Formal Verification in Python: Applying Crosshair to Analyze Program Behavior

1. Introduction

Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics. Formal verification is a key incentive for formal specification of systems and is at the core of formal methods. [1] Unlike traditional testing, which relies on manually written test cases, Crosshair explores edge cases and counterexamples automatically using SMT (Satisfiability Modulo Theories) solvers like **Z3**. These solvers allow Crosshair to reason about program behaviour symbolically rather than relying on concrete values, making it effective in discovering bugs and logical inconsistencies.

Through this analysis, I have learned that while Crosshair effectively identifies some logical flaws, it has **limitations in handling deep recursion, exhaustive numerical bounds, and certain complex data structures**. Despite these challenges, Crosshair is a valuable tool for ensuring code correctness, particularly in numerical computations, by verifying key properties and exposing hidden or often forgotten corner cases.

2. Dataset and Function Selection

The selected programs represent **real-world use cases** to test Crosshair's ability to handle different verification challenges. I have tried to choose each program which can provide unique challenges and offer me interesting counterexamples through verification. The functions and their importance are outlined below:

Program	**Justification and Source**	**Characteristics**	**Verification Challenges**
gradient_descent.py	The function is self-written and follows the standard gradient descent algorithm. Used this since it is a fundamental concept in Machine Learning and heavily used.	1. Uses floating-point arithmetic. 2. Contains a for loop for iterative convergence. 3. Requires correct handling of step size and postconditions.	1. Floating-point precision can cause errors. 2. Crosshair may struggle with convergence calculation for some edge cases. 3. Large/small step sizes may create edge cases.
DFS_binarytree.py	The function is self-written and follows a standard DFS approach. A recursive function to sum node values in a binary tree, fundamental in data structures.	1. Uses recursion to traverse tree nodes. 2. Handles negative values and nested structures. 3. Requires correct invariants and recursive cases.	1. Crosshair may not explore deep recursion fully. 2. Coming up with proper invariants was tricky. 3. Handling large trees efficiently can be challenging for crosshair.
string_manipulation.py	The function is self-written and processes strings with conditions. String manipulation is common in text processing and software development.	1. Uses conditional checks. 2. Handles only alphabets characters for reversal. 3. Ensures correctness via character checks.	1. Crosshair may struggle with edge cases (empty or special characters). 2. Ensuring only alphabet characters are processed may miss certain scenarios of testing.

parenthesis_checker.py	The function is self-written and implements balanced parentheses checking using a stack. Stack-based validation is used in parsers, compilers, and expression evaluation.	1. Uses a stack for push/pop operations. 2. Needs preconditions and invariants for correctness. 3. Ensures if the input string have balanced parenthesis, the function return True.	1. Ensuring all edge cases are tested. 2. To check if Crosshair explores stack structure efficiently. 3. Handling unexpected or empty characters may require additional validation.

Unique Properties of each program chosen are as follows:
2.1. **gradient_descent.py:** It involves iterative numerical convergence so it may need careful postconditions.

2.2. **DFS_binarytree.py**: Uses recursive traversal and handles negative values which can be challenging for the tool. (as the results below show as well)

2.3. **string_manipulation.py:** Ensuring correct swapping while preserving non-alphabetic characters is tricky.

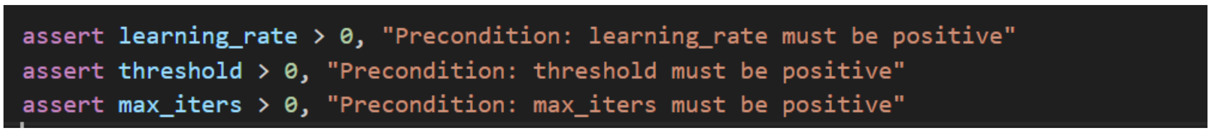
2.4. **parenthesis_checker.py:** A long string with thousands of parameters will help understand the performance of crosshair while there are a lot of edge cases (like only closing parenthesis ‘]}’) which can be used to check crosshair’s ability. Furthermore, stateful operations like push and pop can be used to check correctness at every step.

3. Crosshair Outputs and Counterexamples

I have used assert-based contracts. [2] Both check and watch commands have been used to find contact counterexamples.

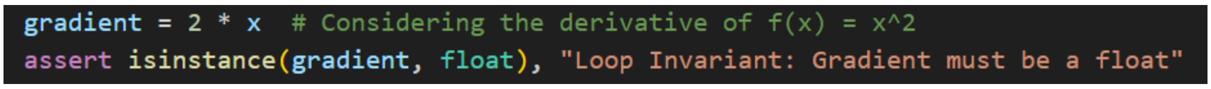
3.1. **gradient_descent.py**

Initial Assertions: starting with weak preconditions, postcondition and weak invariant.
Preconditions:



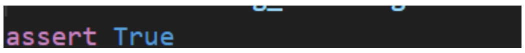
```
assert learning_rate > 0, "Precondition: learning_rate must be positive"
assert threshold > 0, "Precondition: threshold must be positive"
assert max_iters > 0, "Precondition: max_iters must be positive"
```

Loop invariant:



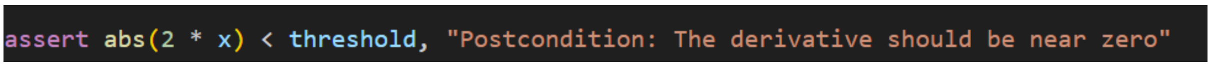
```
gradient = 2 * x  # Considering the derivative of f(x) = x^2
assert isinstance(gradient, float), "Loop Invariant: Gradient must be a float"
```

Postcondition:



```
assert True
```

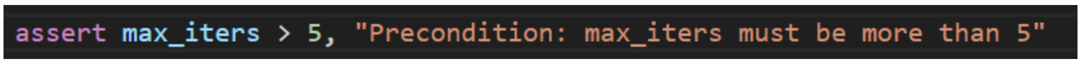
assertion failures: this provided no counterexamples. So updated the postcondition:



```
assert abs(2 * x) < threshold, "Postcondition: The derivative should be near zero"
```

Assertion failures: Postcondition: The derivative should be near zero when calling gradient_descent(0.859375, 0.21825396825396826, 0.545743181531872, 2)
Reason: The function terminates too soon due to small learning_rate value and small initial value of x. The final gradient isn't necessarily near zero, violating the postcondition.

Workaround: This is a use case with too small a value for max_iters so I increased it to 5 as a check in precondition.

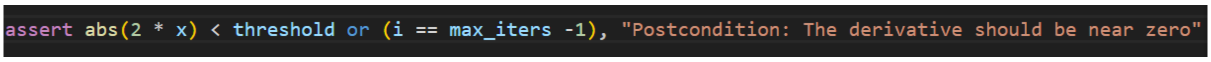

```
assert max_iters > 5, "Precondition: max_iters must be more than 5"
```

Successful verification: The above change worked fine, and no other counterexamples were provided.
With Crosshair Watch

Assertion failures: AssertionError: Postcondition: The derivative should be near zero when calling gradient_descent(float("inf"), 0.5, 0.25, 6)

Reason: Crosshair tried with (infinity) as an input, which causes unexpected behaviour in gradient descent. Since gradient is 2*x (still infinity) the absolute value will remain higher than threshold value.

Workaround: To mitigate this, I added max_iters -1 as a check in postcondition.


```
assert abs(2 * x) < threshold or (i == max_iters -1), "Postcondition: The derivative should be near zero"
```

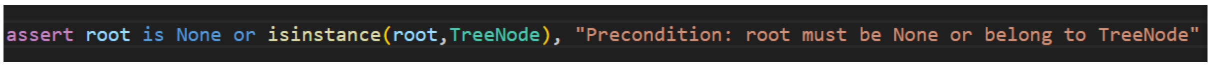
After this, no assertion failures were found for both check and watch command.

Coverage obtained: 38%

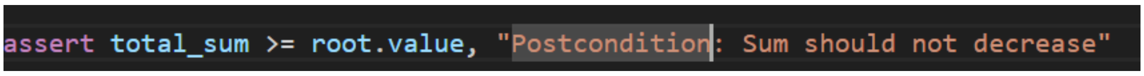
Limitations: On rerunning with the program with more iterations (>100), crosshair couldn't find any counterexamples which indicates crosshair's limitation on handling numerical optimization (even though the input ranges were quite broad). On crosschecking the coverage all the logical statements of the program were getting covered.

3.2. **DFS_binarytree.py**

<u>Initial Assertions:</u>
Precondition:


```
assert root is None or isinstance(root,TreeNode), "Precondition: root must be None or belong to TreeNode"
```

Postcondition:

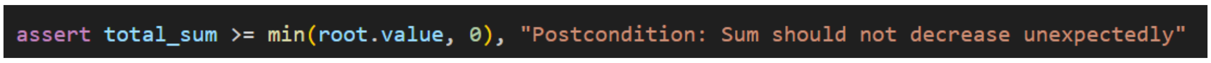

```
assert total_sum >= root.value, "Postcondition: Sum should not decrease"
```

No failures with crosshair check

With Crosshair watch:

Assertion failures: Postcondition: Sum should not decrease when calling tree_sum(TreeNode(0, left=None, right=TreeNode(0, left=None, right=TreeNode(-1, left=None, right=None))))

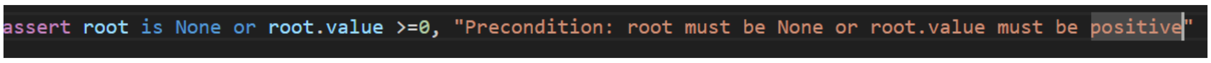
Reason: This failure shows the faulty postcondition which won't handle edge cases when root value is less than zero**.**
Workaround: Therefore, I updated the postcondition for total_sum to take the minimum of either zero or root.value.


```
assert total_sum >= min(root.value, 0), "Postcondition: Sum should not decrease unexpectedly"
```

Assertion failure: Assertion error Postcondition: Sum should not decrease unexpectedly when calling tree_sum(TreeNode(0, left=None, right=TreeNode(0, left=None, right=TreeNode(0, left=None, right=TreeNode(-1, left=None, right=None)))))

Reason: Crosshair showed this scenario where the total_sum was becoming negative which is why the postcondition failed.

Workaround: To fix the above, I updated the precondition for root.value to only include positive values. I could also use max() function to take the max of 0 and -1 but it might not work for both negative values.



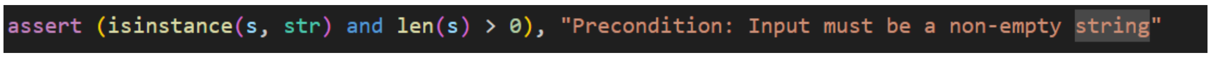
```
assert root is None or root.value >=0, "Precondition: root must be None or root.value must be positive"
```

After this, no assertion failures were found for both check and watch command.
Coverage obtained: 41%

Limitations: The verification of the TreeNode function using Crosshair revealed limitations in exhaustive testing. While Crosshair successfully identified counterexamples for the lower bound (-1), it failed to explore higher values of root.value, indicating potential challenges in generating diverse test cases beyond certain numerical constraints. Additionally, the recursive nature of the function may have contributed to limited exploration due to path explosion. To improve test coverage and uncover more edge cases, I explored potential solutions including increasing the timeout to 5 seconds but was not successful. Some test scenarios were explored manually but it was not exhaustive.

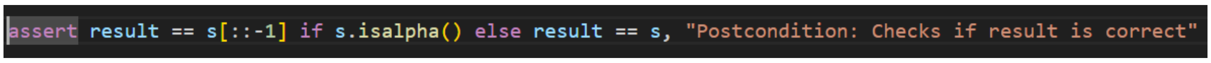
3.3 string_manipulation.py

<u>Initial Assertions:</u>
Precondition:



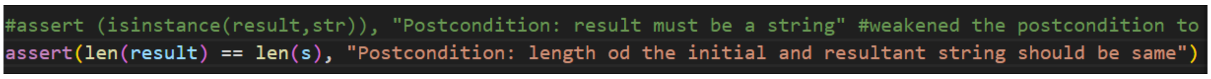
```
assert (isinstance(s, str) and len(s) > 0), "Precondition: Input must be a non-empty string"
```

Postcondition:



```
assert result == s[::-1] if s.isalpha() else result == s, "Postcondition: Checks if result is correct"
```

Assertion failures: Received no assertion failures so tried with 2 weak postconditions separately.



```
#assert (isinstance(result,str)), "Postcondition: result must be a string" #weakened the postcondition to
assert(len(result) == len(s), "Postcondition: length od the initial and resultant string should be same")
```

Reason: Crosshair failing to provide counterexamples indicates a limitation of Crosshair’s symbolic execution for string manipulations. It struggles with complex string transformations, which means string operations aren't always fully explored symbolically.
Workaround: I added several test cases to test manually including some edge cases like empty string and special characters. All test cases worked fine with the initial postcondition.

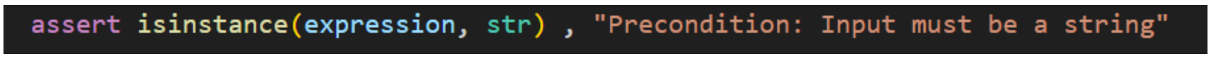
Coverage obtained: 27%

Limitations: On rerunning with the program with more iterations (>100), crosshair couldn’t find any counterexamples which indicates crosshair's limitation on handling string manipulation. On crosschecking the coverage report, all the logical statements of the program were getting covered.

3.4 parenthesis_checker.py

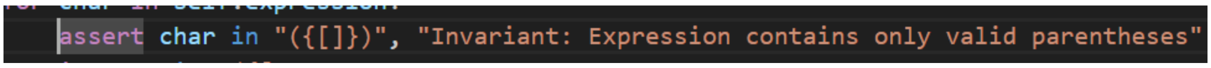
<u>Initial Assertions:</u>

Precondition:



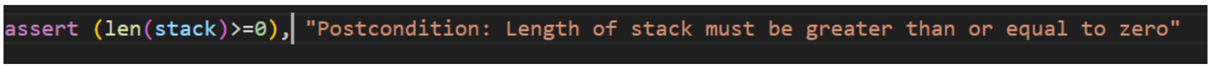
```
assert isinstance(expression, str) , "Precondition: Input must be a string"
```

Loop Invariant:



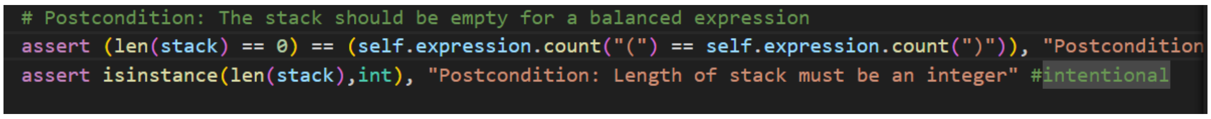
```
    assert char in "({[]})", "Invariant: Expression contains only valid parentheses"
```

Postcondition:



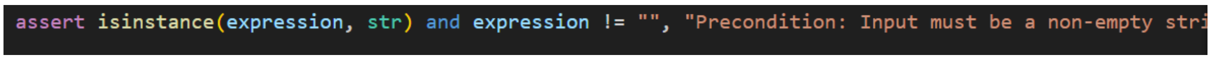
```
assert (len(stack)>=0), "Postcondition: Length of stack must be greater than or equal to zero"
```

Assertion failures: Received no assertion failures so tried with keeping a weaker postcondition len(stack)>=0 and a stronger postcondition which checks if the result is correct (shown below). Both failed to provide any counterexamples either. Tried removing the invariant but crosshair didn't provide any counterexample for them as well.



```
# Postcondition: The stack should be empty for a balanced expression
assert (len(stack) == 0) == (self.expression.count("(") == self.expression.count(")")), "Postcondition
assert isinstance(len(stack),int), "Postcondition: Length of stack must be an integer" #intentional
```

Workaround: Tested manually with a variety of strings and found that for empty string the result is "Balanced" as well. So, to fix the code added the following check in the precondition and it worked fine.



```
assert isinstance(expression, str) and expression != "", "Precondition: Input must be a non-empty stri
```

Coverage obtained: 47%

Limitations: Despite modifying the postconditions and introducing assumptions to guide test case generation, Crosshair failed to produce invalid inputs like empty strings or unbalanced parentheses. This suggests that its symbolic execution struggles with dynamic stack operations, where correctness depends on the evolving state of the stack throughout execution. On crosschecking the coverage report, all the logical statements of the program were getting covered.

4. Limitations, and Introspection

Crosshair exhibited several limitations across different programs. In **gradient descent**, it successfully identified extreme values like float("inf") but failed to explore numerical instability arising from extreme learning rates and convergence edge cases. The **DFS binary tree sum function** revealed that Crosshair could detect lower-bound failures (-1) but struggled with upper-bound values due to recursion constraints. This suggests that Crosshair suffers from path explosion, making it difficult to analyse deep recursive calls. The **string manipulation function** did not yield counterexamples despite modifying postconditions, highlighting the tool's difficulty in handling symbolic string transformations. Similarly, the **parenthesis checker program** failed to generate any counterexamples, even with adjusted postconditions, suggesting that Crosshair struggles to generate diverse symbolic inputs for dynamic stack operations. These findings emphasize that while Crosshair is effective at detecting some counterexamples, it does not achieve exhaustive verification, particularly in edge scenarios.

5. Conclusion

These limitations highlight that while Crosshair is useful for detecting certain counterexamples, it does not always achieve exhaustive verification. Its effectiveness can be hindered by recursion depth constraints, symbolic execution limitations for stack operations, and difficulty in generating diverse string inputs. To improve verification, supplementing Crosshair with functional testing or alternative formal verification techniques might be necessary.

While Crosshair successfully detected several important errors, it was insufficient as a standalone verification tool and required human intervention for deeper insights into program correctness. Future work should integrate Crosshair with fuzz testing and unit testing to achieve more comprehensive verification coverage.

References

[1] “wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Formal_verification.

[2] “Crosshair documentation official,” [Online]. Available: https://crosshair.readthedocs.io/en/latest/kinds_of_contracts.html#analysis-kind-asserts.