MediumDarwin: LittleDarwin Grows with Performance and Research-oriented Extensions

Sajjad Hesamipour*, Thomas Laurent†, Anthony Ventresque‡
School of Computer Science and Statistics
SFI Lero @ Trinity College Dublin
Dublin, Ireland
Email: hesamips@tcd.ie*, tlaurent@tcd.ie†, anthony.ventresque@tcd.ie‡

***Abstract*—Software testing is essential to ensure the reliability and correctness of software systems. However, the effectiveness of testing is highly dependent on the quality of the test suites themselves. Mutation analysis, a powerful technique for evaluating the quality of tests, introduces small changes into the code and checks whether the tests detect them. Despite its strengths, mutation analysis faces challenges in scalability due to the high computational cost of compiling and running tests against mutants.**

This paper presents *MediumDarwin*, a substantially upgraded version of the original LittleDarwin, initially introduced as a research prototype. Our enhanced version retains the original foundational architecture but introduces significant new capabilities and performance optimisations that transform it into a robust platform for both industrial use and advanced research.

The enhancements made to LittleDarwin include: (1) persistent storage of mutation results in a relational database to facilitate advanced analysis, (2) coverage-based test selection optimisation to minimise test executions, (3) implementation of mutant schemata to reduce compilation overhead, (4) enhanced mutation operators alongside safeguards against non-compilable mutants, and (5) dynamic subsumption graph computation for efficient mutant analysis. These innovations collectively improve the tool's scalability and practical utility in software quality assurance in both industrial and research contexts.

A screencast demonstrating the use of MediumDarwin is available at https://www.youtube.com/watch?v=Zsd3pZt63AE.

***Index Terms*—Software Testing, Mutation Analysis, Subsumption Analysis, Mutant Schemata, Java**

I. Introduction

Software testing is a fundamental and indispensable process in developing and maintaining reliable software systems. The primary objective of testing is to ensure that software applications perform according to their intended functionality and remain free from defects. Among various testing quality metrics, mutation analysis stands out as a powerful test adequacy criterion. This approach involves introducing artificial defects (mutations) into the System Under Test (SUT) and assessing the test suite's ability to detect these mutations. While mutation analysis offers valuable insights into a test suite's effectiveness, its computational cost can be a hindrance, particularly for large software projects [1].

LittleDarwin is a prominent mutation testing tool for Java, designed for both academic and industrial use cases. Prior research has shown that the tool lacks performance optimisations [2]. However, its modular architecture enhances flexibility. Moreover, LittleDarwin holds a unique position as the only open-source, state-of-the-art mutation testing tool operating at the Java source-code level. In contrast, most alternatives function at the bytecode level [3]. These features make this tool highly valuable for research and practical applications.

This paper presents significant technical enhancements to LittleDarwin that collectively improve both its computational efficiency and analytical capabilities. We refer to this enhanced version as *MediumDarwin* and make the tool and its source code available online [4]. A key enhancement involves the integration of a SQLite relational database [5], which enables large-scale mutation analysis by streamlining data management and comprehensive reporting. The tool's performance has been optimised through code coverage analysis, allowing selective exclusion of tests that do not exercise mutated statements, thereby reducing execution time without compromising results' validity. Substantial gains in efficiency have been achieved through the implementation of mutant schemata, which consolidates all mutants generated from a Java class into a single file rather than creating individual files for each mutant. This architectural refinement eliminates redundant recompilation cycles while supporting parallel mutant execution. Additionally, the introduction of dynamic mutant subsumption analysis provides researchers with enhanced capabilities for mutant prioritisation and more efficient test suite evaluation.

The integration of a relational database, coverage-driven test selection, mutant schemata, refined mutation operators, and dynamic subsumption analysis collectively represent substantial advancements in the tool's functionality and performance. Particularly, the refined mutation operators significantly reduce non-compilable mutant generation and introduce some missing mutants, providing a more accurate mutation score. These corrections address issues that biased mutation scores and consequently compromised the reliability of results.

The structure of this paper proceeds as follows: Section II introduces fundamental concepts in mutation analysis and LittleDarwin. Section III details the enhancements implemented in *MediumDarwin*. Section IV presents preliminary experimental results, and Section V outlines directions for future research. Finally, Section VI summarises our key contributions and discusses their implications for mutation testing tools.

II. Background

This section presents some background information about mutation analysis and LittleDarwin.

A. Mutation analysis

Mutation analysis produces multiple syntactically distinct variants of the SUT, commonly termed mutants [6] by introducing simple changes (mutations) in the original code. A mutant may include one mutation (First Order Mutants, FOMs) or multiple mutations (Higher Order Mutants, HOMs). Each mutation is defined by its position in the code and operator. The mutation position specifies the location of the mutation within the SUT, typically a line and column number in the source code or a node identifier in the Abstract Syntax Tree (AST), while the mutation operator defines a specific syntactic transformation, e.g., replacing + with $-$, or $\star$ with $/$.

Mutation analysis then runs the SUT's tests against the different mutants it produced and checks whether their results are different. The underlying hypothesis is that a strong test suite should effectively distinguish between the original program and its mutants, as they represent potential faults. Mutants that are successfully detected by the test suite are classified as *killed*, whereas those that remain undetected are labelled as *surviving* mutants. However, it is important to acknowledge that not all surviving mutants necessarily indicate deficiencies in the test suite. In some cases, a surviving mutant may be semantically equivalent to the original program, rendering it undetectable by any test case. Such mutants are referred to as *equivalent* mutants, and their detection is one of the main challenges in mutation analysis [7]. The primary metric used in mutation testing is the mutation score, defined as the proportion of killed mutants to the total number of non-equivalent mutants;

$$MS = \frac{\textit{\# killed mutants}}{\textit{\# mutants} - \textit{\# equivalent mutants}}. \quad (1)$$

Running all tests against different mutants makes mutation analysis computationally expensive, especially on larger projects with a large number of mutants and many tests. This cost hinders the widespread adoption of mutation analysis' widespread adoption in industrial settings [1]. However, it is possible, for each mutant, to identify tests that do not execute the mutated parts of the code and can be excluded while running the mutant, as they cannot detect it. This optimisation strategy, known as regression test selection [8], operates on the principle that only a subset of tests exercises the modified portion of the SUT. We use "regression test selection" and "test selection" interchangeably in the rest of the paper.

In traditional mutation testing, each mutant is typically generated as a separate source file. For compiled languages like Java, this requires recompiling every mutant before executing tests against it, significantly increasing computational overhead. To address this inefficiency, researchers have developed the mutant schemata approach, where all mutations are combined into a single meta-program that only requires one initial compilation. Individual mutations can then be selectively activated at runtime through conditional logic. Mutant schemata is one of the most extensively studied cost-reduction methodologies in mutation testing research [9].

One of the principles in mutation analysis is that not all mutants have equal value [10]. A key approach for evaluating mutant prioritisation lies in examining subsumption relationships between mutants. When mutant A subsumes mutant B, detecting mutant A ensures mutant B will also be killed. This means that if a test suite detects A then B cannot uncover a new fault and there is no need to execute tests against B as we know it is also detected. These relationships can only be empirically approximated by analysing the killing test sets of mutants, where dynamic subsumption is established if the set of tests killing mutant A forms a proper subset of those killing mutant B [11]. This relationship can be formally represented with a graph, where nodes denote mutants and directed edges indicate subsumption relationships.

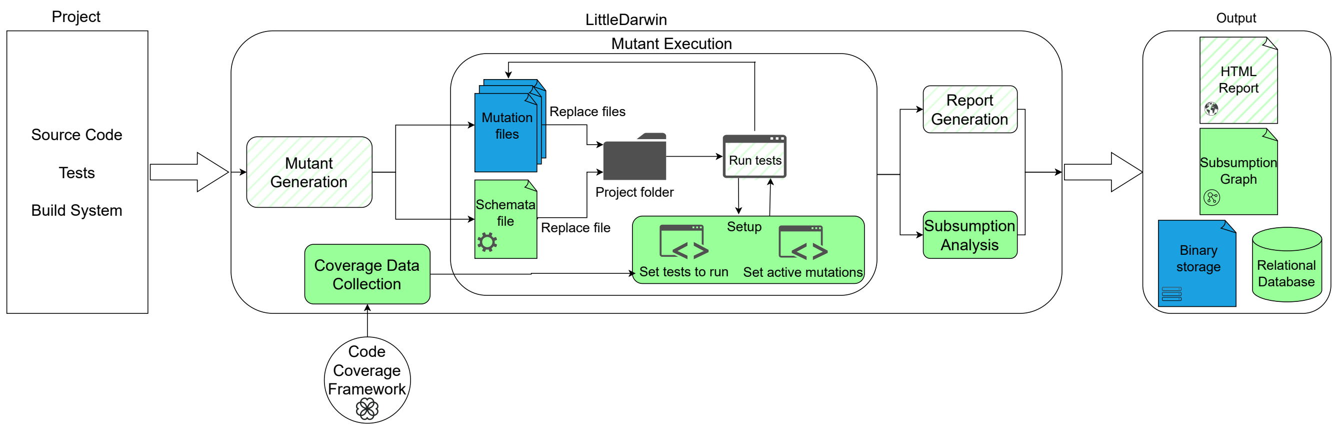
B. LittleDarwin

LittleDarwin is a command-line interface (CLI) mutation testing tool for Java [2]. Users configure it by specifying project paths, build and test commands via CLI parameters. The tool operates in two phases: mutation generation and mutant execution, each controllable via dedicated command-line flags. Figure 1 illustrates LittleDarwin's architecture, with original components highlighted in blue, and the newly added components described in this paper in green.

In the mutation generation phase, LittleDarwin finds all possible mutations for the SUT based on the files the users designated for analysis and the operators they chose to use. It then creates all possible mutants based on the order of mutation the user has chosen and generates a new source file for each mutant. Generating one file per mutant leads to a large number of generated files and transactions with the disk. All these different mutant files contain mostly the same source code with only the mutant's mutation(s). The generated mutants' features are recorded in a binary database (a `shelve` database).

LittleDarwin performs mutation analysis at source-code level. Empirical evidence suggests that source-level mutation analysis offers distinct advantages, including (1) reduced mutant volume and (2) lower rates of equivalent and duplicate mutants compared to bytecode-based approaches [12].

LittleDarwin's mutation operators are applied purely at the syntactic level, omitting semantic checks such as type safety. This approach produces non-compilable mutants, precluding any test execution against them. These mutants provide no value to the mutation analysis and even distort the analysis by artificially inflating the mutation score [13]. Additionally, some bugs in the original LittleDarwin skip some of the possible mutants. Listing 1 shows an example program of these cases. First, the tool generates invalid mutants such as Mutant 0, where the `NullifyInputVariable` operator ignored the `final` modifier on the variable, producing non-compilable code. Second, it misses valid mutation opportuni-



🟦 original LittleDarwin elements; 🟩 Features added in MediumDarwin; ▨ Features modified in MediumDarwin.

Fig. 1: Architecture of LittleDarwin. New components are highlighted in green and the modified ones in hatch green. Dropped elements of the original implementation are highlighted in blue.

```
public boolean[] findHellos(final String[] array) {
    // MUT0 (non-compilable): array = null;
    boolean[] hellos = new boolean[array.length];
    for (int i=0; i<array.length; i++) {
        hellos[i]= array[i].equals("hello");
    }
    // MUT1 (missing): return new boolean[]{};
    return hellos;
}
```

Listing 1: Example for non-compilable mutants generated by LittleDarwin

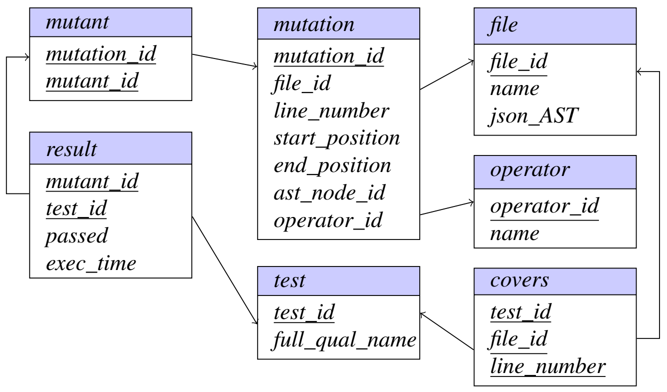


Fig. 2: Relational diagram of the databased used.

ties, such as Mutant 1, where the `RemoveMethod` operator fails to process certain methods due to insufficient type checking of return values.

After generating all mutants, the tool iteratively copies each mutated file to the project directory, replacing the original file. Subsequently, it executes the commands provided by the user to re-compile the project and run the tests against the mutant, recording the command's output in designated text files for further analysis. Finally, it also generates HTML reports summarising mutation outcomes.

III. MediumDarwin

This section outlines the enhancements implemented in the new version of the tool.

A. *Relational database*

This improvement replaces the binary file format used in LittleDarwin to record information about the mutants with a relational database implemented using SQLite [5]. Figure 2 gives an overview of the database's schema. This database records information about the project being analysed (table `file`), the generated mutants (tables `operator`, `mutation`, `mutant`, and `part_of_mutant`), the test suite used for mutation analysis (tables `test` and `covers`), and the results of mutation analysis (table `result`).

The relational architecture facilitates efficient data organization and retrieval (leveraging SQLite, a widely-adopted open-source relational database management system), while enabling sophisticated query capabilities for analytical purposes (e.g., mutation matrix) – functionalities which were not feasible with LittleDarwin's dictionary-based storage. The enhanced tool automatically generates this relational database within the results directory for convenient access.

B. *Coverage-based test selection*

As discussed in Section II, tests that do not cover a mutated line cannot detect the corresponding mutant. Executing them unnecessarily increases the cost of mutation analysis. To address this, MediumDarwin introduces coverage-based test

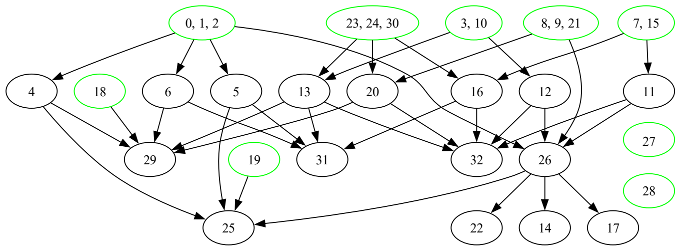


Fig. 3: Subsumption graph for the Triangle project using traditional mutation operators.

selection, reducing redundant test executions, i.e., it only runs tests covering the mutated line(s) against a mutant.

To implement this optimisation, we introduce a coverage data collection phase preceding mutant execution. This phase executes the complete test suite while identifying test-to-line of code coverage relationships. For coverage data collection, we used OpenClover [14], a code coverage data collection tool. MediumDarwin first collects the coverage data provided by OpenClover by running tests against the unmutated SUT and stores it in the relational database (specifically within the `test` and `covers` tables). Later, in mutant execution phase, it excludes tests that do not cover each mutant before running the user-provided test command, and records the result in the relational database.

In order to use OpenClover and to exclude tests our implementation relies on the project under analysis to be built with either Maven [15] or Ant [16], with plans of supporting Gradle [17]. Still, this does not severely limit MediumDarwin as Maven and Ant cover approximately 78% of Java projects according to a 2022 survey [18] and are extensively used by at least 15% of a 2024 survey [19] (only 30.3% declared having used Java extensively).

To enable test selection in MediumDarwin, users only need to execute the tool with the `--code_coverage` (short form `-q`) command-line argument.[1].

C. Mutant Schemata

To further reduce compilation overhead, we integrated mutant schemata into MediumDarwin. As discussed in Section II, this technique reduces compilation overhead by requiring only a single compilation pass for all mutants after schemata generation. Additionally, mutant schemata significantly decreases disk I/O operations by avoiding repetitive file replacements for individual mutants. Furthermore, this approach enables concurrent mutant execution, facilitating parallel processing and performance gains —- capabilities absent in LittleDarwin as it requires file replacement for each mutant. Users can enable this feature via the `--schemata` (short form `-e`) command-line option. The number of concurrent jobs used to execute mutants can be set via `--jobs-no` option. Schemata can be combined with test selection by using both command line options when running the tool.

[1]For Ant projects, where the name of the targets specifying and running the tests can differ from default configurations, users can specify them using the `--test_target_name` or `--junit_target_name` parameters

The mutant schemata implementation, which relies on conditional and ternary expressions, cannot accommodate all mutation types. For instance, mutations involving compile-time constants require per-mutant recompilation, as they cannot be represented through conditional logic. The tool identifies such cases through static code analysis and compilation log inspection, labelling them as Compile-Time Mutants. For these exceptions, MediumDarwin defaults to the original file-by-file replacement strategy. However, since such mutants typically constitute a minor fraction of the total mutant population, the modified tool still achieves substantial reductions in recompilation costs compared to the original version.

D. Refined mutation operators

As explained in Section II, LittleDarwin's original implementation produces non-compilable mutants while skipping some valid mutants. Recognising that non-compilable mutants fail to serve the core purpose of mutation analysis – the assessment of test suite quality – we augmented LittleDarwin's mutation operators to automatically filter out such non-viable mutants through type checking. Additionally, the refined operators now generate originally skipped mutants. Our modifications, which operate without user intervention, demonstrate measurable changes to the mutants generated from a codebase, as quantified by the preliminary evaluation presented in Section IV. This enhancement represents a significant refinement to the tool's core functionality and validity.

E. Dynamic subsumption analysis

Building upon the work by Parsai et al. [20], who first computed dynamic subsumption analysis using LittleDarwin, we have integrated this capability into the tool. The implementation enables users to generate subsumption graphs on demand by specifying a dedicated command-line option (`-s`). To compute the subsumption graph MediumDarwin first obtains the result of each test on each mutant. As this is a costly operation subsumption analysis in MediumDarwin automatically triggers coverage-based test selection to reduce the number of test executions. Once it has obtained the results, the tool automatically constructs the graph structure based on the constructed mutation matrix and exports it in widely compatible formats (`pajek`, `GML`, and `DOT`) for seamless integration with standard graph visualisation tools. Figure 3 presents the resulting subsumption graph for the triangle project, rendered in `graphviz`. The mutant identifiers which are written in green circles are those of the so-called minimal mutants.

IV. Evaluation

This section presents a preliminary evaluation of the impact of the modified mutation operators on mutant generation. We ran both LittleDarwin and MediumDarwin on 15 open source projects of varying sizes and compared the number of mutants generated by each tool. Due to space constraints Table I only

TABLE I: The effect of modifying mutation operators

Project name	LOC	Original	Reduced	Added	Δ(%)
Commons-dbutils	2,227	1,525	468	9	-30.10
Commons-cli	1,971	1,482	269	0	-18.15
Commons-validator	4,659	3,483	593	0	-17.03
Jackson-dataformat-xml	4663	2572	5	6	0.04
Jettison	3177	2210	0	1	0.05
Jackson-core	24635	15890	121	132	0.07

shows results for the 3 projects where MediumDarwin lead to the largest relative increase in the number of mutants and the three projects where it leads to the largest decrease. Full results are available in the paper's companion repository [4].

The data reveals substantial variation in how projects respond to operator modifications. The largest reduction occurred in Commons-dbutils (-30.10%), followed by Commons-cli (-18.15%) and Commons-validator (-17.03%). In contrast, some projects showed negligible changes, with Jackson-dataformat-xml (+0.04%), Jettison (+0.05%), and Jackson-core (+0.07%) experiencing minimal increases. Across all projects, we observed an average reduction of 11.13% in mutant generation.

A key finding is the significant decrease in non-compilable mutants, with 1,456 non-compilable mutants eliminated and only 148 new mutants introduced. The results suggest that coding style has a greater influence on the effectiveness of operator modifications than project size (measured by LOC). For instance, Commons-dbutils (2,227 LOC) showed the largest reduction, while much larger projects like Jackson-core (24,635 LOC) exhibited minimal change.

V. Future work

Here, we outline planned experimental studies for further validation of the impact of the introduced enhancements.

To systematically evaluate the impact of each individual contribution, we plan to conduct rigorous empirical studies assessing the performance characteristics of the implemented features. This investigation will encompass: (1) quantitative analysis of the relationship between code coverage metrics and test selection efficacy, (2) comparative performance benchmarking of different tool versions, and (3) identifying key factors driving improvements. These experiments will provide evidence-based validation of the techniques' effectiveness across diverse development contexts. Also, we will extend support to Gradle projects to enhance practical applicability.

VI. Conclusion

Mutation analysis serves as a critical methodology for evaluating the adequacy of test suites and ensuring software quality. This paper presents different enhancements we made to LittleDarwin, a prominent open-source mutation testing tool for Java that operates at the source code level, how they can be used, and how they can benefit different users.

The contributions of this work include (1) the integration of a relational database system for efficient data management, (2) the incorporation of coverage-based test selection to reduce execution time, (3) the implementation of mutant schemata generation to optimise performance, (4) the refinement of the tool's mutation operators to avoid non-compilable mutants and generate the skipped ones, and (5) the addition of dynamic subsumption analysis capabilities.

These advancements collectively enhance the tool's performance, reliability, and practical utility for both software engineering researchers and practitioners. The extended capabilities increase the tool's efficiency and expand its potential applications in empirical software engineering research.

References

[1] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, Nov. 2019.

[2] A. Parsai, A. Murgia, and S. Demeyer, "Littledarwin: a feature-rich and extensible mutation testing framework for large and complex java systems," in *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*. Springer, 2017, pp. 148–163.

[3] C. Bockisch, G. Taentzer, and D. Neufeld, "Mmt: Mutation testing of java bytecode with model transformation," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2023, pp. 35–39.

[4] S. Hesamipour, "Complexsoftwarelab/mediumdarwin: Release 1.0.0.0," 2025, mutation testing framework. [Online]. Available: https://github.com/ComplexSoftwareLab/MediumDarwin

[5] SQLite, "SQLite: A small, fast, self-contained, high-reliability, full-featured, sql database engine," 2025, accessed: 2025-05-14. [Online]. Available: https://www.sqlite.org/

[6] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, 1978.

[7] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[8] L. Chen and L. Zhang, "Speeding up mutation testing via regression test selection: An extensive study," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*.

[9] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[10] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 80–89.

[11] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014.

[12] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 114–124.

[13] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 354–365.

[14] "Openclover - java, groovy and aspectj code coverage tool," 2023, accessed current date. [Online]. Available: https://openclover.org/

[15] [Accessed 15-05-2025]. [Online]. Available: https://maven.apache.org/

[16] [Accessed 15-05-2025]. [Online]. Available: https://ant.apache.org/

[17] [Accessed 30-05-2025]. [Online]. Available: https://gradle.org/

[18] "2022 java developer productivity report — jrebel," 2022. [Online]. Available: https://www.jrebel.com/resources/java-developer-productivity-report-2022

[19] "2024 Stack Overflow Developer Survey," [Accessed 13-05-2025]. [Online]. Available: https://survey.stackoverflow.co/2024/technology

[20] A. Parsai and S. Demeyer, "Dynamic mutant subsumption analysis using littledarwin," in *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, ser. A-TEST 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–4.