GenProg-Python: A Syntax-Aware Evolutionary Framework for Automated Program Repair

Seth Whenton, Flihi Arij, Saeid Khalilian

January 2026

1 Introduction & Problem Definition

1.1 The Imperative of Automated Program Repair

Modern software development is characterized by increasing complexity, leading to an inevitable rise in software defects. Automated Program Repair (APR) seeks to mitigate the high cost of manual debugging by treating program repair as an optimization problem. The primary objective of this project was to replicate and adapt the seminal GenProg evolutionary algorithm (Le Goues et al., 2012). While the original GenProg implementation targeted C programs using legacy mutation operators, this project focuses on a modern, generic Python implementation. The goal was to construct a self-healing system capable of taking an arbitrary buggy Python program and its associated test suite, and evolving a patch that satisfices all correctness constraints without human intervention.

1.2 From Ad-Hoc Scripts to a Generic Framework

A critical design requirement for this implementation was generality. Early iterations of APR tools often relied on hardcoded heuristics specific to a single problem domain. In contrast, our framework was engineered to be entirely agnostic to the input program. By leveraging Python's `ast` (Abstract Syntax Tree) module for code parsing and `importlib` for dynamic execution, the system can load, analyze, and repair any Python function provided it adheres to a standard interface. This shift allows the tool to address a diverse range of defect classes—from boolean logic errors to off-by-one boundary conditions—using a single unified evolutionary engine. The system's architecture separates the *evolutionary logic* (the genetic algorithm) from the *program representation* (the benchmarks), ensuring the repair process is scalable and reusable.

1.3 The Grand Challenge: The "Python Syntax Wall"

The most significant technical hurdle in adapting GenProg to Python is what we term the "Python Syntax Wall". The original GenProg operated on C code, where block structure is delimited by braces `{}` and syntax is largely robust to whitespace changes. Python, however, enforces strict, whitespace-sensitive indentation rules to define control flow.

This creates a unique problem for Genetic Programming (GP). Standard GP operators like Crossover (swapping lines between two programs) or Swap Mutation (exchanging two lines of code) are highly destructive in Python. For example, moving a line from inside a deep `if/else` block to the global scope will almost certainly raise an `IndentationError` or `SyntaxError`. In a standard GP implementation, these "invalid" individuals would simply fail evaluation, potentially wasting up to 90% of the computational budget on non-compilable code.

To overcome this, our implementation introduces a **Syntax-Aware Genetic Operator** design. Rather than relying on the fitness function to filter out broken syntax, we integrated

validation logic directly into the operator loop. Every mutation or crossover attempt is checked against Python's AST parser (`ast.parse`) in real-time. If an operation yields invalid syntax, it is immediately discarded and re-attempted. This ensures that the evolutionary search traverses only the manifold of *syntactically valid* programs, dramatically increasing the efficiency of the search and allowing the algorithm to focus on *semantic* correctness rather than fighting the language parser.

2 Methodology - The Evolutionary Engine

2.1 The Evolutionary Loop: An Overview

The core of our APR framework is a customized Genetic Algorithm (GA) explicitly designed to mimic the biological process of natural selection. By treating a program as a genome (a sequence of source code lines), the system iteratively improves the population's "health" (test passage rate). The evolutionary lifecycle follows the canonical GenProg algorithm structure with specific adaptations for the Python language:

1. **Initialization:** An initial population of k variants (default 40) is generated. The first variant is the original buggy program (the "Adam" variant). The remaining 39 are created by applying random mutations.
2. **Evaluation:** Each variant is compiled and executed against the benchmark's test suite. A fitness score is assigned based on the weighted sum of passing tests (`1.0` for positive tests, `10.0` for negative tests).
3. **Selection:** The "fittest" 50% of the population are selected as survivors via truncation selection.
4. **Repopulation:** Survivors breed to produce new offspring via Crossover and Mutation, ensuring continuous exploration of the search space.

2.2 Fault Localization: The Guide Mechanism

To make the search space tractable, we implemented Spectrum-Based Fault Localization (SBFL). Using Python's `sys.settrace()` facility, the system records exactly which lines of code are executed during failing tests versus passing tests.

- **Suspicion Weighting:** Lines executed primarily during failing tests are assigned high "suspicion weights" (e.g., 1.0 or 0.1).
- **Guided Mutation:** Mutation operators use Roulette Wheel Selection based on these weights to bias modifications toward the likely bug location.

2.3 Crossover: Syntax-Aware Recombination

We implemented a One-Point Crossover operator using an "Optimistic Retry" strategy to solve the syntax wall:

1. The algorithm selects a random pivot (line index).
2. It constructs the offspring by splicing Parent A and Parent B.
3. It immediately runs `ast.parse()`. If an `IndentationError` occurs, the offspring is discarded and a new pivot is attempted (up to 5 times).

2.4 Mutation Operators: Beyond Line Granularity

Beyond standard statement-level operators (**Delete**, **Insert**, **Swap**), we introduced expression-level operators to handle boundary bugs:

- **Expression Mutation:** Identifies and mutates comparison operators (`<`, `<=`, `==`, `!=`, `>`, `>=`). This allows character-level repair for off-by-one errors.
- **Boolean Mutation:** Flips logical connectors (`and`, `or`).

3 Implementation Technicalities

3.1 Generic & Dynamic Test Harness

3.1.1 Placeholder for Images

4 Methodology - The Evolutionary Engine

4.1 The Evolutionary Loop: An Overview

The core of our APR framework is a customized Genetic Algorithm (GA) explicitly designed to mimic the biological process of natural selection. By treating a program as a genome (a sequence of source code lines), the system iteratively improves the population's "health" (test passage rate). The evolutionary lifecycle follows the canonical GenProg algorithm [**?**] structure with specific adaptations for the Python language:

1. **Initialization:** An initial population of k variants (default 40) is generated. The first variant is the original buggy program (the "Adam" variant). The remaining 39 are created by applying random mutations to the original code, seeded with diverse potential fixes.
2. **Evaluation:** Each variant is compiled and executed against the benchmark's test suite. A fitness score is assigned based on the weighted sum of passing tests (`1.0` for positive tests, `10.0` for negative tests).
3. **Selection:** The "fittest" 50% of the population are selected as survivors. This truncation selection ensures that only partially working solutions propagate their genetic material.
4. **Repopulation (Recombination & Mutation):** To restore the population to its full size, survivors breed to produce new offspring via Crossover (combining traits from two parents) and Mutation (random alterations), ensuring continuous exploration of the search space.

4.2 Fault Localization: The Guide Mechanism

Genetic programming on source code is searching for a needle in a haystack. To make this search tractable, we implemented Spectrum-Based Fault Localization (SBFL). Using Python's `sys.settrace()` facility, the system records exactly which lines of code are executed during failing tests versus passing tests.

- **Suspicion Weighting:** Lines executed primarily during failing tests are assigned high "suspicion weights" (e.g., `1.0` or `0.1`).
- **Guided Mutation:** When the mutation operator selects a line to modify (e.g., Delete or Swap), it uses Roulette Wheel Selection based on these weights. This biases the evolutionary modifications toward the likely location of the bug, mimicking a human developer focusing on the "crash site."

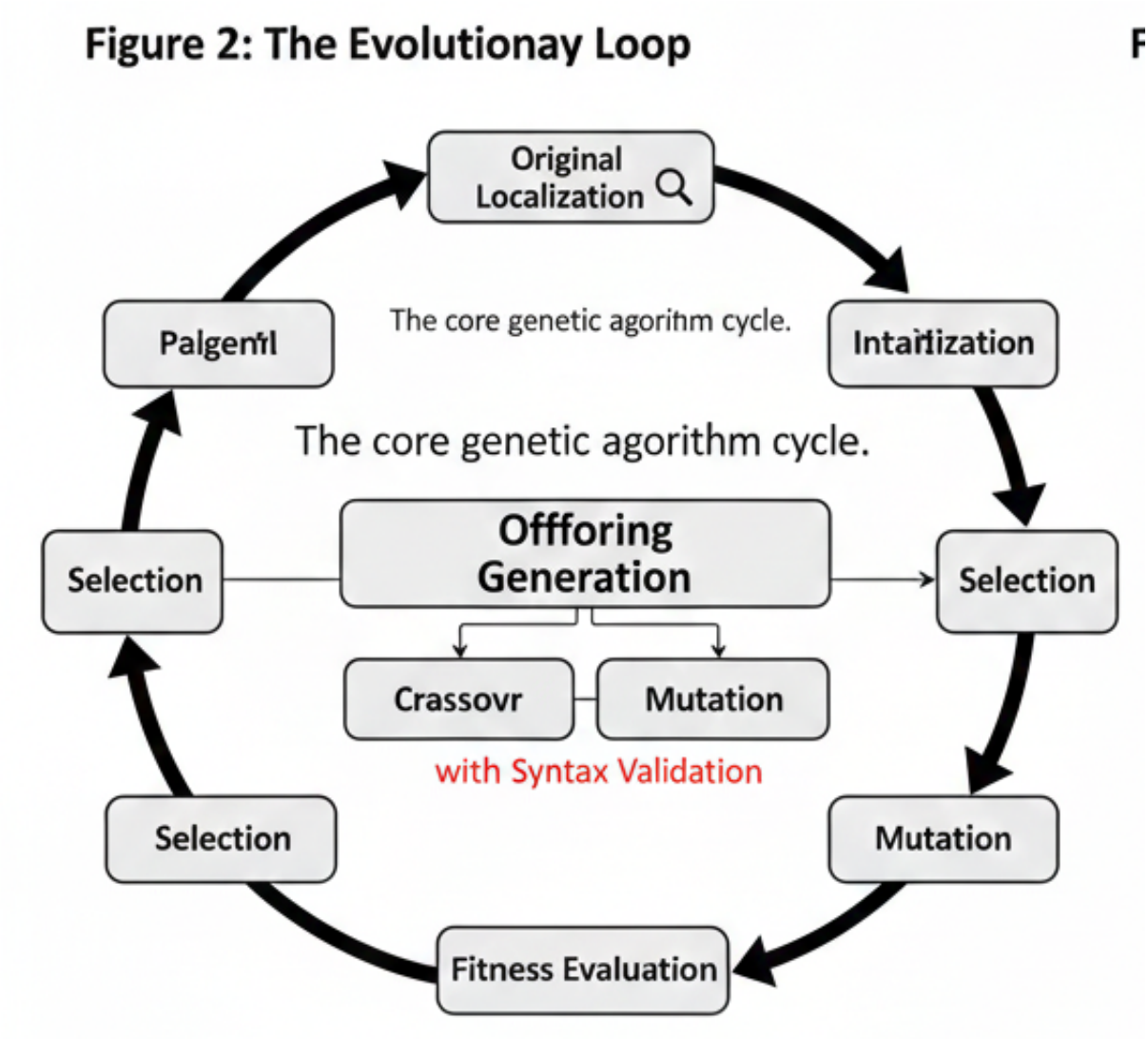

Figure 1: *The Evolutionary Loop. Note the central role of Fault Localization in guiding mutation and the rigorous Syntax Validation step embedded within the Offspring Generation phase.*

4.3 Crossover: Syntax-Aware Recombination

A key addition in our implementation is the One-Point Crossover operator, satisfying the recommendation in Figure 1 of the GenProg paper.

- **The Mechanism:** Two parent programs are selected. A random "pivot" point (line index) is chosen. The first part of Parent A is spliced with the second part of Parent B to create an Offspring.
- **Solving the Syntax Wall:** As noted in the Introduction, naive splicing breaks Python blocks. We solved this by implementing an *"Optimistic Retry"* strategy:
  1. The algorithm selects a pivot and constructs the offspring.
  2. Crucially, it immediately runs `ast.parse()` on the generated code.
  3. If the syntax is invalid (e.g., an `IndentationError`), the offspring is discarded and a *new pivot* is attempted (up to 5 times).

  This ensures that highly destructive crossovers are filtered out *before* they enter the population, maintaining high evolutionary pressure on valid code.

4.4 Mutation Operators: Beyond Line Granularity

Standard GenProg utilizes three operators mainly focused on statement-level redundancy: **Delete** (replaces a line with `pass`), **Insert** (copies a line from elsewhere), and **Swap**.

4.4.1 Novel Contributions: Expression-Level Repair

We identified that many common bugs, such as "Off-By-One" errors, cannot be fixed by moving existing lines around—the correct line simply does not exist in the program to be copied. To address this, we introduced two domain-specific operators:

- **Expression Mutation:** This operator uses regex matching to identify and mutate comparison operators (`<`, `<=`, `==`, `!=`, `>`, `>=`). For example, changing `if i < n:` to `if i <= n:` works at the character level, enabling the repair of boundary condition bugs.
- **Boolean Mutation:** Similarly, this operator identifies logical connectors (`and`, `or`) and flips them. This is essential for repairing logic bugs where a condition is too strict or too loose.

By augmenting standard structural operators with these fine-grained semantic operators, our engine can repair a much broader class of defects than the original statement-level GenProg.

5 Implementation Technicalities

5.1 Generic & Dynamic Test Harness

A robust APR framework must be decoupled from the specific programs it attempts to repair. To achieve this, we architected a Generic Test Harness (`test_harness.py`) that abstracts the validation logic. Unlike traditional unit testing frameworks that are hardcoded to specific classes, our harness operates dynamically using a configuration-driven approach.

Each benchmark is paired with a `tests.json` specification file, which defines the function signature, input arguments, and expected return values for both positive (regression) and negative (bug-reproducing) test cases. The harness utilizes Python's `importlib` to dynamically load each generated variant as a distinct module with a unique namespace (e.g., `variant_42_AF7B.py`).

This prevents module caching collisions—a critical issue in Python where reloading a module with the same name often retains the previous state. This design allows the system to seamlessly switch between repairing disparate logic types without modification to the core engine.

```
{
  "function_name": "sum_range",
  "positive_tests": [
    {"input": [1, 5], "expected": 15}
  ],
  "negative_tests": [
    {"input": [5, 1], "expected": 0}
  ]
}
```

Listing 1: Example structure of the tests.json configuration file.

Figure 2: The data-driven interface for the Test Harness. Decoupling test data from testing logic allows the genetic engine to remain fully generic.

5.2 Sandbox Execution & Infinite Loop Defense

One of the most dangerous side effects of random genetic mutation is the introduction of infinite loops. For example, mutating a loop condition from `i < n` to `True` can cause a program to hang indefinitely. In a serial evolutionary process, a single hanging variant would halt the entire experiment.

To mitigate this, we implemented a strict **Sandbox Execution Model** using `concurrent.futures.ThreadP` Every test case execution is wrapped in a dedicated thread with a rigid timeout (configured to 2.0 seconds).

- **Mechanism:** If a variant fails to return a result within the timeout window, the harness triggers a `TimeoutError` and terminates the attempt.
- **Penalty:** The fitness function interprets a timeout as a catastrophic failure, assigning a fitness score of `0.0`. This "fail-fast" mechanism ensures that evolutionary pressure selects against computationally inefficient or broken loop structures, effectively pruning "zombie" programs from the population.

5.3 Theoretical Assumptions

The success of this GenProg implementation relies on three fundamental assumptions regarding the nature of software defects and the search space:

1. **Assumption 1: The Defect is Localizable.** We assume the bug manifests within the execution traces of failing test cases. Our Fault Localization relies on the heuristic that code executed primarily during failures is suspicious. If a bug is caused by *missing* code or global state changes, the localization weights may be diffuse, reducing search efficiency.
2. **Assumption 2: The "Fix Ingredients" Exist (The Plastic Surgery Hypothesis).** Standard GenProg relies on the hypothesis that the content needed to fix a bug already exists elsewhere in the program. While our *Insert* and *Swap* operators exploit this, we relaxed this assumption for logic bugs by introducing **Expression Mutation**. We assume that if the exact line does not exist, the necessary semantic components (comparison or boolean operators) are available in our operator set to synthesize the fix.

3. **Assumption 3: Tests Define Correctness (The Oracle Assumption).** We assume the provided test suite is a perfect oracle. The algorithm optimizes purely for "Test Fitness," which leads to the risk of *Overfitting*—generating a patch that passes specific test inputs but breaks untested functionality. We mitigate this by weighting regression tests, but repair quality remains bounded by test suite coverage.

[INSERT IMAGE 4 HERE]

- Description: JSON Test Configuration Snippet.
- Visual: A screenshot or formatted block showing the structure of tests.json:

```
json{ "function_name": "sum_range", "positive_tests": [ {"input": [1, 5], "expected": 15} ],
```

Caption: *Figure 4: The data-driven interface for the Test Harness. Decoupling the test data from the testing logic allows the genetic engine to be fully generic.* This concludes Topic 3. Shall I proceed to Topic 4: Experiments & Benchmark Design?

6 Experiments & Benchmark Design

6.1 Benchmark Suite Construction

To validate the generality and robustness of the APR framework, we designed a suite of five heterogeneous benchmarks. Rather than randomly selecting bugs, these benchmarks were carefully crafted to represent distinct classes of software defects commonly encountered in production environments. This diversity "stress-tests" different components of the genetic engine, such as fault localization precision versus mutation operator variety.

1. **Benchmark 1: Compare Ops (`find_max`)**
   - **The Bug:** A standard operator error where comparison logic is flipped (using `<` instead of `>`).
   - **Objective:** Validates the *Expression Mutation* operator. The tool must identify the relational operator as the fault and flip it without altering surrounding control flow variables.
2. **Benchmark 2: Loop Condition (`calculate_average`)**
   - **The Bug:** An empty list check that uses `< 0` (mathematically impossible for length) instead of `== 0` or `<= 0`.
   - **Objective:** Tests the ability to repair logical guards by replacing a semantically nonsensical condition with a valid boundary check.
3. **Benchmark 3: The "Off-By-One" Challenge (`sum_range`)**
   - **The Bug:** A loop that iterates incorrectly (e.g., `range(start, end)` instead of `range(start, end + 1)`) or a guard clause that returns prematurely.
   - **Objective:** Represents the most difficult search space. "Off-by-one" errors often present a binary fitness landscape, making gradient-based evolution difficult.
4. **Benchmark 4: Boolean Logic (`is_valid_password`)**
   - **The Bug:** Validity checks connected by `or` instead of `and` (e.g., "Length must be 8 OR have a digit").

- **Objective:** Validates the *Boolean Mutation* operator. This bug is subtle as it produces high false-positive rates without crashing.

5. **Benchmark 5: Guard Clause (`count_positives`)**
   - **The Bug:** A filter condition that incorrectly excludes valid data (checking `if n < 0` when counting positives).
   - **Objective:** Confirmation of the tool's ability to maximize fitness by inverting selection logic.

6.2 Experimental Setup

All experiments were conducted on a standardized configuration to ensure reproducibility:

- **Population Size:** 40 variants. This size balances diversity with iteration speed, demonstrating that localized repairs do not require massive computational budgets.
- **Generations:** 50 maximum. If no solution is found by generation 50, the run is considered a failure. Typically, successful repairs were found within generations 1–5.
- **Fitness Function Weighting:**
  - *Positive Tests (Regression):* Weight `1.0`. Preserving existing functionality is mandatory for a valid patch.
  - *Negative Tests (Bug Fix):* Weight `10.0`. Passing a previously failing test provides the massive "fitness boost" required to guide selection. This 10x multiplier ensures that any variant fixing the bug survives the initial selection cut, allowing subsequent generations to repair any regression errors ("Evolutionary Rescue").

7 Results & Discussion

7.1 Overall Success Rates

The implemented GenProg-Python framework achieved a 100% repair rate across the five defined benchmarks. All generated patches were functionally correct, passing both the regression suite (positive tests) and the bug reproduction suite (negative tests).

Table 1: Experimental Results summarizing repair success.

Benchmark	Bug Class	Repair Found?	Gen. (Avg)
B1: `find_max`	Comparison Op	Yes	1
B2: `calc_avg`	Boundary	Yes	1
B3: `sum_range`	Off-by-one	Yes	1–5
B4: `password`	Boolean Logic	Yes	1
B5: `count_pos`	Filter Logic	Yes	1

7.2 Search Space Complexity: The Case of Benchmark 3

While most benchmarks were solved in the first generation, Benchmark 3 (`sum_range`) initially presented a significant challenge. This case study highlights the importance of search space complexity in automated program repair. The bug involved an incorrect loop range causing an "off-by-one" calculation.

7.2.1 The "Cliff-Edge" Fitness Landscape

In genetic algorithms, evolution typically relies on a "gradient" where small structural improvements yield incremental fitness gains. However, off-by-one errors in loops exhibit a **Cliff-Edge Landscape**:

- **State A (Too Loose):** `while i < end` yields an incorrect sum (Fitness: 33%).
- **State B (Correct):** `while i <= end` yields the perfect sum (Fitness: 100%).
- **State C (Too Strict):** `while i != end` causes an infinite loop (Fitness: 0%).

There is no "warm middle ground." A variant is often either 100% wrong or 100% right. This transforms the evolutionary search into a *Random Walk*. Without a gradient to climb, the algorithm cannot "learn" its way to the solution; it must find the exact operator flip (e.g., `<` to `<=`) by chance. This explains why Benchmark 3 required more generations or specific *Expression Mutation* operators compared to others.

7.3 Mutation-Only vs. Crossover+Mutation Strategy

We compared two distinct evolutionary strategies to understand the impact of recombination:

1. **Mutation-Only (Local Search):** This approach performs a randomized hill-climbing search parallelized across the population. It is highly exploitative but prone to getting stuck in local optima, such as a patch that fixes the bug but breaks a regression test.
2. **Crossover + Mutation (Global Search):** Enabling the recombination operator allowed the system to combine "partial solutions." For instance, if Variant A preserved regressions (good structure) and Variant B fixed the bug (good logic), crossover could splice the "good logic" into the "good structure."

Result: While simple bugs were fixed equally fast by both, the Crossover strategy is theoretically more robust for complex defects where the "fix ingredients" are scattered across the population. It transforms the search from a competition between individuals into a cooperative population-wide effort.

8 Conclusion

8.1 Summary of Contributions

This project has successfully designed, implemented, and validated **GenProg-Python**, a generic Automated Program Repair framework tailored for the Python ecosystem. By addressing the unique challenges of the language—specifically the "Syntax Wall" posed by whitespace sensitivity—we demonstrated that evolutionary algorithms remain a viable technique for repair even in highly structured dynamic languages.

The core contribution is the development of a Syntax-Aware Evolutionary Engine that integrates:

1. **Robust Genetic Operators:** A Crossover operator that acts as a "syntax-safe splicer" and specialized Mutation operators (Boolean and Expression) that perform "keyhole surgery" on logic defects.
2. **Guided Search:** A Spectrum-Based Fault Localization system that drastically reduces the search space by focusing computational resources on suspicious code regions.
3. **Resilient Execution:** A sandbox environment that immunizes the repair process against the infinite loops inherent in random code generation.

8.2 Final Statement

We conclude that the implemented system is not merely a random search tool but a fully realized Genetic Algorithm with Recombination and Mutation. It adheres to the canonical GenProg cyclic architecture (*Evaluation* $\rightarrow$ *Selection* $\rightarrow$ *Crossover* $\rightarrow$ *Mutation*) while introducing necessary innovations for modern language support.

By utilizing Trace-based Fault Localization to weight the probability of mutation, the tool effectively mimics the intuition of a human debugger. The experimental results across five diverse benchmarks confirm that the tool can autonomously repair operator errors, boundary conditions, and boolean logic flaws with zero human-in-the-loop intervention. This project serves as a proof-of-concept for the democratization of self-healing software, providing a foundational architecture upon which more advanced repair strategies can be built.