# Checking Test Suite Efficacy Through Dual-Channel Techniques

Constantin Cezar Petrescu[1 [0009−0003−4359−2894]],
Sam Smith[1 [0000−0003−3480−4341]],
Alexis Butler[2 [0000−0002−4199−684X]], and
Santanu Kumar Dash[1 [0000−0002−5674−8531]]

[1] University of Surrey, Guildford, Surrey, United Kingdom
[2] Royal Holloway, University of London, Egham, Surrey, United Kingdom
c.petrescu@surrey.ac.uk, sam.edw.smith@proton.me,
Alexis.Butler.2023@live.rhul.ac.uk, s.k.dash@surrey.ac.uk

**Abstract.** Dynamic Call Graphs trace program execution and are used to model function coverage. They help identify which function calls are missed but do not offer insights on whether those calls are important to cover. We propose a weighted representation of control flow called Natural Call Graphs (NCGs), which can be used to identify important function calls. These weights represent the relevance of the callee to the caller and are computed using information-theoretic reasoning on tokens in the functions. We create a dataset of 1,234 manually verified function calls, containing a mix of relevant and irrelevant functions, from ten Python open-source projects. On this dataset, our approach achieves a peak precision of 78% and a recall of 94% in identifying relevant functions missed by tests.

**Keywords:** Dual-channel Research · Software Testing · Program Analysis.

## 1 Introduction

Identifying functions that are missed by a test is essential for improving the test, and consequently, function coverage. Developers, who are under time-to-market pressure, rarely aim for full function coverage and can benefit from feedback on which function calls to prioritise during testing [13]. For example, a function call may invoke a logger to log diagnostics and this may not be a part of the software requirement. In this case, testing should prioritise calls that are important for the requirement and deprioritise calls to the logger. While Dynamic Call Graphs can help identify the calls that are missed, they do not offer insights on the importance of missed functions.

In this research, we present a technique to rank function calls to establish relevance of the callee to the caller. Our hypothesis is that a callee would contain tokens similar to those in the caller if it is helping the caller in its core objective. On the other hand, it will contain tokens dissimilar to the caller if it is performing

housekeeping tasks like diagnostic logging, that are unimportant to the caller. Our approach falls under the domain of Dual-Channel Software Engineering [5]. Dual-channel software engineering uses information from both the natural language and algorithmic channels in the code. Information from function tokens has been recently used to improve various software engineering tasks such as program hardening [8] and commit deconflation [16]. We present an extension of Call Graphs called Natural Call Graphs whose edges are weighted based on the importance of the callee to the caller. We show that these weights can be relied on to identify important functions that should be covered in a test but are missed by it.

Our main contributions are a methodology and a tool (Section 3) which can rank function calls, based on their importance, that are missed by a test. We evaluate our tool on a ground truth which consists of 1,234 manually vetted cases from the Dynamic Call Graphs of 4,004 integration tests from ten open-source projects of varied popularity (683 - 6.6K stars on GitHub) and size (21K - 449K LOC). Our dataset and tool are publicly available [3]. Our tool achieves a peak precision of 78% and a recall of 94% in identifying relevant functions that are missed (Section 4). We present a selection of cases in Section 4.5 to highlight the capability of the tool to correctly identify functions that should be considered for integration testing.

## 2 Background

In this section, we discuss a motivating example for our work and introduce key terminology before providing an overview of our approach.

### 2.1 Motivating Example

Figure 1 presents a motivating example collected from file localdb.py from Conan [6]. Function `get_login` performs a query on the database to retrieve data, which requires a connection to be established. Function `_connect` is used to return this connection object, making it highly relevant for function `get_login`. In addition, it can be seen that a set of tokens (`connection`, `connect`, `self`) from the caller appear also in a high frequency in the callee. This supports our hypothesis that sharing of tokens between functions correlates with the callee being relevant to the caller.

### 2.2 Our approach

In this section, we give an overview of the program representations we use for identifying important functions that are missed during testing. The details of our approach can be found in Section 3. We first provide some key definitions for terms that are used in the paper.

*Natural Call Graph (NCG).* An NCG is a weighted call graph and it can be defined as a tuple $NCG = (V, E)$. V represents the sets of all functions in the

[3] https://github.com/Constantin-Petrescu/FindIT/

```
1 def get_login(self, remote_url):
2   """ Returns login credentials. This method is also in charge of expiring them. """
3   with self._connect() as connection:
4     try:
5       statement = connection.cursor()
6       statement.execute('select user, token, refresh_token from ...')
7       ...
```

Listing 1: Caller

```
1 def _connect(self):
2   connection = sqlite3.connect(self.dbfile)
3   connection.text_factory = str
4   try:
5     yield connection
6   finally:
7     connection.close()
```

Listing 2: Callee

Fig. 1: Motivating example from Conan library. The caller `get_login` runs a query on the database by executing the callee, `_connect`, to receive a database connection object.

program, such that any function $f_i$ that belongs to the program, then $f_i \in V$. $E$ is the set of direct weighted edges representing the function calls between vertices, where the edge weights represent the callee's relevance to the caller. Such that, if $f_1, f_2 \in V$, there is an edge $f_1 \rightarrow f_2 \in E$ and the relevance of $f_2$ to $f_1$ is defined as the relative importance of $f_2$ to $f_1$'s objective. We compute relevance using conditional entropy of the tokens in $f_2$ with respect to $f_1$.

The NCG contains all methods, and we do not wish to check the relevance of methods that have already been covered by a test. Thus, we end up using two types of NCGs. First is the *Static NCG* which coincides with the NCG of the program. Second is a *Dynamic NCG*, which is a sub-graph of the application's *static NCG* containing only those nodes and edges that are traversed in a test execution. Then, we explore the functions that can be called from the nodes in the *Dynamic NCG* to identify their importance. Based on the relevance scores of the function calls, we classify them as *Core* or *Fringe*, defined below.

*Core and Fringe.* For a caller $f_1$ and a callee $f_2$, the function $f_2$ is a *core* function for $f_1$ if the successful completion of $f_1$'s objective depends on it. Otherwise, it is a *fringe* function for $f_1$.

*Cross-module calls.* While the example in Figure 1 reinforces our hypothesis, cross-module calls are an exception to this. Larger requirements are often implemented across multiple modules that use different namespaces and potentially feature varied function tokens. In order to rank function calls in such cases, we need to identify callees that use dissimilar tokens but are an important part of the software that implements the requirement. In order to identify such methods, we reason over a sequence of function calls instead of a single call. We call this sequence a Candidate Path.

*Candidate Path.* A Candidate Path is sequence of four functions $f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4$ where $f_1, f_2, f_3 \in Dynamic\,NCG$. This means that $f_1$, $f_2$ and $f_3$ are already covered by an integration test. The fourth function $f_4 \in Static\,NCG$ is an untested function at a one-hop distance from $f_3$ but not covered by a test.

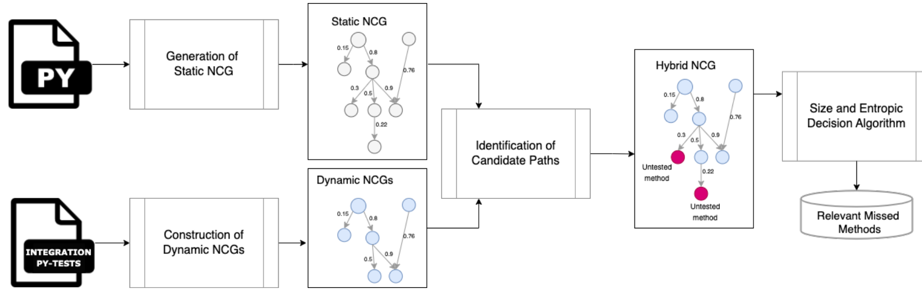



Fig. 2: Our tool's software architecture to identify relevant missed methods by integration tests.

$f_4$ is identified by statically extracting the call graph, and therefore a call to it is represented using the arrow $\xrightarrow{s}$ where $s$ stands for static.

The Candidate Paths allow us to contextualise the relevance score for $\xrightarrow{s}$, by considering the scores for prior calls $f_1 \rightarrow f_2 \rightarrow f_3$, before classifying it as either *core* or *fringe*. An abnormal relevance score could mean a cross-module call.

## 3 Methodology

This section presents a description of the techniques used by the tool. Our tool is presented in Figure 2 and it consists of four components: Generation of the Static NCG, Construction of the Dynamic NCGs, Identification of Candidate Paths by mapping the Static and the Dynamic NCGs, and the Decision Algorithm.

### 3.1 Generation of the Static NCG

Our tool takes any Python project as input and generates a Static NCG. To construct these, we made use of PyCG [20], the most complete tool for Python project Call Graph generation at the time. We extended the PyCG project to collect tokens for both algorithmic and natural language channels by collecting tokens from the function signature, parameters, and body. Conditional entropy [7] is computed between the tokens in the caller and callee and used as edge weights between functions. We store the Static NCG in JSON format.

### 3.2 Construction of the Dynamic NCGs

For each integration test the tool generates a Dynamic NCG by tracing the integration test execution and mapping the trace output with fully qualified function names. These are then used to identify Candidate Paths.

Initially, the integration tests are executed and traced. Based on the project dependencies, execution of the integration tests is performed using a testing framework such as Pytest [12], Nose [18] or Tox [22]. We experimented with

multiple tracing libraries, but the most reliable results came from using the built-in library Trace [21]. Thus, during the execution of an integration test, Trace is used to collect the execution stacks and generate a report with functions calls.

The output files from Trace for each passed integration test contain sets of caller-callee pairs. Since the caller-callee functions have partially qualified names, the second step requires parsing the project files to complete the names. The tool uses regular expression matching to map the partially qualified names to the fully qualified names. With the mapping complete, the edge weights of the Dynamic NCG are set using the entropic information from the project's Static NCG.

### 3.3 Candidates Paths

The tool identifies sequences of three tested functions from the Dynamic NCG. The initial step of Candidate Path generation is the identification of all pairs of two connected dynamic edges where the last node is a leaf. A function is deemed a leaf when it does not appear as a caller in any edges. The tool performs two passes over the edges: first, all edges with a leaf are identified; second, all edges where the callee is the caller of a leaf node are identified to form a sequence of three tested functions.

The next step is to form possible Candidate Paths by finding a function in the Static NCG that is one-hop away from the leaf dynamic node. This requires an iteration over all the edges from the Static NCG. A potential Candidate Path exists if a static edge is found where the caller coincides with the leaf node from the dynamic edge. The last step is to validate that the potential Candidate Path is an actual Candidate Path: this is done by checking that the static edge is not tested by any integration test.

### 3.4 Size and Entropic Decision Algorithm

This component provides the mechanism to decide if the statically selected function is *core* or *fringe* for the path. The Decision Algorithm can leverage two potential indicators size and entropy. A Special Method Filter [2] is also applied to identify *fringe* cases. While the tool can use each indicator and the filter separately or together, the optimal configuration is decided based on the performance analysis presented in Section 4.3.

#### Size Decision

Size is used as an indicator to mark a Candidate Path as *core* or *fringe*. The Decision Algorithm computes the size of the callee as the percentage size relative to the caller. Usually, a small callee can be viewed as a function with only one goal and few instructions. Some examples of such cases are wrappers, getters and setters. Short callee cases could mislead an Entropic Decision Algorithm since the callee could have only different tokens compared to the caller. We hypothesise that the entropic decision can be used if the callee and the caller have

comparable sizes. The goal of the Size Decision Algorithm is not to detect very long functions, which are poor programming practices [14], but rather functions of relatively similar sizes. The Decision Algorithm initially uses the size filter and then passes the Candidate Paths for entropic evaluation.

**Entropy Decision**

The Decision Algorithm uses the entropic values on the path to choose if a Candidate Path is *core* or *fringe*. For a path $f_1 \rightarrow f_2 \rightarrow f_3 \xrightarrow{s} f_4$, the tool computes the entropic metric as the difference between the conditional entropy of $f_4$ given $f_3$ and $f_2$ and the conditional entropy of a $f_3$ given $f_2$ and $f_1$. We use two functions to contextualise a call; using one would lead to fluctuations in the conditional entropy, in case the sizes of the caller and callee are imbalanced. Based on the entropic value, there are three distinct interpretations: similar entropic values, negative values and positive values.

Similar entropic values mean that both the dynamic average and the conditional entropy of the static edge have similar values. This means that the callee is comparable in size with the rest of the functions and shares a consistent amount of tokens. In other terms, the callee performs similar instructions compared to the rest of the functions from the Candidate Path. In this case, we hypothesise that an integration test should also test the static function. A negative entropic value shows that the callee is more diverse and possibly larger than the functions from the rest of the path. We continue the hypothesis by affirming that more diverse functions are desired to be tested in integration tests since they perform new instructions compared to the rest of the path.

A large entropic value shows that the static conditional entropy has a significantly smaller value. This can indicate one of two aspects about the static callee. One aspect is that the callee has a very small number of tokens, and in general that the callee can be considered a wrapper for other functions. The other aspect is that the same tokens are used in the statically selected function. This means that the callee provides utility functionalities for the existing objects. Part of our hypothesis is also that large entropic values indicate functions of lesser importance for the tested path. Thus, the Entropic Decision Algorithm will mark such paths as *fringe* cases, while the rest will be marked *core*.

**Special Method Filter**

Python contains a set of special 'Dunder' methods for built-in data types and classes [2]. Some examples of such methods are: `__cmp__`, `__get__`, `__next__`, `__main__` and many others. In Candidate Paths, such methods will rarely provide any relevant instruction with respect to the path. Thus, this filter ensures that Candidate Paths where the static caller is a special method are marked by the Decision Algorithm as *fringe*. This will aid the tool avoid in marking cases as false positive and provide the users the chance to inspect more relevant *core* cases.

| Name | Stars | LOC | Static NCGs Nodes | Static NCGs Edges | Integration Tests |
|---|---|---|---|---|---|
| Conan | 6.6K | 108K | 3,458 | 1,932 | 2,043 |
| Faust | 6.5K | 46K | 2,662 | 1,676 | 1,081 |
| Docker-API | 6.1K | 21K | 738 | 1,242 | 402 |
| Pex | 2.2K | 449K | 1,800 | 5,110 | 288 |
| Strawberryfields | 683 | 32K | 2,607 | 1,853 | 133 |
| iSort | 5.6K | 23K | 349 | 269 | 23 |
| Emcee | 1.3K | 370K | 155 | 82 | 19 |
| Tox | 3.2K | 260K | 517 | 5,770 | 9 |
| RxPY | 4.4K | 39K | 2,995 | 444 | 3 |
| Sockeye | 1.2K | 15K | 785 | 3,450 | 3 |

Table 1: Shows the benchmark's details: the high-level overview of projects, their Natural Call Graphs representation, the number of integration tests.

## 4 Evaluation

This section provides insights into the performance and utility of our approach by addressing the following research questions:

*RQ1* To what extent do the size and popularity of Python open-source projects influence the quality of their integration tests? (Section 4.1)

*RQ2* To what extent can the tool's performance be improved using different configurations of the Decision Algorithm: a size decision, an entropic decision, or a combination of both? (Section 4.3)

*RQ3* Can the tool consistently distinguish between *core* and *fringe* cases? (Section 4.4)

*RQ4* Are the statically selected functions important to *core* paths? (Section 4.5)

$\mathbf{RQ_1}$ aims to assess how integration testing relates to a project's size and popularity to determine if an integration testing tool is necessary. $\mathbf{RQ_2}$ investigates whether an entropic decision alone is sufficient for identifying *core* cases and whether considering function size can aid in identifying *fringe* functions. $\mathbf{RQ_2}$'s goal is to determine the optimal configuration based on a sample dataset consisting of 10% of the data. $\mathbf{RQ_3}$ evaluates the performance of our approach on the remaining 90% of the data for practical use in software development. $\mathbf{RQ_4}$ aims to generate insights about the nature of the *core* functions.

### 4.1 Benchmark

To answer $\mathbf{RQ_1}$, we constructed a benchmark of ten open-source Python projects listed in Table 1. The selection process for these projects considered two main factors. The first factor was that the projects had a set of integration tests that

| Name | Integration | Dynamic NCGs | | Candidate Paths | | | Ground Truth | |
|---|---|---|---|---|---|---|---|---|
| | Tests | Nodes | Edges | Total | Unique | Unique Untested | *Core* Cases | *Fringe* Cases |
| Conan | 2,043 | 19,531 | 19,869 | 1,659 | 778 | 649 | 490 | 159 |
| Faust | 1,081 | 1,512 | 1,800 | 312 | 26 | 26 | 4 | 22 |
| Docker-API | 402 | 14,843 | 19,474 | 2,630 | 159 | 121 | 18 | 103 |
| Pex | 288 | 14,492 | 16,406 | 1,363 | 90 | 78 | 45 | 33 |
| Strawberryfields | 133 | 7,170 | 7,682 | 3,725 | 244 | 194 | 137 | 57 |
| iSort | 23 | 1,240 | 1,336 | 470 | 37 | 22 | 6 | 16 |
| Emcee | 19 | 348 | 343 | 19 | 10 | 9 | 5 | 4 |
| Tox | 9 | 269 | 301 | 50 | 10 | 10 | 7 | 3 |
| RxPY | 3 | 219 | 308 | 87 | 59 | 58 | 7 | 51 |
| Sockeye | 3 | 719 | 989 | 177 | 95 | 67 | 26 | 41 |
| **Totals:** | 4,004 | 60,343 | 68,508 | 10,465 | 1,508 | 1,234 | 745 | 489 |

Table 2: Shows the Dynamic NCGs built on the integration tests, the distribution of the Candidate Paths across the benchmark, and the distribution of *Core* and *Fringe* cases from the Ground Truth.

could be successfully executed. The second factor considered was the popularity and the size of the project. The selection was made based on a sorted list of the most starred projects. Six projects are relatively small, between 10K and 50K lines of code, while the remaining four are larger, ranging from 108K to 449K lines of code.

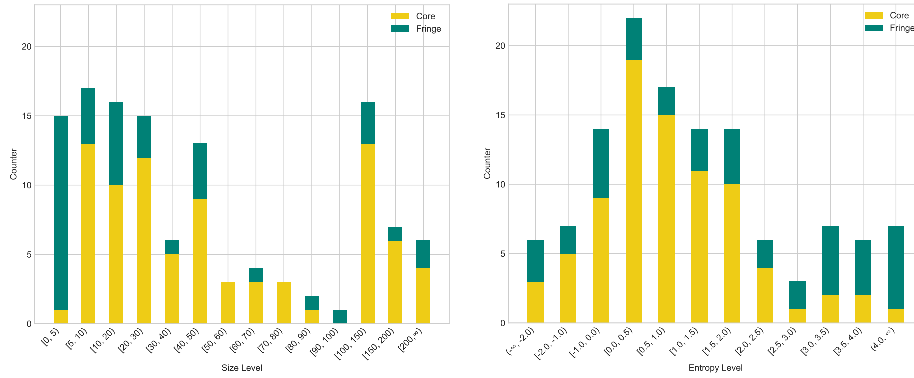
We generated a Static NCG for each project in the benchmark. The general trend observed is that a project's size somewhat correlates with the number of functions and function calls. However, there are some irregularities, such as Strawberryfields, which has only 32K lines of code but 2,607 functions and 1,853 edges. This is because Strawberryfields is written in an object-oriented style and it prioritises efficiency and portability, which requires a heavy reliance on overloading and inner functions. Interestingly, Table 1 shows that the number of stars or the size of a project does not necessarily correlate with the number of integration tests. This highlights the need for tools to identify which parts of a program have been missed in integration testing.

### 4.2 Establishing Ground Truth

Table 2 shows the distribution of core and fringe cases generated from the benchmark. We used 4,004 integration tests to generate Dynamic NCGs. Our tool found 10,465 Candidate Paths by analysing the Static and Dynamic NCGs. We reduced the number of Candidate Paths to 1,234 by removing exact duplicates

| Size Level | Precision | Recall | Accuracy | F1-score | Entropy Level | Precision | Recall | Accuracy | F1-score |
|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0.67 | 1.0 | 0.68 | 0.81 | ∅ | 0.67 | 1.0 | 0.67 | 0.80 |
| **[0, 5)** | **0.76** | **0.99** | **0.78** | **0.86** | [3.5, ∞) | 0.68 | 0.96 | 0.73 | 0.82 |
| [0, 10) | 0.76 | 0.83 | 0.71 | 0.79 | [2.8, ∞) | 0.72 | 0.94 | 0.75 | 0.83 |
| [0, 20) | 0.79 | 0.71 | 0.68 | 0.75 | **[2.6, ∞)** | **0.74** | **0.94** | **0.76** | **0.84** |
| [0, 50) | 0.80 | 0.40 | 0.53 | 0.53 | [2.4, ∞) | 0.73 | 0.92 | 0.75 | 0.83 |
| [0, 100) | 0.82 | 0.28 | 0.48 | 0.41 | [1.0, ∞) | 0.73 | 0.63 | 0.63 | 0.69 |
| [0, 150) | 0.75 | 0.11 | 0.38 | 0.19 | [0.0, ∞) | 0.72 | 0.17 | 0.39 | 0.31 |
| [0, 200) | 0.60 | 0.04 | 0.34 | 0.07 | [-1.0, ∞) | 0.75 | 0.07 | 0.35 | 0.13 |
| | | | | | [-2.5, ∞) | 0.81 | 0.01 | 0.33 | 0.02 |

Table 3: Shows the Precision, Recall, Accuracy and F1-score when the tool marks cases as *fringe* on different levels on the sampled dataset. Left side shows performance for size, while right side shows for entropic level.



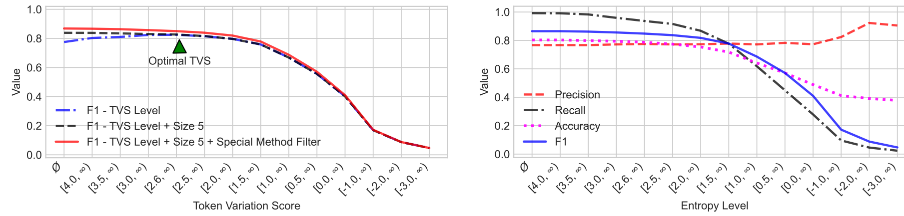

(a) *Core* and *fringe* cases filtered on size. (b) C*ore* and *fringe* cases filtered on entropy.

Fig. 3: Distribution of *core* and *fringe* cases on the sampled dataset.

and paths that had already been tested. To establish the ground truth, each case in the final dataset was labelled as either *core* or *fringe*.

During the labelling process, three raters were involved. Two raters were male PhD students residing in the United Kingdom, with 4 and 8 years of programming experience. The third rater was a male software testing engineer residing in Romania with three years of programming and testing experience. The raters were granted access to the source code for each case and were tasked with evaluating the importance of an untested function to the rest of the tested functions within a candidate path.

Out of the 1,234 cases, the raters have marked 777, 755 and 776 cases as being *core*. The ground truth consists of 786 *core* and 448 *fringe* cases, representing 64% and 36% of the candidate paths, respectively. To assess the level of agreement among the raters, we computed the inter-rater agreement using Cohen's Kappa coefficient. The coefficient was calculated for each pair of raters,




(a) Comparison of F1-scores at different entropic levels

(b) Evaluation metrics for the optimal configuration.

Fig. 4: Tool's performance under different settings on the testing set.

resulting in the values: 0.80, 0.83 and 0.89. These high agreement values indicate a substantial level of consensus among the raters. Based on the ground truth, we initially sample a set of 10% of the data to optimise the size and entropic Decision Algorithm. Then, we evaluate the tool's performance using the rest of the data from the ground truth.

### 4.3 Optimal Configuration for Decision Algorithm

The first step is to understand if the relationship between the sizes of the callee and the caller can influence the decision process of the cases. We hypothesise in Section 3.4 that a small sized callee would provide little insight into the entropic decision, which could significantly increase false positives. For this reason, Figure 3a presents the distribution of *core* and *fringe* cases from the sampled dataset. It can be noticed that there are a large number of *fringe* cases in the low size levels. As the callee's size increases, the number of *core* cases also grows. Cases with a very small callee also translate into fewer operations performed by the callee. In most of these cases, the callees were either utility functions, logging functions or functions to initialise objects. Since a high proportion of grouped *fringe* cases can be seen, we compute and present in Table 3 the precision, recall, accuracy and F1-score for all levels. The F1-score hits the peak at a value of 0.86, where [0, 5) is selected as the size limit for deciding *core* and *fringe* cases. This means the tool would automatically mark each case as a *fringe* where the callee's size is proportionally smaller than 5% of the caller.

Our hypothesis from Section 3.4 suggests that callee-caller candidates with similar entropy values translate into cases where the callee performs *core* actions with respect to the caller. To ensure the validity of the hypothesis, the sampled dataset is analysed from an entropic perspective. Again, Figure 3b presents the distribution of cases. Compared to the size distribution, most entropic levels contain a slightly higher number of *core* cases compared to *fringe* cases. However, the exception levels are when the entropic levels are high and the *fringe* cases become predominant. The goal of the entropic decision is to filter out cases where the callee is very different from the callee. This can be noticed from the larger number of *fringe* cases at the high entropic levels. The metrics computed for all entropic levels are presented in Table 3. Choosing the entropic level at 2.6 makes

the F1-score reach the highest value of 0.84. The tool will use this entropic level to mark any candidate cases where callee's conditional entropy is higher than 2.6 as *fringe*.

### 4.4 Optimal Configuration Over Ground Truth

Based on the sampled dataset, the size and entropic levels are $[0, 5)$ and $[2.6, \infty)$. To evaluate that these levels can be used in tools for software development, an evaluation of the performance for the tool is performed on the remaining 90% of the data from the ground truth. Figure 4a presents the comparison of the F1-score between three different settings of the tool over the ground truth: only entropic, entropic with size, and entropic with size along with Special Method Filter (presented in Section 3.4). The plots reinforce that the entropic level $[2.6, \infty)$ can be selected accurately since the tool nearly reaches the peak performance. In addition, it can be noticed that the optimal configuration of $[0, 5)$ for size, $[2.6, \infty)$ for conditional entropy with Special Method Filter performs with 4% in precision, and 4% in accuracy better compared to the settings where only the entropic level is used or where the entropic and size levels are used. Thus, the performance of the tool is presented in Figure 4b. $\mathbf{RQ_3}$ is answered positively since the tool achieves a precision of 0.78, a recall of 0.94, an accuracy of 0.79 and an F1-score 0.85.

### 4.5 Qualitative Analysis

To answer $\mathbf{RQ_4}$, we present one *core* case in addition to the motivating example from Section 2.1. Additionally, we present two fringe cases, one identified based on entropic difference and the other based on the size difference.

**Case I - *Core*** Figure 5 presents a *core* case collected from Strawberryfields. *test_parameters_with_operation* belongs to file *test_parameters_integration.py*, and the calls from the path can be found in *bosonicbackend/backend.py*. Function `run_prog` runs a Strawberryfields program using the bosonic backend. Then, `init_circuit` starts to instantiate the photonic quantum circuit by initialising the `weights`, `means` and `covs` depending on the different classes of quantum states, such as Cat, Fock or Gaussian. Based on the candidate path, the program enters the Cat state through the function `prepare_cat` from Listing 3. The target of this function is to compute the arrays of weights, means and covariances. It can be seen from Lines 5, 11 and 14 that there are multiple ways to compute the arrays based on different conditions. The untested function `prepare_cat_real_rep` is called by `prepare_cat` on Line 14. This callee is presented in Listing 4 and it continues `prepare_cat`'s objective by computing the arrays in case it is a real-valued state. Lines 4-8 provide a brief overview of the mathematical operations performed by `prepare_cat_real_rep` to calculate `weights`, `means` and `covs`. This case is marked as *core* by the tool due to its entropic difference of 0.02. The strong similarity between the tokens of both functions, as evident in Listing 3 and Listing 4, is also indicated by the entropic difference. Since `prepare_cat` is tested and it shares a similar objective with `prepare_cat_real_rep` (both computing the `weights`, `means` and `covs`), it should be encouraged to also test function `prepare_cat_real_rep`.

```python
def prepare_cat(self, a, theta, p, representation, ampl_cutoff, D):
  """Prepares the arrays of weights, means and covs for a cat state..."""
  ...
  # Case alpha = 0, prepare vacuum
  if np.isclose(a, 0):
    weights = np.array([1], dtype=complex)
    means = np.array([[0, 0]], dtype=complex)
    covs = np.array([0.5 * self.circuit.hbar * ... ])
    return weights, means, covs
  ...
  if representation == "complex":
    return weights, means, covs
  ...
  return self.prepare_cat_real_rep(a, theta, p, ampl_cutoff, D)
```

Listing 3: Caller of the untested function

```python
def prepare_cat_real_rep(self, a, theta, p, ampl_cutoff, D):
  """Prepares the arrays of weights, means and covs for a cat state..."""
  ...
  weights = np.cos(phi) * even_terms * ...
  ...
  means = norm * np.concatenate(weights_real, weights)
  ...
  cov = np.array([0.5 * hbar, 0], [0, (E * v) / ...
  ...
  return weights, means, cov
```

Listing 4: Untested function

Fig. 5: *Core* case from Strawberryfields library. Path is formed from functions: `run_prog`, `init_circuit`, `prepare_cat` and `prepare_cat_real_rep`. Function `prepare_cat_real_rep` computes a set of arrays as a continuation of `prepare_cat` due to specific conditions.

**Case II - *Fringe based on Entropic Difference*** Figure 6 presents a *core* case from *Sockeye*. *test_seq_copy* belongs to file *test_seq _copy_int.py* and the functions are from files: *translate.py* and *inference.py*. The first function, `read_and_translate`, reads input and initiates the translation process by calling the next node in the candidate path. `translate.translate` starts recording the time for logging purposes and starts the actual translation process using `Translator.translate`. Listing 5 shows a brief part of function `Translator.translate`. This function is 125 lines long and it uses a model to translate the input received. When the results output is combined, it will call the untested function, as shown in Lines 4-7 from Listing 5. `_remove_target_prefix_tokens` presented in Listing 6 removes a number of elements from the beginning of `target_ids` and returns the modified list. Although `_remove_target_prefix_tokens` is used twice in `Translator.translate`, offering code reduction and reusability benefits, it has little value for the functions from the candidate path. The tool marks this case as *fringe* to its entropic difference of 3.27, surpassing the entropic level of 2.6. The entropic difference is high due to the extensive actions performed by `Translator.translate` and the token similarity across functions.

**Case III - *Fringe based on Size Difference*** A *fringe* case from Docker-Py is presented in Figure 7. *test_run_with_error* belongs to file *models_containers_test.py* and the calls are from: *models/containers.py* and *types/containers.py*. Function `create` is used to create a container without starting it. Then `_create_container_args` takes user arguments and transforms them into container argu-

```
1 def translate(self, trans_inputs: List[TranslatorInput], fill_up_batches: bool = True)
      -> List[TranslatorOutput]:
2  """Batch-translates a list of TranslatorInputs, returns a list of TranslatorOutputs."""
3   ...
4   if num_target_prefix_tokens > 0 and ...:
5     translation.target_ids = \
6   _remove_target_prefix_tokens(translation.target_ids,
7    translation.target_ids, num_target_prefix_tokens)
8   ...
```

Listing 5: Caller of the untested function

```
1 def _remove_target_prefix_tokens(target_ids, num_target_prefix_tokens)
2  """Remove target prefix tokens from target Ids"""
3   starting_idx = min(len(target_ids), num_target_prefix_tokens)
4   return target_ids[starting_idx:]
```

Listing 6: Untested function

Fig. 6: *Fringe* case due to large entropic difference from Sockeye library. The tested functions collected for this case are: `read_and_translate`, `translate.translate` and `Translator.translate`. Function `_remove_target_prefix_tokens` removes a specific number of elements from the beginning of a list.

ments. Inside `_create_container_args`, a `HostConfig` object is created which triggers a call to `HostConfig.__init__`. Part of the initialisation function is presented in Listing 7. This function validates the inputs received and sets the values as the fields of the `HostConfig` object. In case the input is not as expected, `host_config_value_ error` is called (Lines 4-5). As Listing 8 shows, the function `host_config_value_error` generates the error message based on the parameter and its mismatched value. While `host_config_value_error` improves code reusability, it offers little value from testing it. Our tool marks this path as *fringe* since the Decision Algorithm detects that the callee is too small compared to the caller.

## 5 Threats to Validity

*Internal threats* The tool is subject to two possible internal threats. First threat appears in the generation of the Static NCG from the tool generating the call graph. PyCG achieved a high precision of 99.2%, with an adequate recall of 69.9% [20]. This means that while it correctly identifies instances of function calls, there will be some functions that PyCG will not identify. Inherently, our Static NCG will miss these edges. The goal of our work is not to improve on PyCG, but rather to generate NCGs to suggest missed functions for integration testing.

On the other hand, the construction of the Dynamic NCGs is exposed to a different threat since they are built differently. We utilise Python Trace to track functions' execution while running a test and it outputs functions with partial qualified names. These names are matched with fully qualified names using regular expression matching. This approach is consistent, but functions with identical names in the same namespace may not be matched correctly.

```
1 def __init__(self, version, ...):
2   ...
3   if userns_mode != "host":
4     raise host_config_value_error(
5       "userns_mode", userns_mode)
6   self['UsernsMode'] = userns_mode
```

Listing 7: Caller of the untested function

```
1 def host_config_value_error(param, param_value):
2   error_msg ='Invalid value for {0} param:{1}'
3  return ValueError(error_msg.format(param, param_value))
```

Listing 8: Untested function

Fig. 7: *Fringe* case due to small size difference from Docker-PY library. The path is: `create`, `_create_container_args`, `HostConfig.__init__` and `host_config_value_error`. Function `host_config_value_error` generates the error message if the parameter's value is wrong.

*External threats* Ideally, the natural language channel should harmonise with code instructions. Our tool relies on the relation between the natural language tokens and code instructions to identify the callee's relevance to the caller. However, there will be cases when the natural language is generic or carelessly chosen. Such cases likely become false positives due to the nature of our approach. The users of our tool can inspect *fringe* cases and sometimes identify names carelessly selected.

## 6 Related Work

This section presents the founding notions of dual-channel research and approaches of dual-channel in testing.

**Dual-channel Research** Natural language channel information and its potential benefits in software engineering tasks have been studied for many years. Due to a lack of validation that similar names represent the same thing, Anquetil and Lethbridge defined reliable naming conventions and provided a system along with a set of conditions to assess the efficiency of naming conventions [1]. Caprile and Tonella extended their analysis by examining the lexical, syntactical, and semantical structure of the identifiers [3]. The authors present many potential areas where natural language information could be used, such as program maintainability, program analysis and name recommendations. In fact, they developed a tool to provide more meaningful names for methods [4].

While natural language information has been used for many years, dual-channel research was born based on the *naturalness* property [10]. Hindle et al. built an n-gram language model to harvest and interpret the repetitive patterns as statistical properties. The model was used for code completion as a plugin for Eclipse IDE. Next, Tu et al. added that code is also *localised*, which means that code is locally repetitive [23]. The authors proved that these local repetitions

appear at the file level. Extended n-gram model with a "cache" to capture the local patterns has accuracy increased by 9.4%. The *naturalness* and *localness* properties paved the way for the dual-channel research area.

Casalnuovo et al. formalised that source code is formed using two communication channels [5]. First is the algorithmic channel, which represents all the instructions that the computer executes. The second channel is represented by the natural language channel, which represents the identifiers and comments used in the code. The role of the second channel is to present the purpose of the code in a human-friendly format. Dual-channel research represents solutions that leverage the connection between the two channels and it has been recently used to improve various software engineering tasks [19,17,8,16]. Our work shows that the *localness* property between caller-callee relationships can be used to construct NCGs and identify missed functions for integration testing.

**Dual-Channel Solutions in Testing** Some areas of testing already benefit from using dual-channel approaches. One example is test prioritisation. Quicker identification of failing tests has been achieved by selecting tests based on the similarity computed between code and natural language information [15,9]. Another area is test generation, where models generate assert statements based on the patterns between functions and their tests [24,11]. While our work does not directly compare, we drew inspiration for our methodology. The biggest challenges for dual-channel approaches are finding the suitable intermediate representation and determining the granularity level to capture the correct patterns. In our work, we combined these ideas by constructing NCGs and omitting comments from the function's tokens. We also use the similarity between functions to identify *core* or *fringe* functions.

## 7 Conclusion

This work shows that the relationship between functions can be extracted and used to create new program representations like NCGs. We demonstrate how such representation can aid various software engineering tasks by developing an approach to detect functions missed in integration testing. Our tool achieved an accuracy of 78% with a recall of 94%. However, it could perform even better on projects with clear and established coding and testing guidelines.

The techniques used in this research are not restricted to Python or specific types of testing. While the outcomes may vary, this work can be extended for unit testing or to support any programming language. One prospect would be to examine how testing in different programming languages influences dual-channel approaches. We believe that novel representations including dual-channel information can be used to guide future program analysis techniques. Although such work may be challenging, it may have the potential to substantially enhance the state of program analysis.

## References


1. Anquetil, N., Lethbridge, T.C.: Assessing the relevance of identifier names in a legacy software system. In: Conference of the Centre for Advanced Studies on Collaborative Research (1998)
2. Beazley, D.M.: Python Essential Reference (3rd Edition). Sams, USA (2006)
3. Caprile, B., Tonella, P.: Nomen est omen: analyzing the language of function identifiers. Sixth Working Conference on Reverse Engineering (Cat. No.PR00303) pp. 112–122 (1999)
4. Caprile, B., Tonella, P.: Restructuring program identifier names. Proceedings 2000 International Conference on Software Maintenance pp. 97–107 (2000)
5. Casalnuovo, C., Barr, E.T., Dash, S.K., Devanbu, P., Morgan, E.: A theory of dual channel constraints. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results. pp. 25–28. ICSE-NIER '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377816.3381720
6. Conan: Conan: A python package manager. https://docs.conan.io/en/latest/howtos/other_languages_package_manager/python.html (2015), accessed on 2023-02-16
7. Cover, T.M., Thomas, J.A.: Entropy, Relative Entropy, and Mutual Information, chap. 2, pp. 13–55. John Wiley and Sons, Ltd (2005). https://doi.org/https://doi.org/10.1002/047174882X.ch2
8. Dash, S.K., Allamanis, M., Barr, E.T.: Refinym: Using names to refine types. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 107–117. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236042
9. Greca, R., Miranda, B., Gligoric, M., Bertolino, A.: Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration. In: 2022 IEEE/ACM International Conference on Automation of Software Test (AST). pp. 115–125 (2022). https://doi.org/10.1145/3524481.3527223
10. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. pp. 837–847. ICSE '12, IEEE Press (2012)
11. Kampmann, A., Havrikov, N., Soremekun, E.O., Zeller, A.: When does my program do this? learning circumstances of software behavior. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1228–1239. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409687
12. Krekel, H., Team, P.D.: Pytest - testing framework. https://pytest.org (2003), accessed on 2023-03-03
13. Martin, D., Rooksby, J., Rouncefield, M., Sommerville, I.: 'good' organisational reasons for 'bad' software testing: An ethnographic study of testing in a small software company. In: Proceedings of the 29th International Conference on Software Engineering. pp. 602–611. ICSE '07, IEEE Computer Society, USA (2007). https://doi.org/10.1109/ICSE.2007.1
14. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)

15. Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A.: Fast approaches to scalable similarity-based test case prioritization. In: Proceedings of the 40th International Conference on Software Engineering. pp. 222–232. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180210
16. Partachi, P.P., Dash, S.K., Allamanis, M., Barr, E.T.: Flexeme: Untangling commits using lexical flows. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 63–74. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020)
17. Pârtachi, P.P., Dash, S.K., Treude, C., Barr, E.T.: Posit: Simultaneously tagging natural and programming languages. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1348–1358. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380440
18. Pellerin, J., Team, N.D.: Nose - testing framework. https://pypi.org/project/nose/ (2010), accessed on 2023-03-03
19. Petrescu, C.C., Smith, S., Giavrimis, R., Dash, S.K.: Do names echo semantics? a large-scale study of identifiers used in c++'s named casts. Journal of Systems and Software **202**, 111693 (2023). https://doi.org/https://doi.org/10.1016/j.jss.2023.111693
20. Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D., Mitropoulos, D.: Pycg: Practical call graph generation in python. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1646–1657 (2021)
21. Team, P.D.: Trace - python module to trace program's execution. https://docs.python.org/3/library/trace.html (1991), accessed on 2023-03-03
22. Tox: Tox - automation project. https://tox.wiki/en/latest/index.html (2010), accessed on 2023-02-16
23. Tu, Z., Su, Z., Devanbu, P.: On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 269–280. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635875
24. Watson, C., Tufano, M., Moran, K., Bavota, G., Poshyvanyk, D.: On learning meaningful assert statements for unit test cases. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1398–1409. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380429