is essentially identical between the baseline and Compás campaigns. The one exception is instructive: for ZIP, free baseline generation realizes only 62% of the feature values, whereas targeting recovers all of them and covers five additional lines. Under per-value slicing, 100% feature diversity is guaranteed by construction and serves as a sanity check on end-to-end validity preservation.

Generation throughput (Table 2, columns Tput F/C) shows that slicing preserves the generator's sustained emission rate: targeted and baseline produce distinct inputs at comparable speed. Feature-focused generation is at or above the baseline rate for PNG, ZIP, and TOML and within measurement noise on GIF, MP3, PDF, ELF, and HTML; JPEG is the only subject where the smaller sliced grammar generates clearly slower, and TIFF's rate is too noisy to compare. Each slice pins one feature value and removes the corresponding alternatives from the grammar, producing a smaller specification with fewer derivation choices, so FANDANGO's evolutionary search rejects fewer invalid candidates per generation cycle.

> Compás automatically featurizes all ten specifications and produces slices that preserve input validity, feature diversity, and aggregate coverage, confirming that featurization is a preserving transformation.

4.2 RQ2: Focusing

Although RQ1 established that aggregate coverage is the same for both campaigns, the two approaches differ fundamentally in how they reach feature-specific code. Baseline generation covers feature-specific regions *incidentally*, as a side-effect of random exploration across the full specification. Feature-focused generation covers them *deliberately and reliably*. This distinction is critical when a tester needs to exercise a specific feature, e.g., to validate a recent change to the animation handler or to reproduce a bug that manifests only under progressive JPEG encoding.

Table 2: Baseline (FANDANGO, F) and targeted (Compás, C) campaign summary across the ten subjects.

Subj.	Valid	Div F	Div C	PyCov	Cov	Arc F	Arc C	Tput F	Tput C
ELF	100%	100%	100%	33.1%	697	8.6 ± 0.2	8.4 ± 0.1	1007.6 ± 2.7	1001.2 ± 1.8
GIF	100%	100%	100%	39.9%	226	27.5 ± 0.1	27.5 ± 0.1	11.0 ± 0.4	11.3 ± 0.2
HTML	100%	100%	100%	30.9%	153	16.2 ± 0.3	16.2 ± 0.1	236.0 ± 11.1	228.5 ± 10.2
JPEG	100%	100%	100%	34.6%	136	20.3 ± 0.2	20.4 ± 0.1	5.5 ± 0.3	4.3 ± 0.5
MP3	100%	100%	100%	43.8%	416	4.7	4.7	4.5	4.5
PDF	100%	100%	100%	21.7%	656	4.4	4.4	1341.4 ± 2.5	1336.0 ± 2.7
PNG	100%	100%	100%	41.8%	333	37.3 ± 0.1	37.3 ± 0.1	17.6 ± 0.3	18.5 ± 0.5
TIFF	100%	100%	100%	37.9%	310	9.3 ± 1.2	10.6 ± 0.6	7.0 ± 8.2	7.1 ± 8.1
TOML	100%	100%	100%	43.2%	239	32.9 ± 0.2	33.0 ± 0.1	212.4 ± 6.7	225.2 ± 5.5
ZIP	100%	62%	100%	29.3%†	309†	3.2	3.7	130.7 ± 0.1	133.9 ± 0.5

†PyCov and Cov coincide for F and C on every subject except ZIP, where the baseline (F) covers 29.0%, 304 lines.

Table 3: Focusing effectiveness on the manually annotated intended regions.

Subj.	Vals.	Reach T	Reach B	First T	First B	Rate T	Rate B
ELF	4	100%	100%	1	6.5 ± 2.4	100%	14%
GIF	3	100%	100%	1	2.0 ± 1.1	100%	50 ± 1%
HTML	3	100%	100%	1	4.8 ± 3.8	100%	27 ± 1%
JPEG	3	100%	100%	1	3.0 ± 0.9	100%	25%
MP3	5	100%	100%	1	3.0	100%	20%
PDF	4	100%	100%	1	3.5 ± 0.9	100%	20%
PNG	10	100%	100%	1	2.2 ± 0.5	100%	32%
TIFF	2	100%	100%	1	18.5 ± 12.2	100%	11 ± 3%
TOML	4	100%	100%	1	3.1 ± 2.0	100%	17 ± 1%
ZIP	3	100%	67%	1	11.4 ± 0.5	100%	33%
Total	**41**	**100%**	**98%**	**1**	**4**	**100%**	**26%**

We evaluate focusing against *manually annotated intended regions*. For each feature value we can localize, we read the parser source and annotate the *intended region*: the set of lines that implement the handling of that value, for example the branch that dispatches on a given compression method or the block that parses a particular metadata chunk. These regions are derived from the implementation, not from any generated corpus, so they are an unbiased ground truth, independent of the slices whose focusing we measure. To the best of our knowledge, no feature-to-code mapping exists for these parsers independent of coverage analysis, which is precisely why we use manual annotation, fixed before any experiment, as the ground truth. For each annotated value, we then compare the slice targeting it against the baseline on that value's intended region. Table 3 summarizes the results.

Across the 41 annotated feature values, targeted slices reach their intended region at least once within their 50-input budget in every case (Reach T 100%). The baseline reaches most regions eventually, but misses ZIP's third region entirely (67% reach), and elsewhere arrives later and far less consistently. The median first hit is 1 generated input for the targeted slice, compared with 4 for the baseline. This aggregate median understates the gap on the harder subjects: the baseline needs a median of 18.5 inputs to first reach the hardest TIFF region, 11.4 for ZIP, 6.5 for ELF, and 4.8 for HTML, whereas the targeted slice reaches its region almost immediately (median 1 input for every subject).

On average, 100% of a targeted slice's inputs exercise the intended feature-specific region, versus 26% for the baseline. ELF, TOML, and PDF show the sharpest contrast (100% targeted versus 14%, 17%, and 20% baseline, respectively), while PNG, despite having the largest

Table 4: Sibling-level slice analysis. Subject totals (values with exclusive lines / total values) appear in parentheses.

Subj.	Feat.	#	Dist	Excl.
ELF (6/11)	ShVar	7	9.4%	symtab(+50), note(+45), dynamic(+36), rela(+29), nobits(+2)
	EType	4	0.1%	core(+1)
GIF (7/18)	Animation	2	18.1%	animated(+41)
	Transparency	2	15.0%	transparent(+26), opaque(+8)
	Disposal	3	10.4%	background(+19), previous(+11)
	PaletteScope	2	5.8%	global(+8), local(+5)
HTML (5/13)	Construct	8	8.1%	comment(+9), entity(+9), pi(+9), charref(+8), self-close(+1)
JPEG (9/15)	Metadata	4	11.2%	exif(+14), icc(+7), comment(+6)
	ColorMode	3	10.9%	cmyk(+9), grayscale(+1), rgb(+1)
	Encoding	2	7.4%	baseline(+9), progressive(+1)
	Subsampling	3	0.5%	sub_444(+1)
MP3 (6/9)	Id3v1Tail	5	2.3%	genre(+16), comment(+2), title(+1), artist(+1), album(+1)
	V2Frame	4	0.7%	comment(+5)
PDF (5/13)	Obj4	5	5.4%	runlength(+21), flate(+14), asciihex(+10), ascii85(+8), raw(+1)
PNG (18/29)	Actl	2	33.9%	apng(+109), static(+4)
	Text	4	8.0%	itext(+22), ztext(+15), text(+11)
	Srgb	2	7.5%	no(+20), yes(+5)
	Iccp	2	7.5%	no(+14), yes(+11)
	Chrm	2	6.3%	no(+16), yes(+5)
	Gama	2	6.0%	no(+16), yes(+4)
	Phys	2	2.7%	yes(+9)
	Exif	2	1.5%	yes(+4), no(+1)
	Interlace	2	0.6%	adam7(+2)
	ColorDepthMode	9	0.5%	indexed_8(+7)
TIFF (9/17)	Compression	5	10.3%	raw(+25), jpeg(+2)
	ColorMode	6	7.0%	gray(+2), y444(+2)
	Metadata	4	7.0%	xmp(+4), exif(+2), icc(+1)
	Orientation	2	2.8%	top_left(+7), rot90(+1)
TOML (11/15)	Value	10	17.1%	array(+23), inline(+22), datetime(+6), literal_str(+7), date(+1), bool(+4), basic_str(+2), float(+1)
	Key	3	6.5%	dotted(+17), quoted(+3)
	Comment	2	2.5%	present(+6)
ZIP (4/8)	LfhMethod	2	5.3%	deflated(+13), stored(+3)
	ExtraRecord	2	1.6%	3_records(+5)
	ArchiveComment	2	0.3%	absent(+1)

configuration space (9,216 valid configurations), improves from 32% to 100%. This means that nearly every input from a focused slice is *useful* for the intended testing goal, whereas roughly three-quarters of baseline inputs exercise unrelated code.

> Feature-focused generation provides reliable, on-demand access to feature-specific code that baseline generation reaches only by chance. Compás reaches chosen feature-specific regions with a mean hit rate of 100% versus 26% for baseline.

4.3 RQ3: Code coverage

Compás produces a *map* from specification features to the code each slice exercises; here we ask how informative the map is.

For each group, we compute the set of lines covered by each value's slice and check whether any are *sibling-exclusive*, i.e., triggered by that slice but by no other value in the same group.

Table 4 summarizes the results. Across the ten subjects, 80 of 148 feature values trigger code that their siblings do not, spanning formats where nearly every feature is distinct and formats where only a few are. This information is invisible to baseline generation: without per-feature slicing, one cannot determine that `apng` exercises 109 lines that `static` does not, or that TIFF's `raw` compression triggers 25 exclusive lines. Such mappings support regression testing (e.g., re-running only the animation slice after a change to the animation handler) and debugging (e.g., narrowing a crash to lines exclusive to a single compression method).

Table 5: Per-subject traceability and greedy covering summaries, with each slice annotated by its marginal gain ('+new') and cumulative coverage percentage.

Subj.	Hit	Own	Top	Uniq	Covering slice set
ELF	4/4	100%	4/4	4/4	EType=core (+947, 100%); EType=rel (+1, 100%); 2/11 sel.
GIF	3/3	100%	1/3	1/3	Animation=animated (+280, 100%); 1/18 sel.
HTML	3/3	87%	2/3	2/3	Doctype=none (+180, 100%); 1/13 sel.
JPEG	3/3	100%	3/3	3/3	Quality=low (+162, 100%); 1/15 sel.
MP3	5/5	100%	5/5	5/5	V2Frame=comment (+514, 99%); V2Frame=title (+3, 100%); 2/9 sel.
PDF	4/4	100%	4/4	4/4	Version=1.4 (+788, 100%); 1/13 sel.
PNG	10/10	74%	10/10	10/10	Phys=yes (+392, 100%); 1/29 sel.
TIFF	2/2	100%	2/2	2/2	ColorMode=gray (+363, 98%); ColorMode=y444 (+7, 100%); 2/17 sel.
TOML	4/4	92%	3/4	3/4	Comment=present (+294, 99%); Key=quoted (+2, 100%); 2/15 sel.
ZIP	3/3	100%	3/3	3/3	ExtraRecord=3_records (+364, 100%); ArchiveComment=present (+1, 100%); 2/8 sel.
Total	41/41	95%	37/41	37/41	—

Sibling-exclusive lines show that distinct code exists; we next ask whether it is *attributable* to the intended feature. Reusing the manually annotated regions of Section 4.2, but now comparing each slice against its siblings rather than against the baseline, we measure separation: whether a slice reaches its own region (Hit), with strictly higher recall than every sibling (Top), and with no sibling reaching it at all (Uniq). Table 5 reports these per-subject traceability metrics alongside the greedy reduction results discussed in Section 4.4. Every annotated slice reaches its own region, and seven of the ten subjects separate cleanly, with each annotated region reached only by its own slice.

However, several feature values cannot be attributed to any Python line: some decoders run in native code (TIFF `lzw`/`packbits`/`deflate`, JPEG entropy, GIF LZW), and some values denote the *absence* of an optional record. Whether a specification feature is visible at line granularity depends on how the parser is written, not on the feature itself. For JPEG, the Quality group shows no sibling-exclusive lines, consistent with quantization handling living in `libjpeg`; Compás still exposes differential behavior in ColorMode, Metadata, and Encoding.

> Feature slices provide a structured map from specification features to the code each exercises, which baseline generation cannot produce. Feature-to-code distinctness holds wherever the parser branches on the feature.

4.4 RQ4: Minimization

Different slices cover different lines, and we can find a subset of slices whose union covers all lines reached by all slices combined.

Table 5 shows the covering results. ZIP is the most informative case: its two-slice cover spans two feature groups (ExtraRecord

and ArchiveComment), showing that independent dimensions of the format contribute complementary coverage. TIFF and MP3 similarly need two slices, two ColorMode values and two V2Frame values respectively, whereas most subjects can be reduced to a single slice. When a single slice suffices, this is itself a diagnostic finding: it reveals that the remaining feature values contribute no unique code coverage, indicating weak feature-to-code alignment for those subjects. This is implementation-specific, since an implementation can just handle a subset of the features a format defines.

The key insight is that minimization and differentiation measure different things, at different scopes. RQ3 compares slices *within* a feature group; RQ4 compares *across* all groups globally. Thus, a slice can be *sibling-distinct* yet *globally redundant*. A globally redundant slice remains valuable for traceability (RQ3); minimization tells the tester which slices are *additionally* necessary for coverage.

Beyond the reduction in suite size, the key advantage over classical test case minimization is *explainability*: the retained slices carry feature labels that directly inform fuzzing campaign design. For TIFF, the covering set {ColorMode=gray, ColorMode=y444} tells the tester exactly which format dimensions matter for coverage. In a CI/CD pipeline, instead of launching a single expensive campaign against the full TIFF specification, the tester runs two focused campaigns, each operating on a smaller, pruned grammar. Each campaign explores fewer alternatives and converges faster, and they can run in parallel on separate workers. In a regression check after a parser change, this means reaching the same coverage in a fraction of the time, using a suite whose inputs are individually traceable to specific features.

> Featurization enables principled, explainable test suite minimization, reducing the covering suite to 1–2 labeled slices per subject and reaching each region in up to 18.5× fewer inputs than the baseline.

4.5 Discussion

The four research questions build a cumulative argument:

(1) **Correctness (RQ1):** Featurization is a safe transformation that preserves validity, diversity, and aggregate coverage, establishing the baseline against which the structural benefits are measured.
(2) **Control (RQ2):** Feature slices provide reliable, on-demand access to feature-specific code regions, replacing the incidental coverage of baseline generation with deliberate targeting.
(3) **Traceability (RQ3):** Slices produce a structured map from specification features to code regions, enabling testers to understand which feature exercises which parser logic. Baseline generation cannot provide this information.
(4) **Efficiency (RQ4):** The traceability structure enables principled minimization, reducing the test suite to 1–2 slices per subject while reaching each region in up to 18.5× fewer inputs than the baseline.

Aggregate coverage is essentially the same for baseline and Compás (Table 2), ZIP aside, where targeting recovers under-sampled feature values. This is by design: the contribution is not more coverage, but *coverage with structure*.

A baseline specification and Compás's slices both reach the same code; the difference is that Compás tells the tester *why* each input exists, *which* feature it targets, and *what* code it is expected to exercise. This structure is what enables the demonstrated benefits: targeted testing of features, feature-to-code traceability for regression testing and debugging, and principled test suite minimization.

FFTG gives the tester a semantic control layer for grammar-based testing: coverage differences become attributable to explicit feature values and comparable against sibling alternatives, helping decide which campaigns to run, skip, or prioritize.

4.6 Threats to Validity

Construct validity. We use Python line coverage as the primary measure of code-level differentiation between feature slices. Line coverage is a coarse proxy: two slices may execute the same lines with different values, triggering different behavior without any visible difference in coverage. Our arc-distance metric partially mitigates this by comparing execution traces at the branch level, but subtler semantic differences (e.g., different quantization tables in JPEG) remain invisible at both levels. The sibling-exclusive line metric counts lines unique to a slice within its feature group; a line may appear "exclusive" because it correlates with the pinned feature, not because it is causally triggered by it.

Internal validity. Each slice generates n=50 inputs, a budget selected after pilot runs with $n \in \{1, 5, 50, 100\}$. Moving from n=5 to n=50 did not change aggregate coverage, traceability, or the size of the minimized slice sets because the semantically relevant choices are fixed as explicit alternatives rather than discovered stochastically. We authored each FANDANGO specification ourselves, from published format documentation, so the extracted feature groups reflect the capabilities we chose to model, such as color modes and compression methods. A differently scoped grammar could expose a different feature model and, in turn, different slices and traceability results. We mitigate this by following the documented structure of each format, and by reporting subjects where the feature-to-code alignment is weak alongside those where it is strong.

External validity. We evaluate on ten subjects across seven parser libraries: four image formats (GIF, JPEG, PNG, TIFF) in Pillow, an audio format (MP3) in Mutagen, and archive, document, binary, configuration, and markup formats (ZIP, PDF, ELF, TOML, HTML) via `zipfile`, `pypdf`, `pyelftools`, `tomllib`, and `html.parser`. The image formats are well-suited to feature-oriented analysis because they have clearly delineated optional capabilities (e.g., metadata chunks, color modes), while MP3, ZIP, PDF, ELF, TOML, and HTML add non-image cases spanning archive, document, binary, configuration, and markup parsing. Broader domains, or parsers with less feature-aligned code, may show weaker slice differentiation.

5 Limitations

Specification effort. The featurization is automatic, but FFTG can only expose variability already modeled in the input specification. More complete specifications yield more informative featurization and slice-level analyses. This is already visible in our results: GIF compresses to a single Python-level covering slice despite having seven sibling-distinct slices, MP3 exposes only two feature groups

and correspondingly shallow global differentiation, and JPEG's `Quality` group does not separate at the Python level even though other JPEG groups do. These cases do not invalidate FFTG; rather, they show that its explanatory power scales with the semantic resolution of the specification. Recent work on *mining* [8] and *learning* [35] specifications could reduce this burden further.

Code alignment. FFTG assumes that feature-level distinctions in the specification correspond to distinct code paths in the implementation under test. When this alignment holds, slices produce measurably different execution footprints and the minimized suite retains meaningful code coverage diversity. When it does not, slice differentiation will be weak and the approach offers little advantage over undirected generation. Our results already illustrate this spectrum: some feature groups (e.g., JPEG `Quality`) show only marginal Python-level differentiation, likely because the logic resides in a native C library rather than in the instrumented Python code.

Minimization and test quality. Our minimization retains the slices needed to preserve *aggregate coverage*, but coverage is only a proxy for fault-detection power. Two suites that reach the same lines or branches need not be equally effective at exposing defects: a dropped slice may exercise a covered line with an input that triggers a bug the retained slices do not, for instance a boundary value or an unusual feature combination that happens to hit the same code. The covering subset is therefore best read as a way to remove coverage-redundant campaigns from a regression budget, not as a claim that the discarded slices are worthless; each still carries a feature label and can be re-run when its feature is under scrutiny. This coverage-adequacy gap is shared by all coverage-based minimization; empirically, minimized suites often retain most of the fault-detection power of the full suite [58].

Feature interactions. Currently, Compás tests one feature value at a time: each slice pins a single feature value and leaves the remaining features unconstrained. This means that bugs triggered only by specific *combinations* of feature values may not be systematically reached. While the unconstrained features are still sampled during generation, this sampling is random rather than deliberate, so interaction coverage depends on chance rather than on a principled strategy. Extending Compás with pairwise or t-wise combinatorial testing [12] over the extracted feature model is a natural next step.

6 Related Work

Automatic test generation. Prior work spans model-based generation from use cases [40], feedback-directed random generation [41], search-based testing with whole-test-suite optimization [23, 36], and recent LLM-based approaches to test and oracle generation [30, 47, 50]. These approaches synthesize executable test *code* for APIs or units. FFTG instead synthesizes structured *input data* from format specifications, keeping generation black-box while enabling feature-level targeting, comparison, and minimization.

Language-based testing. Grammar-based test generation dates back to compiler-testing work by Burkhardt [11] and Hanford [27]; Purdom [45] formalized production-covering sentence generation, Lämmel [33] framed grammar-based testing systematically, and Godefroid et al. [25] combined grammars with symbolic execution in grammar-based whitebox fuzzing. Havrikov and Zeller [28] introduced k-path coverage, a structural criterion that systematically covers combinations of grammar elements and their interactions, achieving significantly higher code coverage than random grammar-based generation. Steinhöfel and Zeller [51] developed ISLa, a specification language that adds semantic constraints over grammar elements, enabling the generation of inputs that satisfy both syntactic and context-sensitive properties. Zamudio Amaya et al. [61] introduced FANDANGO, which replaces symbolic constraint solving with evolutionary search, trading exhaustiveness for the ability to express constraints as arbitrary Python predicates and for greater scalability. Our work builds on this line of research: we take the specifications used by language-based generators and extract from them a feature model that structures the testing campaign around semantically meaningful input variability.

Coverage-guided fuzzing. Coverage-guided greybox fuzzing, pioneered by AFL [60] and formalized as a Markov chain by Böhme et al. [10], mutates seed inputs and retains those that increase code coverage. While highly effective for unstructured inputs, coverage-guided fuzzers struggle with highly structured formats because most mutations produce invalid inputs that are rejected early. Structure aware fuzzers address this limitation. Aschermann et al. [2] proposed Nautilus, which combines grammar-based generation with coverage-guided feedback, using grammars to ensure syntactic validity while letting coverage guide exploration. Pham et al. [44] introduced AFLSmart, which leverages input format models to perform smart mutations that respect the chunk structure of complex file formats. Padhye et al. [42] presented Zest, which converts random generators into parametric generators. Holler et al. [29] proposed LangFuzz, which recombines code fragments from a grammar to fuzz language interpreters. These approaches use grammars or format models as a means to improve fuzzing throughput; they do not expose the specification's variability as a configurable feature model. FFTG is complementary: the slices it produces could serve as feature-targeted seeds for any of these fuzzers, enabling them to explore in a more structured way.

Product line testing and sampling. Software product line (SPL) engineering [1, 15] manages families of software products through systematic variability modeling. Testing SPLs is challenging because the number of valid product configurations grows combinatorially with the number of features. Cohen et al. [18] proposed combinatorial interaction testing (CIT) for highly configurable systems in the presence of constraints, generating test configurations that cover all t-way feature interactions while respecting inter-feature dependencies. Perrouin et al. [43] and Johansen et al. [31] developed automated and scalable t-wise test case generation strategies for SPLs, using SAT solvers to handle large feature models with constraints. Varshosaz et al. [57] provided a comprehensive classification of product sampling strategies for SPLs, distinguishing among coverage-based, dissimilarity-based, and random sampling approaches. These ideas have been used to test configurable software product lines. Our work brings them instead to *input* variability: FFTG extracts a feature model from a specification and derives slices analogous to product configurations.

Variability analysis. Feature models, introduced in the FODA method [32], are the standard formalism for representing variability in SPLs. Prior work has established their formal semantics [20, 48], automated analysis [5], standardized textual notation [6], tool support [24, 56], and use for predicting non-functional properties [26]. Our featurization procedure aligns with these formalizations: alternatives become alternative groups, optional constructs become optional features, and cross-tree constraints are lifted into Excludes relations. The key difference is that our models are extracted automatically from input specifications and describe variability of *inputs*, not software products. Concretely, Compás serializes each extracted feature model to UVL [6], so its output interoperates directly with existing feature-model analysis and sampling tools; the UVL captures exactly the variability we reason about, the alternative groups and lifted Excludes relations, while the remaining Python `where` predicates stay in the FANDANGO specification and are enforced at generation time.

7 Conclusions and Future Work

In this paper, we have introduced *Feature-Focused Test Generation* (FFTG), an approach that bridges specification-based testing and software product line engineering. To the best of our knowledge, this is the first work to take techniques from product line engineering and apply them to specification-based testing over input variability. By automatically extracting a feature model from an input specification, FFTG enables generating inputs that target specific format features while enabling traceability to the code they exercise. Our evaluation on ten real parser specifications across seven parser libraries shows that: (1) the featurization is fully automatic and produces slices that preserve validity and testing effectiveness; (2) the feature slices trigger code paths that their siblings do not, confirming that specification-level features induce distinct execution footprints; and (3) all annotated slices reach their own intended regions, while greedy minimization can substantially reduce the retained covering suite. This makes FFTG a practical approach for structuring testing campaigns over real specifications around semantically meaningful input features, opening new directions in targeted testing, feature traceability, and test suite minimization. Our future work will focus on the following topics:

Pairwise and combinatorial testing. Generating slices for feature *pairs* (or higher-order combinations) would extend combinatorial techniques [17, 43] from software product line testing to input specifications. Since enumerating all valid full configurations is infeasible, it raises an important question that we do not study here: whether interaction-focused pruning preserves, or even sharpens, the feature interactions [16] that matter in the specification.

Targeted fuzzing campaigns. The slices produced by Compás are not limited to single-input generation. A long-running fuzzing campaign restricted to a single slice could deeply explore one feature in isolation, potentially finding bugs that shallow whole-specification fuzzing misses. Combining per-value slices with coverage-guided fuzzers could yield campaigns that are both *targeted* and *adaptive*.

Feature-to-code traceability. The mapping from grammar features to code regions opens applications beyond test minimization. A developer modifying a specific parser handler could automatically identify which feature slices to re-run, enabling *regression test selection* [22] guided by the feature model. Conversely, given a bug report in a specific code region, the feature model could identify which input features reach that code, guiding targeted reproduction.

Feature diversity. Our results suggest that *feature diversity* is a promising adequacy criterion for specification-based testing. In particular, it offers a way to ask whether semantically meaningful input variants have been exercised. Formalizing feature diversity as a test adequacy criterion and studying its relationship to fault detection is a promising direction.

Protocols and interaction grammars. Recent advances extend language-based testing from file formats to communication protocols, modeling them as *interaction grammars* that tag each message element with the party responsible for producing it and fold states, messages, and transitions into a single grammar [34]. Like format specifications, these interaction grammars expose alternative, optional, and mutually exclusive structure, such as optional extensions, negotiated capabilities, and message or record types. Featurizing an interaction grammar would then let a tester target a single protocol capability, for instance a particular SMTP command or DNS record type, trace how the implementation handles it, and minimize protocol test suites. Protocol design is thus another domain where feature modeling and feature-aware testing apply, making the extension of Compás to interaction grammars a natural next step.

Acknowledgments

We are grateful to the reviewers for their insightful comments. Bridging distinct fields means navigating different standards and methodologies, which makes reaching consensus on an interdisciplinary paper somewhat of a challenge (to say the least). We sincerely appreciate the reviewers' engagement with this project. We also extend our gratitude to Sven Apel, whose feedback and observations made this submission stronger and sharpened our positioning.

This research was funded by the European Union: ERC 101093186 and ERC 101179366. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was also partially supported by the Spanish State Research Agency with the following grants: PREMISE (PID2025-171313OB-I00).

Gaetano Sapia (Author 2) and Marcel Böhme (Author 5) conducted this work while affiliated with MPI-SP.

Data Availability Statement

The source code of Compás and all experimental results are available in the replication package. The repository includes a step-by-step guide to reproduce all results (see `README.md`, Section "Evaluation").

https://doi.org/10.6084/m9.figshare.31855447.v3

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. Software Product Lines. In *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Berlin Heidelberg, 3–15. doi:10.1007/978-3-642-37521-7_1

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society. doi:10.14722/ndss.2019.23412

[3] Eva Barrena, Sergio Bermudo, Alfredo G. Hernández-Díaz, Ana Dolores López-Sánchez, and José Antonio Zamudio Amaya. 2025. Finding the Minimum k-Weighted Dominating Sets Using Heuristic Algorithms. *Mathematics and Computers in Simulation* 228 (2025), 485–497. doi:10.1016/j.matcom.2024.09.010

[4] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*. Springer, 7–20. doi:10.1007/11554844_3

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636. doi:10.1016/j.is.2010.01.001

[6] David Benavides, Chico Sundermann, Kevin Feichtinger, José A Galindo, Rick Rabiser, and Thomas Thüm. 2025. UVL: Feature Modelling with the Universal Variability Language. *Journal of Systems and Software* 225 (2025), 112326. doi:10.1016/j.jss.2024.112326

[7] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering*, Oscar Pastor and João Falcão e Cunha (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–503. doi:10.1007/11431855_34

[8] Leon Bettscheider and Andreas Zeller. 2025. Inferring Input Grammars from Code with Symbolic Parsing. *ACM Trans. Softw. Eng. Methodol.* (November 2025). Just Accepted. doi:10.1145/3776743

[9] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, New York, NY, USA, 713–724. doi:10.1145/3368089.3409729

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. doi:10.1145/2976749.2978428

[11] W. H. Burkhardt. 1967. Generating Test Programs from Syntax. *Computing* 2, 1 (01 Mar 1967), 53–73. doi:10.1007/BF02235512

[12] Andrea Calvagna and Angelo Gargantini. 2012. T-Wise Combinatorial Interaction Test Suites Construction Based on Coverage Inheritance. *Software Testing, Verification and Reliability* 22, 7 (2012), 507–526. doi:10.1002/stvr.466

[13] Justin Campbell and Mike Walker. 2020. Microsoft Announces New Project OneFuzz Framework, an Open Source Developer Tool to Find and Fix Bugs at Scale. Microsoft Security Blog. https://www.microsoft.com/en-us/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/.

[14] Noam Chomsky. 1956. Three Models for the Description of Language. *IRE Transactions on Information Theory* 2, 3 (1956), 113–124. doi:10.1109/TIT.1956.1056813

[15] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[16] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, and C.J. Colbourn. 2003. Constructing Test Suites for Interaction Testing. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 38–48. doi:10.1109/ICSE.2003.1201186

[17] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2006. Coverage and Adequacy in Software Product Line Testing. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 53–63. doi:10.1145/1147249.1147257

[18] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 129–139. doi:10.1145/1273463.1273482

[19] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[20] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference (SPLC 2007)*. IEEE, 23–34. doi:10.1109/SPLINE.2007.24

[21] Merijn de Jonge and Joost Visser. 2002. Grammars as Feature Diagrams. In *ICSR-7 Workshop on Generative Programming (GP2002)*.

[22] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology* 52, 1 (2010), 14–30. doi:10.1016/j.infsof.2009.07.001

[23] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Szeged, Hungary). ACM, 416–419. doi:10.1145/2025113.2025179

[24] José A. Galindo, Jose-Miguel Horcas, Alexander Felferning, David Fernandez-Amoros, and David Benavides. 2023. FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B* (Tokyo, Japan) *(SPLC '23)*. Association for Computing Machinery, New York, NY, USA, 16–19. doi:10.1145/3579028.3609008

[25] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Tucson, AZ, USA). ACM, 206–215. doi:10.1145/1375581.1375607

[26] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311. doi:10.1109/ASE.2013.6693089

[27] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (December 1970), 242–257. doi:10.1147/sj.94.0242

[28] Nikolas Havrikov and Andreas Zeller. 2020. Systematically Covering Input Structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 189–199. doi:10.1109/ASE.2019.00027

[29] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. USENIX Association, Bellevue, WA, 38–38. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[30] Soneya Binta Hossain and Matthew B Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 1475–1487. doi:10.1109/ICSE55347.2025.00098

[31] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*. Association for Computing Machinery, New York, NY, USA, 46–55. doi:10.1145/2362536.2362547

[32] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Carnegie Mellon University, Software Engineering Institute. doi:10.21236/ADA235785

[33] Ralf Lämmel. 2001. Grammar Testing. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 201–216. doi:10.1007/3-540-45314-8_15

[34] Alexander Liggesmeyer, José Antonio Zamudio Amaya, and Andreas Zeller. 2025. Language-Based Protocol Testing. *arXiv preprint arXiv:2509.20308* (2025). arXiv:2509.20308 [cs.SE] doi:10.48550/arXiv.2509.20308

[35] Kuangxiangzi Liu, Dhiman Chakraborty, Alexander Liggesmeyer, and Andreas Zeller. 2025. Synthesizing Precise Protocol Specs from Natural Language for Effective Test Generation. Accepted at the 19th IEEE International Conference on Software Testing, Verification and Validation (ICST 2026). arXiv:2511.17977 [cs.SE] doi:10.48550/arXiv.2511.17977

[36] Phil McMinn. 2004. Search-Based Software Test Data Generation: A Survey. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156. doi:10.1002/stvr.294

[37] Jonathan Metzman and Oliver Chang. 2021. ClusterFuzzLite: Continuous Fuzzing for All. Google Online Security Blog. https://security.googleblog.com/2021/11/clusterfuzzlite-continuous-fuzzing-for.html.

[38] Andrew Murray, Hugo van Kemenade, wiredfool, Jeffrey A. Clark, Alexander Karpinsky, Ondrej Baranovič, Yay295, Christoph Gohlke, Jon Dufresne, Matthew Brett, DWesl, David Schmidt, Konstantin Kopachev, Alastair Houghton, REDxEYE, Russell Keith-Magee, Sandro Mani, Steve Landey, Aarni Koskela, Josh Ware, vashek, Piolie, Jason Douglas, Stanislau T., David Caro, Uriel Martinez, and Riley Lahd. 2026. *python-pillow/Pillow: 12.1.1*. doi:10.5281/zenodo.18604291

[39] Mutagen Developers. 2026. Mutagen Documentation. Accessed March 22, 2026. https://mutagen.readthedocs.io/

[40] Clementine Nebut, Franck Fleurey, Yves Le Traon, and J-M Jezequel. 2006. Automatic Test Generation: A Use Case Driven Approach. *IEEE Transactions on Software Engineering* 32, 3 (2006), 140–155. doi:10.1109/TSE.2006.22

[41] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84. doi:10.1109/ICSE.2007.37

[42] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3293882.3330576

[43] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 459–468. doi:10.1109/ICST.2010.43

[44] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2019), 1980–1997. doi:10.1109/TSE.2019.2941681

[45] Paul Purdom. 1972. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375. doi:10.1007/BF01932308

[46] Debra Richardson, Owen O'Malley, and Cindy Tittle. 1989. Approaches to Specification-Based Testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*. Association for Computing Machinery, New York, NY, USA, 86–96. doi:10.1145/75308.75319

[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105. doi:10.1109/TSE.2023.3334955

[48] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007), 456–479. doi:10.1016/j.comnet.2006.08.008

[49] Kostya Serebryany. 2017. OSS-Fuzz – Google's Continuous Fuzzing Service for Open Source Software. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[50] Jiho Shin, Nima Shiri Harzevili, Reem Aleithan, Hadi Hemmati, and Song Wang. 2024. Retrieval-Augmented Test Generation: How Far Are We? *arXiv preprint arXiv:2409.12682* (2024). doi:10.48550/arXiv.2409.12682

[51] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. doi:10.1145/3540250.3549139

[52] Dominic Steinhöfel and Andreas Zeller. 2024. Language-Based Software Testing. *Commun. ACM* 67, 4 (March 2024), 80–84. doi:10.1145/3631520

[53] Phil Stocks and David A. Carrington. 1996. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering* 22, 11 (1996), 777–793. doi:10.1109/32.553698

[54] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 1–45. doi:10.1145/2580950

[55] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 254–264. doi:10.1109/ICSE.2009.5070526

[56] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). doi:10.1016/j.scico.2012.06.002

[57] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3233027.3233035

[58] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1998. Effect of Test Set Minimization on Fault Detection Effectiveness. *Software: Practice and Experience* 28, 4 (1998), 347–369. doi:10.1002/(SICI)1097-024X(19980410)28:4<347::AID-SPE145>3.0.CO;2-L

[59] Neal E Young. 2008. Greedy Set-Cover Algorithms (1974–1979, Chvátal, Johnson, Lovász, Stein). In *Encyclopedia of Algorithms*. Springer US, 379–381. doi:10.1007/978-0-387-30162-4_175

[60] Michał Zalewski. 2018. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2026-07-30.

[61] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA040 (June 2025), 23 pages. doi:10.1145/3728915

Received 2026-03-26; accepted 2026-06-18