Pandas DataFrame: OSS Validation Study

Cole M. Thornton
Purdue University
Long Beach, California, USA
thornt45@purdue.edu

C.P. Pacaba
Purdue University
Seattle, Washington, USA
cpacaba@purdue.edu

Sriram Ganesh
Purdue University
Seattle, Washington, USA
Ganesh24@purdue.edu

ABSTRACT

INTRODUCTION:

Pandas is an essential data analysis package that offers data scientists and data analysts the ability to manipulate and process data within the Python ecosystem. DataFrame, one of the most popular classes within the Pandas library, offers users the ability to control and analyze data within a two-dimensional array.

STATE OF PRACTICE:

DataFrame is currently being used as the primary data structure class within the Pandas library. It is primarily used as a statistical data analysis tool, facilitating users' ability to manipulate data using structured data sets. It provides users with similar functionality to what is found within competing data analysis software such as R, MATLAB, STATA, or SAS.

CONTRIBUTION:

This project provides the Pandas userbase with an in-depth assessment of the DataFrame class. The validation study conducted aims to instill confidence among users regarding its robustness.

METHOD:

Robustness of DataFrame was quantified by applying three unique advanced software validation techniques: combinatorial testing, fuzz testing, and metamorphic testing. Program tests were created for all three techniques and were used to test DataFrame. A total of 12,365 test cases were run, where no errors were observed. This suggests that the DataFrame class is robust with a high level of confidence.

1 INTRODUCTION

1.1 Overview

Open-source projects offer transparency and opportunities to collaborate with large groups of people within a community-driven development environment. As a result, many unique and helpful programs have been released to the public for free. However, this accessibility is not without tradeoffs. Many open-source projects have been launched without undergoing rigorous software validation, primarily due to the unique challenges it poses and the associated costs. Lack of software validation puts many programs at risk of performing in unexpected ways that deviate from their intended functionality. Identifying these unexpected program behaviors or bugs is not always obvious, and they can often remain hidden in many popular open-source libraries.

1.2 Validation Target: Pandas DataFrame

Pandas, an open-source data manipulation library for Python, is widely recognized for its user-friendly data structures and functions. Within the Pandas library, DataFrame is a popular class that provides users with the ability to manipulate, clean, and transform data. Used across diverse domains such as finance, economics, neuroscience, web analytics, and data science, DataFrame simplifies complex data manipulation tasks. In finance and economics, it aids in extracting insights for strategic decision-making, while in neuroscience, it streamlines the exploration of brain activity data. Web analytics professionals leverage Pandas DataFrame for in-depth user behavior analysis, and in data science, it is an essential tool for preparing and transforming datasets for analysis and machine learning.

1.3 Problem

Due to the open-source nature of the library, DataFrame may contain undiscovered software defects that could affect program behavior. Despite its popularity, the extent of rigorous software validation completed for the DataFrame class remains unclear to its userbase. Through comprehensive testing, this paper aims to reach consensus on the reliability and robustness of DataFrame, aligning it with the high standards expected in critical data manipulation tasks across various industries.

1.4 Planned Approach

This paper will evaluate the robustness of DataFrame by applying advanced software validation techniques to identify potential software defects. In this paper, each team member was responsible for one of three validation techniques: metamorphic testing, fuzz testing, or combinatorial testing. These techniques were selected to cover a wide range of potential issues, including defects that may fall outside traditional software validation testing. Before any code was written, each member took time to research each method and build familiarity with DataFrame. Program tests were designed for each respective technique and used to run multiple test cases. The output for each test case was

reviewed by the team in order quantify a level of confidence in the robustness of the DataFrame class.

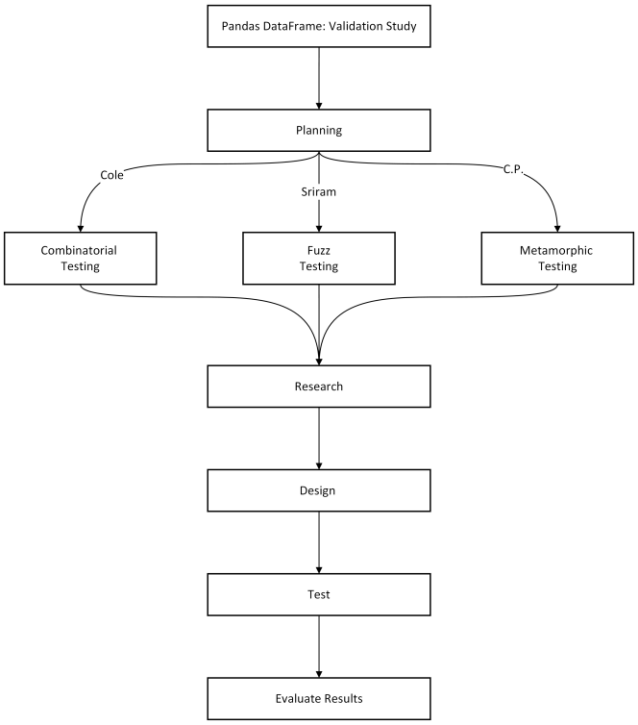

Figure 1. Flow chart depicting a high-level project overview. Each team member researched and applied an advanced form of software validation in an attempt to identify flaws within DataFrame.

2 MOTIVATION

2.1 Motivation for Validating Pandas

Pandas is a fundamental tool for data analysis in Python and has revolutionized the way people can manipulate and process data. Pandas is a central part of DataFrame and is a powerful data structure for various data manipulation tasks. However, due to the open-source nature of Pandas, which fosters community collaboration and innovation, there are also potential challenges related to code quality and reliability. The reliability and robustness of Pandas is paramount. Pandas is used widespread across domains such as finance, economics, data science, and more, it is important to ensure the integrity of its core components. For context, each year Stack Overflow conducts a developer survey. Between 2013 and 2022, in Python alone, there was a rise in 21.9% to 48.1% for developers having done extensive development work in Python. [12] Focusing on Pandas, the developer community is 5X larger than the Spark/ Hadoop communities which are also used for data analysis. Looking at the figure below, between 2019 and 2022, there was a rise in Pandas users from 12.7% to 25.0% for all developers in the survey conducted by stack overflow. Looking at Python users alone, around 52% to 55% of all Python users use pandas. [12]

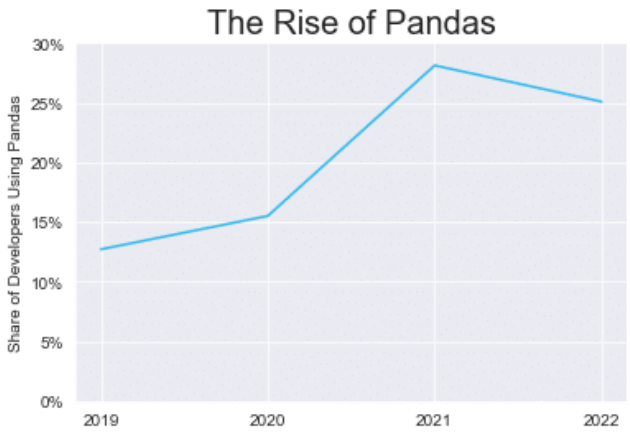

Figure 2: This Figure shows the significance of Pandas in the developer community. As the use of Python grows, Pandas is also growing with more and more developers utilizing it.

2.2 Motivation for Validating Pandas DataFrame

Any vulnerabilities or defects within Pandas, particularly within the DataFrame class, could have far-reaching implications, potentially leading to erroneous analyses, flawed decision-making, and compromised data integrity. DataFrame itself is important to validate because they are faster, easier to use, and more powerful than tables or spreadsheets when it comes to analytics on big data. They are an integral part of NumPy and Pandas. The motivation behind this validation project stems from the recognition of Pandas' indispensable role in data manipulation tasks and the imperative to ensure its reliability and effectiveness. By subjecting the DataFrame class to rigorous testing methodologies, including combinatorial testing, fuzz testing, and metamorphic testing, we aim to uncover latent defects and vulnerabilities that may undermine its functionality, performance, stability, or security. Through systematic validation, we seek to boost confidence in DataFrame's robustness, thereby enhancing its utility and trustworthiness for data scientists, analysts, and organizations relying on Pandas for critical data manipulation tasks. Any defects or predicaments we come across will be communicated to the Pandas community to strengthen the open-source program.

This validation initiative aligns with the idea of open-source development, emphasizing the commitment to transparency, quality, and continuous improvement. By identifying and addressing potential shortcomings in Pandas, we contribute to its ongoing evolution and strive to maintain its status as a highly reliable tool in data manipulation libraries. Ultimately, our motivation lies in strengthening Pandas as a reliable and indispensable tool for data practitioners, empowering them to extract meaningful insights and drive innovation with confidence.

3 BACKGROUND AND RELATED WORK

Software testing is a critical process when developing software of high quality. As stated by Edger Dijkstra, “testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence [1]. Over the years, various methods of software testing have been developed to catch defects early on within the development process. In this paper, we will introduce and apply some advanced techniques used for software validation.

3.1 Combinatorial Testing Background

Combinatorial testing. One method of testing we may approach is combinatorial testing, a technique that involves testing combinations of inputs to identify errors that are caused from the interactions between parameters [2]. One example of the most basic form of combinatorial testing is pairwise testing. This type of testing focuses on testing all possible pairs of input parameters within a system. While not complete, this method is “useful at checking for simple, potentially problematic interactions with relatively few tests” [2]. However, this form of testing will only catch errors for interactions that involve two input parameters.

	Parameters									
Test	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

Figure 3. Three-way covering array for 10 parameters with two values each. Any three columns, selected in any order, contain all eight possible values of three parameters: 000, 001, 010, 011, 100, 101, 110, 111. [2]

Testing higher-degree interactions with combinatorial testing can be achieved through the use of “a covering array, a mathematical object that covers all t-way parameters at least once” [2]. This advanced form of combinatorial testing extends the testing coverage to catch software errors that involve complex reactions between three or more variables, without having to test all cases exhaustively. While these methods are effective at generating test suites, it is equally important to validate the combinatorial models being used and if the tests being generated are of sufficient quality. Currently, there are validation techniques to check constrained combinatorial models for faults, such as test suites that are incomplete or testing tools that generate incorrect test cases [3].

3.2 Fuzzing Testing Background

Fuzz testing. Another related validation method we may pursue is fuzzing, a technique which involves injecting invalid or random inputs into a system to detect vulnerabilities through the use of fuzzers [4]. Traditional fuzzing techniques are typically composed of four main stages: test-case generation, program execution, runtime state monitoring, and analysis of crashes [4]. Traditional fuzzing techniques work by generating a large set of inputs and injecting the inputs into the target software. As the software runs, the fuzzer monitors the program and automatically records information related to failures or unexpected behavior within the program. This information is fed back into the test-case generation stage to generate new test cases on the next iteration. The figure below shows the basics of fuzzing.

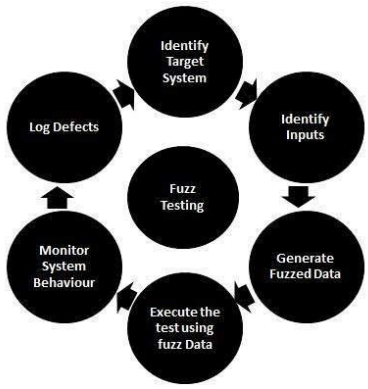

Figure 4. The basics of fuzzing include identifying the system, inputs to generate fuzzed data, executing the test and monitors the system to perform further iterations.

Despite this technique's effectiveness, traditional fuzz testing requires careful thought on its integration to maximize its effectiveness in identifying vulnerabilities. The introduction of static and dynamic analysis has promoted the development of fuzzing techniques and made the traditional fuzzing process more intelligent [4]. Another potential improvement to fuzzing methods includes the integration of machine learning algorithms. Studies have shown that machine learning fuzzers improved code test coverage by an average of 17.3% [4]. This is due to targeted testing, historical analysis of past data sets/tests, generation of realist inputs, and varying testing complexity.

3.3 Metamorphic Testing Background

Metamorphic Testing. We may supplement our validation campaign with metamorphic testing. It can be difficult and time-consuming to generate numerous test cases, without knowing what the expected output should be. Metamorphic testing addresses this problem by focusing on the relationship between a test case and its respective test result, referred to as an input-

output pair [5]. With this method, new test cases are generated based on the input-output pairs of previous, successful test cases [5]. This is achieved by applying a transformation to the original input and verifying that the transformed output satisfies some relation to the original output. If relations do not hold between outputs, then there is high likelihood that there is a flaw or bug in that portion of the software.

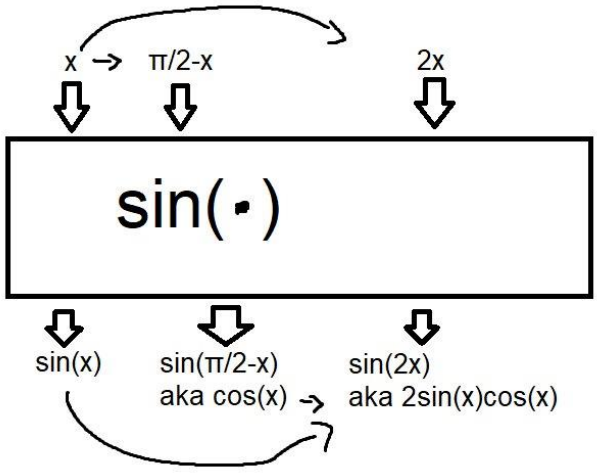

Figure 5. A basic example of a metamorphic test on an implementation of sin(). We presume $\sin(x)$ and $\sin\left(\frac{\pi}{2}-x\right)$ to be correct, and we use it to validate that an execution of $\sin(2x)$ is correct. Note that we need to use the function we are testing multiple times (here, 3 times) to test it metamorphically.

3.4 Miscellaneous Testing Techniques

Google OSS-Fuzz. In 2016, Google launched an open-source project called OSS-Fuzz in response to a security vulnerability within the Transport Layer Security (TLS) protocol, [6]. The goal of this project is to uncover software errors using fuzz testing strategies. Developers can use OSS-Fuzz to generate randomized inputs to intentionally cause unexpected behaviors or crashes within their programs. From this testing, developers are able to identify software defects that typically fall outside project requirements. These types of bugs are typically what lead to security vulnerabilities or reliability problems [7]. Our validation campaign will likely leverage Google OSS-Fuzz as an additional form of vulnerability testing with the hope of implementing some more advanced techniques within the domain of fuzz testing.

AllPairs OSS. AllPairs is an open-source test tool that is capable of generating a set of test cases using the "pairwise combinations" method [8]. This tool works by taking input parameters fed by the user and outputting pairwise combinations of variables through the use of an algorithm. From this output, users are able to reduce the number of test combinations of variables to a lesser set that covers most test cases [8]. Our combinatorial testing approach will likely be similar to AllPairs with some added functionality to cover higher-degree interactions that involve three or more parameters. This would likely be accomplished through the use of a cover array algorithm.

3.5 Current Validation of Pandas

In addition to exploring various testing methods that could be used to validate Pandas, the team explored the Pandas DataFrame Developer testing and issues pages located at: https://github.com/pandas-dev/pandas/tree/main/pandas/tests/frame. The Pandas developers ensure reliability through unit testing, where individual components are scrutinized to ensure accurate functionality. Integration testing validates interactions between Pandas modules, while regression testing safeguards against new updates disrupting existing features. Employing continuous integration tools like Travis CI or GitHub Actions, the developer team automates testing processes to catch errors early and maintain code integrity throughout development.

Specifically, relating to DataFrame, recent DataFrame test files in the Pandas project include unit tests to validate individual methods (test_add_prefix_suffix.py), integration tests to verify interactions between operations (test_combine.py), regression tests to ensure new changes don't introduce bugs (test_drop_duplicates.py), and continuous integration practices to maintain code integrity across development (test_fillna.py). The team did not notice much fuzzing, combinatorial, and metamorphic testing, and chose to explore those testing techniques to enhance the validation process of DataFrame..

4 REQUIREMENTS

A notable requirement for this project was to apply an advanced method of software validation. There are many different types of software validation techniques that have been popularized in the field. However, each technique has its strengths and weaknesses. It is possible that DataFrame is weak to certain types of testing scenarios while appearing to be robust in other areas, hence the need to cover a wide breadth of advanced testing strategies. This drove the need to use multiple testing strategies during this validation campaign. During the scope of this exercise, the team used combinatorial testing, metamorphic testing, and fuzz testing to probe DataFrame from multiple angles in an effort to uncover unexpected program behavior. In addition, the team also aimed to keep the testing simple and easy to replicate, facilitating others to able to use similar approaches in other areas outside this paper.

5 TEAM MANAGEMENT STRUCTURE

The team for this effort includes three graduate students where each team member held similar roles during the scope of the project. Each team member was responsible for one validation technique, and led the design, implementation, and analysis tasks for their assigned technique. Team members were required to meet on a weekly basis and encouraged to provide constructive feedback with respect to project deliverables, goals, and tasks. All

project decisions were made with group consensus, either verbally during the weekly meetings or via text in an online group chat.

6 TOOLS

Execution: Before we can run tests on Pandas, we need Pandas installed and all files and programs required by Pandas installed. For Pandas itself, we used Pandas version 2.2.2. Pandas runs on Python, which we used Python version 3.10.11 for. Python also comes with Pip, a program that we used to help facilitate installation of Pandas (and its necessary libraries), and we used pip version 23.3.2. Each member may have done some programming on other versions of this software, such as older or unreleased versions of Python or Pandas, however we re-executed all tests on the freshest possible environment to eliminate any deviation and to account for any new releases of Pandas and its requirements.

Creation and modification of tests: To create and modify tests, we use development environments, or IDEs. IDEs are used to simplify parts of the development process, such as variable name autofilling. Given that the purpose of IDEs is to streamline the editing experience and make it better for the specific developer using it, each member used their IDE of preference. We believe this shouldn't affect the results of the tests, as they only make the experience of editing better, and shouldn't affect the execution of the code. C.P. used PyCharm version 2023.3.5, Cole used VScode version 1.88.1, and Sriram used Spyder Version 3.3.6.

Collaboration: To collaborate on tests and to ensure everyone's test results matches on everyone's computers, we use version control software, namely GitHub. This allows us to have one "universal codebase" to act both as a backup and as a public codebase for other people to look at. C.P. also uses GitHub Desktop version 3.3.9 to make using GitHub easier.

7 DESIGN

7.1 METAMORPHIC

7.1.1 Metamorphic Overview

One form of testing we employed was Metamorphic testing. Metamorphic testing is a form of testing where you presume one result, f(x), is correct, and use it to validate that a second result, f(x'), is correct. We decided to validate the construction of the DataFrame from raw data, and specifically test that manipulating a DataFrame is equivalent to manipulating the raw data. For example, if you measured a racecar's top speed in many scenarios (varying weather, climate, ice, road and tire type), and you measured it in meters per second but you wanted to release to the American market, you might want to convert the data to mph. If you have a lot of data, this could be a lot of multiplication – however, if the data is in a DataFrame, doing one multiplication on the DataFrame should be equivalent to that same multiplication on every element of the DataFrame. This is the specific DataFrame feature that we chose to validate metamorphically.

7.2 FUZZING

7.2.1 Fuzzing Overview

Fuzzing is the technique of injecting random inputs to detect vulnerabilities. It can be combined with techniques like machine learning to improve test coverage [4]. Fuzzing testing can be directly applied to the DataFrame class within the Pandas library to assess its resilience and reliability in handling various data scenarios. This involves subjecting DataFrame instances to randomized or unexpected data inputs, such as data of different types, sizes, and formats, to evaluatemetr how well the class manages edge cases and unexpected data structures. Furthermore, parameter fuzzing can be employed to explore the behavior of DataFrame methods under extreme parameter values or invalid combinations, ensuring robustness in parameter handling. Integration fuzzing can scrutinize DataFrame's interactions with other Pandas functions and external libraries, while performance fuzzing can stress-test DataFrame operations with large datasets to identify potential inefficiencies. Finally, security fuzzing can be utilized to uncover and mitigate potential security vulnerabilities within the DataFrame class, safeguarding against malicious inputs or exploitation attempts. By applying fuzzing testing techniques to the DataFrame class, it can enhance its stability, efficiency, and security, strengthening the reliability of data manipulation tasks performed with Pandas.

7.3 COMBINATORIAL

7.3.1 Combinatorial Overview

Lastly, the final validation technique explored was combinatorial testing. Combinatorial testing is a software validation method that involves testing combinations of inputs to identify errors that are caused from the interactions between parameters [2]. Combinatorial testing was used to determine if there was an unusual combination of inputs within the DataFrame class that would result in unexpected program behavior. This can be accomplished by passing different combinations of inputs to the DataFrame class and monitoring the respective output. The design framework for combinatorial testing usually includes a type of test covering array where each test case within the array represents a specific combination of inputs. This covering array can then be used to feed test cases into a test suite to undergo software testing.

8 METHOD

8.1 METAMORPHIC

8.1.1 Metamorphic Methodology

To implement metamorphic testing, we split the full metamorphic test into two sections. First, we generate many "Test DataFrames". The goal is to run metamorphic testing on every possible DataFrame that could exist. In practice, there are infinitely many possible DataFrames that could exist, so having the test DataFrames being equal to all DataFrames – even all small DataFrames – is impossible. Therefore, we only hope that the set of Test DataFrames is a representative sample of all DataFrames, even if it isn't a complete sample. Second, we then run each type of metamorphic test (addition, multiplication, adding columns, etc.) on each of these DataFrames, and check for any errors. If no errors are found, then report how many tests were completed and some misc. data, such as how many test DataFrames there are.

In fact, due to the nature of metamorphic testing, both of these are very similar tasks: When a metamorphic test executes, it uses a DataFrame df1, creates a different DataFrame df2, and compares them together. We can reuse that new DataFrame df2: For the first section, to generate the many test DataFrames, we instead have a small number of root DataFrames. Then, we apply all of the metamorphic testing functions to these root DataFrames, saving each instance of df2 after each metamorphic test. This way, we reuse the metamorphic tests to also assist with generating the test DataFrames. The second section is done as stated above: We apply all of the metamorphic testing functions to the test DataFrames, checking for assertion failures and counting assertions.

8.1.2 Metamorphic Implementation Example

Consider a simple test shown in the figure below. This test validates that the absolute value of a DataFrame is equivalent to applying the absolute value to all of its elements.

```
def test_absolute_value(dfb: DFBuilder.DFBuilder):
    ret = [dfb.deepcopy()]
    if not dfb.check_types((int, float, complex)):
        return ret

    df = dfb.to_df()
    df = df.abs()

    dfb2 = dfb.map_elements(lambda x: abs(x))
    df2 = dfb2.to_df()

    assert_dfs_equal(df, df2, dfb.count_asserts)

    ret.append(dfb2)
    return ret
```

Figure 6. Code snippet for metamorphic algorithm. Note the general flow: Check the test would be valid, run f(g(x)), run g(f(x)), check they're equal, and finally give the newly constructed "raw DataFrame" back to use to validate more metamorphic tests.

To roughly go over this test: First, we check if the test makes sense for this data. For example, it doesn't make sense to check abs("cat"), so in that case we report there are no new test DataFrames created, and exit. If it does make sense to test this data, then we perform the metamorphic test: We construct DF = abs(DataFrame(data)), in other words we calculate the absolute value of the DataFrame directly. Then, we construct DF2 = DataFrame(abs(data)), instead calculating the absolute value of the data before constructing a DataFrame. If math on a DataFrame is equivalent to math on its data, then DF = DF2 – so we assert DF = DF2, then check for errors. Finally, we report that this new DataFrame, constructed from the absolute value of the data, *could* be used as a test DataFrame. Note that both sections run the assertions. However, only section two has count_asserts = true, to keep the test report accurate. Otherwise, executions on the root DataFrames would be counted in both section 1 and section 2, giving an inaccurate number. Also note that both sections report on *potential* new test DataFrames. However, only in the first section do they get properly saved.

8.1.3 Metamorphic Implementation Computational Cost

The computational overhead of the metamorphic implementation varies wildly based on the number of tests and number of unbuilt test DataFrames used. Additionally, there is certainly some unnecessary computational overhead from using asserts even when they aren't counted, as well as from calling the same function multiple times with the same inputs. However, all of these can be summarized with a computational *time* calculation: Because the metamorphic tests complete in under 15 seconds (estimated) using only one thread on a laptop (specs: Alienware M15 R2), we conclude that the speed of calculation on such an old device means that there is not a significant computational cost.

As an aside, it is worth noting that the metamorphic tests do have a framework to perform multi-threading. However, while this framework is used, it was not completed, but due to the current computation time it would most likely need to be completed if more tests or test unbuilt DataFrames were added.

8.2 FUZZING

8.2.1 Fuzzing Methodology Details

To implement fuzzing within the DataFrame class, we first employed basic fuzzing techniques to ensure the robustness of DataFrame operations under various data inputs. Before creating specific tests, the fuzzing algorithm was run with sample test cases with expected outputs, as well as deliberately manipulating the inputs to see if the fuzzer correctly catches those manipulations. Then, the fuzzer was applied to Dataframe as shown in the figure below.

```python
def generate_random_str(length: int) -> str:
    """Generate a random string of specified length."""
    characters: str = string.ascii_letters + string.digits + string.punctuation
    result_string: str = ''.join(random.choice(characters) for _ in range(length))
    return result_string

def generate_larger_dataframe(num_rows: int) -> pd.DataFrame:
    """Generate a DataFrame with random data."""
    data = {'Name': [], 'Age': [], 'City': []}
    for _ in range(num_rows):
        # Generate random data for each column
        data['Name'].append(generate_random_str(random.randint(5, 20)))
        data['Age'].append(random.randint(20, 60))
        data['City'].append(generate_random_str(random.randint(5, 20)))
    return pd.DataFrame(data)

def fuzzer() -> str:
    """Generator function to yield random strings."""
    while True:
        # Generate a random string of random length
        yield generate_random_str(random.randint(1, 100))

def sample_function(input_str: str, df: pd.DataFrame) -> None:
    """Apply operations to the DataFrame based on the input string."""
    try:
        if 'a' in input_str:
            # Attempt to add a row to the DataFrame
            row_data = {'Name': 'John', 'Age': 25, 'City': 'Boston'}
            df.loc[len(df)] = row_data
        elif 'dr' in input_str:
            # Attempt to drop a column from the DataFrame
            df.drop(columns=['City'], inplace=True)
        elif 'un' in input_str:
            # Attempt to perform an unsupported operation on the DataFrame
            df = df.some_unsupported_operation()
```

Figure 7: This code snippet shows a random input being generated, random DataFrame to test, the fuzzer function, and operations applied to the DataFrame. This allows for different sizes of DataFrame and operations to be added and deleted to attempt to test the integrity of the DataFrame.

```python
def main():
    # Generate a DataFrame with 5000 rows
    df = generate_larger_dataframe(5000)

    results = []

    # Iterate through the fuzzer and apply operations to the DataFrame
    for i, input_str in enumerate(fuzzer(), start=1):
        # Make a copy of the DataFrame for testing
        df_copy = df.copy()

        # Call sample_function with the input string and the DataFrame copy
        sample_function(input_str, df_copy)

        # Append the DataFrame copy after applying the operation to results
        results.append(df_copy)

        # Print progress every 100 iterations
        if i % 100 == 0:
            print(f'Completed {i} iterations')

        # Break after 500 iterations
        if i >= 500:
            break
```

Figure 8. This is the main function where a specific row size of DataFrame can be specified. After operations are applied for each iteration the DataFrame is printed.

Looking at the code above, first, the generate_random_str function is utilized to generate random strings of specified lengths, which are essential for creating diverse data inputs. Next, the generate_larger_dataframe function is employed to generate a DataFrame with 5000 rows or whatever size the user desires, containing randomly generated data for the 'Name', 'Age', and 'City' columns. Subsequently, the fuzzer function is introduced as a generator function that yields random strings, serving as input for testing DataFrame operations. The sample_function is then defined to apply operations to the DataFrame based on the input string generated by the fuzzer. This function simulates a range of DataFrame operations, including adding rows, dropping columns, modifying data, sorting, resetting the index, deleting rows, and filling NaN values, while incorporating error handling to catch exceptions. After applying the operations, the export_results_to_file function exports the DataFrame copy after each operation to a text file named 'results.txt'. Finally, in the main function, the DataFrame is iterated over using the fuzzer, and operations are applied sequentially to generate results. Progress is printed every 100 iterations, and after 500 iterations, the results are exported to the 'results.txt' file. By iterating through a series of fuzzed inputs and observing the DataFrame's behavior after each operation, we could identify potential vulnerabilities or issues in DataFrame manipulation functions, ensuring their reliability and stability in real-world data scenarios.

8.2.2 Fuzzing Methodology Computational Costs

The computational cost of the fuzzing process depends on factors such as the number of iterations performed by the fuzzer and the complexity of DataFrame operations. Generating random strings and creating the DataFrame incur minimal computational overhead. The main computational cost lies in applying operations to the DataFrame in the sample_function, where the complexity varies based on the operation. Overall, while the fuzzing process is manageable for moderate-sized DataFrames, it may become more resource-intensive for larger DataFrames or complex operations.

8.2.3 Fuzzing Testing Completion Criteria

The completion criteria for the fuzzing process are determined by the number of iterations specified in the main function. In this implementation, the fuzzing process iterates through a series of random inputs generated by the fuzzer until a predefined number of iterations is reached. In the provided code, the completion criteria are set to terminate after 500 iterations. This ensures comprehensive testing of DataFrame operations under various input scenarios. Determining when to stop the iterations will be based on factors such as achieving satisfactory test coverage through 1000s of runs, detecting a sufficient number of errors, and/or reaching a overall consensus with the team.

8.3 COMBINATORIAL

8.3.1 Combinatorial Methodology Details

When developing the covering array, it was clear that there were numerous test cases that could possibly be ran and that testing each input exhaustively within the DataFrame class would take a considerable amount of time beyond the scope of this project. In order to overcome this challenge and minimize scope, a few assumptions were made to simplify the covering array generation while keeping test cases meaningful. For each parameter within DataFrame (Data, Index, Columns, and Dtype), one of three inputs would be passed. For example, three distinct two-dimensional array sizes were passed to the Data parameter which were defined as: small (ten by ten array), medium (one hundred

by one hundred array), and large (one thousand by one thousand array). For other DataFrame parameters, arrays of similar size were passed, where each array contained variables of one of three types: integers, floats, or strings. For every test case, each input type passed to DataFrame is represented by a number one, two, or three as depicted in figure 9.

DataFrame Covering Array

Test Case #	Data	Index	Coulmn	Dtype
1	1	1	1	1
2	1	1	1	2
3	1	1	1	3
4	1	1	2	1
5	1	1	2	2
6	1	1	2	3
7	1	1	3	1
8	1	1	3	2
9	1	1	3	3
10	1	2	1	1
11	1	2	1	2
12	1	2	1	3
13	1	2	2	1
14	1	2	2	2
15	1	2	2	3
16	1	2	3	1
17	1	2	3	2
18	1	2	3	3
19	1	3	1	1
20	1	3	1	2
21	1	3	1	3
22	1	3	2	1

Legend

Data
1 - Small 2D array, 100 elements
2 - Medium 2D array, 10000 elements
3 - Large 2D array, 1000000 elements

Index
1 - Array of integers
2 - Array of floats
3 - Array of strings

Columns
1 - Array of integers
2 - Array of floats
3 - Array of strings

Dtype
1 - Array of integers
2 - Array of floats
3 - Array of strings

Figure 9. Covering array with legend detailing assumptions made for each DataFrame parameter (Data, Index, Columns, and Dtype). Each row represents a test case described by a sequence of numbers. The legend shows the relationship between numbers (1, 2, 3) and values that are passed to DataFrame.

After generating the covering array, it was imported into a function called test_system, as seen in figure 10. Each row within the covering array is extracted and passed to test_system as function parameters until all test cases have run. This approach helped streamline the testing process and allowed for tests to run semi-autonomously. Unfortunately, each test case had to be manually reviewed to find errors which required more time than anticipated.

```
test_case = 1
with open("test_case_array.txt") as fo:
    lines = fo.readlines()
    for line in lines:
        values = line.strip().split()
        params = [int(value) for value in values]
        print(f"\nTest case #: {test_case}\nCovering Array Input: {params[0]},{params[1]},{params[2]},{params[3]}\nDataFrame Output:")
        test_case += 1
        print(test_system(params[0],params[1],params[2], params[3]))
```

Figure 10. This is the main body of the testing program. For each iteration of the for loop, a test case is imported from the covering array text file and is passed to the test_system function. This program prints the output of test_system to be reviewed by the user.

```
# Combinatorial Testing - This function works by accepting an argument of 1, 2, or 3.
def test_system(array_size, index_type, column_type, data_type):    # Note array size
    # Array size is assigned here. Array sizes are split up in three distinct sizes,
    if array_size == 1:                                             # small size 2D a
        size = 10
        array = generate_array_size(size)                           # value to be use
    elif array_size == 2:                                           # medium size 2D
        size = 100
        array = generate_array_size(size)
    elif array_size == 3:                                           # large size arra
        size = 1000
        array = generate_array_size(size)
    else:
        print("Error, invalid input for array_size")

    # Index_type is assigned here. The type is either float, string, or integer. Note
    if index_type == 1:                                             # generates an in
        index  = generate_integer_array(0, size)
    elif index_type == 2:
        index = generate_float_array(0, size)                       # generates an in
    elif index_type == 3:
        index = generate_string_array(0, size)                      # generates an in
    else:
        print("Error, invalid input for index_size")
```

Figure 11. This code describes the behavior of the test_system function. Each argument passed to test_system undergoes an if-else check to determine which test case trait is present. Within each branch of the if-elif code block, a value is assigned depending on the test case. Separate functions, such as generate_array_size and generate_integer_array are used to generate the arrays needed to test DataFrame.

```
    return pd.DataFrame(array,index, column, data)
```

Figure 12. Test_system function output. At the end of test_system, a set of test case values are assigned and passed to DataFrame to generate an output. This output is returned and is printed in the main program.

9 RESULTS

9.1 METAMORPHIC

9.1.1 METAMORPHIC RESULTS

As described in section 8.1, the first section of metamorphic testing turns some root DataFrames into many test DataFrames. Fro this project, we start with two root DataFrames, and convert it into 566 test DataFrames. The first root DataFrame is built to represent string-based DataFrames, consisting of only strings of text, while the second root is built to represent many number-based DataFrames, including integers, real numbers, and NaN values. In the second section, we then run each type of metamorphic testing to each of these test DataFrames. From this, the counter counts 9484 tests run metamorphically without any errors found.

9.1.2 METAMORPHIC RESULT IMPLICATIONS

Because of the large number of successful metamorphic tests, we conclude that manipulating a DataFrame is equivalent to

manipulating its individual data points. Additionally, due to the repeated construction of DataFrame from data (without any issues), it also shows that constructing DataFrames is very robust.

However, there were some limitations to these metamorphic tests. For example, all of these tests were tests involving only one DataFrame, and most of the tests did some operation with the DataFrame and another, singular value. We did not test many operations between multiple DataFrames: For example, a very common operation is matrix multiplication between two matrices. Given that DataFrames are commonly used as matrices, testing this function could also have been useful.

9.2 FUZZING

9.2.1 Fuzzing Testing Evaluation

Evaluating the fuzzing validation, after subjecting the DataFrame to 2800 runs of the fuzzer, no errors were detected, indicating a high level of robustness and reliability in handling various data inputs and operations. A sample output is shown in the figure below, where the input string and resulting DataFrame are printed.

```
Run #100
Input: M|1wa1~9wfq8^nY6NbKo?d`k{u`?,SY$*/]bq$Xw%)-
WYa:RDHz9~QR<F0<z1%:>zWcd2(ePF7[YM<3S
DataFrame after operation:
                    Name  Age                  City
0     ocf?MJ~]P_%RHRH\`4I   46         )TrqB8"|Kt'AyN
1           n+])=Y4d}xztMY   44         TWt2e!$4'"1oFu
2             VBD0)sZ'~$$z   42            %V8|2cK$+$+
3                   i"nd?O   53      e~:_r8s9^ZmZ&\AN+
4                 uc,3JD,    43                k&mkeh!
...                    ...  ...                   ...
996               }sU{|}F    28        ,=dy>Hl,1-L|1zS
997          ~)am."R)kLHsL   39   Bj"^x<3$J{h-l$jZ}.lP
998             6qBcbe#z?e   39      V6>G5_#T7&Hv4=Tw(
999              Ybe0|?9?K   20         2Gb>%w\9_pA`w-
1000                 John    25                 Boston

[1001 rows x 3 columns]
================================================
```

Figure 13. This is an example output of the Fuzzer for Run#100 for a 1000 row DataFrame. Each row respresents an entry with random strings for 'Name', 'City' and 'Age.' An additional row of John, 25, and Boston is appended, indicting successful execution of the operation to add a row in the 'sample_function' part of the code.

9.2.2 Fuzzing Defect Detection

If the DataFrame did not pass the series of operations that were performed on it, then an "error" would be printed instead. For all runs, no "error" was seen printed. Trying different rows, operations, and DataFrames, there was a success in outputing the correct manipulated DataFrame. When analyzing the outputs based on what was expected from the manipulations, no defects were detected by the code. This result was expected as all runs successfully passed.

9.2.3 Potential Fuzzing Testing Improvements

This successful validation underscores the effectiveness of the fuzzing approach in ensuring the stability of DataFrame functions. Many different manipulations were done to the DataFrame to create errors in any part of the output, but none were discovered. Moving forward, the next steps involve expanding the scope of DataFrame manipulations and inputs to encompass a broader range of potential use cases. Additionally, integrating feedback from the pandas community regarding the tests we performed, will further refine the fuzzing process and address any existing or emerging concerns. Furthermore, the plan includes leveraging more advanced fuzzing tools and techniques to enhance the effectiveness and efficiency of the testing process, thereby continuously improving the dependability of DataFrame operations.

9.3 COMBINATORIAL

9.3.1 Combinatorial Testing Evaluation

As described in figure 11, each parameter within test_system undergoes an if-else check that sets the input parameter that will be passed to DataFrame. For example, if test case #5 is ran, the covering array will have passed one, one, two, and two to test_system. Therefore, array_size will have been passed a one. This sets the variable size to ten (representing a ten-by-ten array) and generates an array of integers with the appropriate size using a generate_array_size function. Similarly, the index_type parameter will have been passed a one leading to the labels consisting of an array of integers. Both column_type and data_type parameters will have been passed a two, resulting in the column labels becoming an array of floats and the DataFrame data type being set to float. This test case example and its associated output can be seen in figure 14, which depicts a ten-by-ten array with the traits described above.

```
Test case #: 5
Covering Array Input: 1,1,2,2
DataFrame Output:
    0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0
0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0
1  11.0  12.0  13.0  14.0  15.0  16.0  17.0  18.0  19.0  20.0
2  21.0  22.0  23.0  24.0  25.0  26.0  27.0  28.0  29.0  30.0
3  31.0  32.0  33.0  34.0  35.0  36.0  37.0  38.0  39.0  40.0
4  41.0  42.0  43.0  44.0  45.0  46.0  47.0  48.0  49.0  50.0
5  51.0  52.0  53.0  54.0  55.0  56.0  57.0  58.0  59.0  60.0
6  61.0  62.0  63.0  64.0  65.0  66.0  67.0  68.0  69.0  70.0
7  71.0  72.0  73.0  74.0  75.0  76.0  77.0  78.0  79.0  80.0
8  81.0  82.0  83.0  84.0  85.0  86.0  87.0  88.0  89.0  90.0
9  91.0  92.0  93.0  94.0  95.0  96.0  97.0  98.0  99.0 100.0
```

Figure 14. An example output described by test conditions associated with test case #5. The test number and covering array inputs are printed at the top of the output to serve as a reference.

9.3.2 Testing Results

Over eighty-one test cases were ran using this combinatorial approach, where no unexpected program behavior was observed during review. These results suggest that DataFrame is particularly robust with respect to issues that stem from problematic combinations of variables.

9.3.3 Potential Combinatorial Testing Improvements

One possible improvement to the testing process would be to identify a method to automatically identify test cases that result in an error. This would reduce the amount of review time required for each test case and would make running more tests feasible, as each output required manual review. Another method that may improve testing would be expanding the covering array coverage either by adjusting assumptions or using complex algorithms to aid in covering array generation.

10 CHANGE IMPACT ANALYSIS

The integration of metamorphic, combinatorial, and fuzzing testing methodologies into the validation process of Pandas and DataFrame holds profound implications for future users of these essential data manipulation tools. By subjecting Pandas and DataFrame to rigorous testing, we not only ensure the reliability and correctness of their functionalities but also mitigate potential risks and vulnerabilities that could impact users' data integrity and decision-making processes. Metamorphic testing, by validating the expected behavior of DataFrame transformations, enhances users' confidence in the accuracy of their data analysis results, empowering them to make informed decisions based on testing insights. Combinatorial testing optimizes the efficiency of testing processes, allowing users to navigate complex datasets and configurations with ease while minimizing the likelihood of unexpected errors or inconsistencies. Additionally, fuzzing testing serves as a critical line of defense against potential security threats, safeguarding users' data from malformed inputs. By prioritizing the integrity, reliability, and security of Pandas and DataFrame, these testing methodologies pave the way for a robust future for the Pandas community. This is where data-driven decision-making is underpinned by trust, transparency, and ethical considerations, ensuring the continued relevance and impact of these powerful data manipulation tools in diverse domains and industries.

11 SOFTWARE LICENSE

The team used Python and Pandas which is a free software released under the three-clause BSD license.

12 CONCLUSIONS

As described in the above sections of this paper, we tested the latest version of Pandas (version 2.2.2) with Python 3.10.11 using multiple types of advanced software validation. We completed 81 combinatorial tests, 2800 fuzz-based tests, and 9484 metamorphic-based tests. None of these tests found any bugs, unexpected crashes, or errors. Therefore, we conclude that the Pandas open-source python library is very robust with a high level of confidence.

REFERENCES

[1] E. Dijkstra, “The Humble Programmer,” Communications of the ACM 15, 10, pp. 859–866, 1972.

[2] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.

[3] P. Arcaini, A. Gargantini, and P. Vavassori, “Validation of Models and Tests for Constrained Combinatorial Interaction Testing,” 2014, pp. 98–107.

[4] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A Systematic Review of Fuzzing Based on Machine Learning Techniques," IEEE, Research Article, 2024.

[5] T. Y. Chen, S. C. Cheung, and S. M. Yiu, “Metamorphic Testing: A New Approach for Generating Next Test Cases,” 1998.

[6] Google, "OSS-Fuzz: Continuous Fuzzing for Open Source Software," [Online]. Available: https://google.github.io/oss-fuzz/. Accessed: Feb. 1, 2024.

[7] Google, "Why Fuzz? - OSS-Fuzz documentation," [Online]. Available: https://github.com/google/fuzzing/blob/master/docs/why-fuzz.md. Accessed: Feb. 1, 2024.

[8] AllPairs, "allpairspy: A Python module for pairwise test coverage," [Online]. Available: https://pypi.org/project/allpairspy/. Accessed: Feb. 1, 2024.

[9] “Pandas Issues,” GitHub. [Online]. Available: https://github.com/pandas-dev/pandas/issues. [Accessed: 01-Feb-2024].

[10] “Washington State Profile and Energy Estimates.” EIA, U.S. Energy Information Administration, 18 Apr. 2024, www.eia.gov/state/analysis.php?sid=WA.

[11] “California State Profile and Energy Estimates.” EIA, U.S. Energy Information Administration, 20 Apr. 2023, www.eia.gov/state/analysis.php?sid=CA. Conference Name:ACM Woodstock conferenceConference Short Name:WOODSTOCK’18Conference Location:El Paso, Texas USA ISBN:978-1-4503-0000-0/18/06Year:2018 Date:June Copyright Year:2018 DOI:10.1145/1234567890 RRH: F. Surname et al.

[12] Olson, P. (2022, August 1). Pandas is now as popular as python was in 2016. Ponder. https://ponder.io/pandas-is-now-as-popular-as-python- was-in-2016/

APPENDIX A – PROJECT ARTIFACTS

GitHub repository: https://github.com/cpaca/ECE595-OSS-Validation/tree/main. This repository represents an online copy of our codebase for us to use as a backup in-case of corruption and as a means of collaboration for our team as well as a means for us to check each other’s results and work.

Pandas GitHub: https://github.com/pandas-dev/pandas This repository stores the main source code of Pandas. It also stores issues related to Pandas, as well as

Python: https://www.python.org/downloads/ Python – the program – is an executor that can execute Python programs, and is required to execute Pandas. We also used Pip, packaged with Python, to facilitate installation. We used python version 3.10.11, pip version 23.3.2, and pandas version 2.2.2.

PyCharm (IDE): https://www.jetbrains.com/pycharm/
Spyder (IDE): https://www.spyder-ide.org/
VSCode (IDE): https://code.visualstudio.com/
PyCharm, Spyder, and VSCode are all IDEs used to help with development of Python programming.

APPENDIX B – ETHICAL ASSESSMENT

Cole Thornton: The work completed during the scope of this project has ethical value, demonstrating principles 3.08 and 4.03 as described in the *Software Engineering Code of Ethics and Professional Practice*. As depicted in section 3.08, validating DataFrame helped ensure that it "satisfied users' requirements" under a variety of different testing conditions and is robust with a high level of confidence. Since this is one of the most popular classes within the Pandas library, the results of this paper have the potential to positively impact multiple industries within the data analysis field. As described in section 4.03 of the IEEE, the team has maintained professional objectivity throughout the course of the project by remaining impartial when evaluating the results obtained through the validation of DataFrame. Furthermore, we provided transparency with respect to each testing method used.

Sriram Ganesh: From my perspective, there is also substantial ethical value that this project reveals. By subjecting the DataFrame class to rigorous testing including metamorphic, combinatorial, and fuzzing techniques as well as performing additional iterations as needed, the team made sure to prioritize integrity and reliability of data manipulation operations. Being such a largely used tool, it is necessary to perform thorough tests of Pandas and present any issues that occur well-documented. Following the IEEE code of ethics, the process of validation allows us to "disclose promptly factors that might endanger the public or the environment." Being such a widely used data analysis tool, through validation, we are making sure that future data analysis that occurs from using Pandas is accurate. By clearly documenting our test processes, our team has ethically validated the robustness of pandas. Additionally, we have clearly prioritized continuous improvement as new tools are added to Pandas. The work the team has done, can indeed to be used to prove data integrity and reliability in reference to the DataFrame class. Following the code of ethics, we have made sure to "acknowledge and correct errors in code, creating honest and realistic claims based on the results of our test".

C.P. Pacaba: In my opinion, this project grants positive ethical value and produces no negative ethical value. For this point, I will primarily reference the IEEE Code of Ethics Section 1: "to hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, to protect the privacy of others, and to disclose promptly factors that might endanger the public or the environment". The positive ethical value comes from pandas' use in ML models. Pandas is extremely widely used in many ML models to the extent that it's used for Microsoft's "Data Science for Beginners" repository and is one of many libraries supported by the datasets provided by Hugging Face, a place where many pre-trained data science models are shared. Therefore, validation of Pandas leads to validation that these ML models work as designed and as expected. I was, however, able to find some (small) negative ethical value: The electrical usage from doing these tests could have some environmental effect. However, we didn't rent out anything significant here (e.g. supercomputers) and the computers we did use are located in Washington State and California (in the US), which the US Energy Information Administration reports two of the top three states in terms of renewable energy generation. [10][11]. Because of this, I don't believe a power consumption argument would be very strong.

APPENDIX C – POSTMORTEM

In postmortem, the team has reflected on the material we presented in our project proposal and the outcomes. We were able to assess what went well, what went poorly, and identify two specific failures during the project, along with their root causes. Looking at the proposal, the team chose to investigate Pandas, due its critical role in diverse industries. The proposal involved ensuring Panda's reliability in handling complex data manipulation tasks. Specifically, to focus on a class in Pandas, to create a specific scope for the semester, the team focused on DataFrame. The proposed work involved addressing critical bugs within Pandas, using state-of-the-art testing and validation techniques such as combinatorial, fuzz, and metamorphic testing. The testing process we proposed was iterative, involving cycles of testing, issue identification, and resolution. As issues are identified through the testing methods outlined above, they will be prioritized based on severity. To evaluate the success of the efforts, we had aimed to measure the bug resolution rate, testing coverage, issue severity reduction, feedback from development teams, and stability and reliability with changes. A summary of our proposed work from the proposal is shown below.

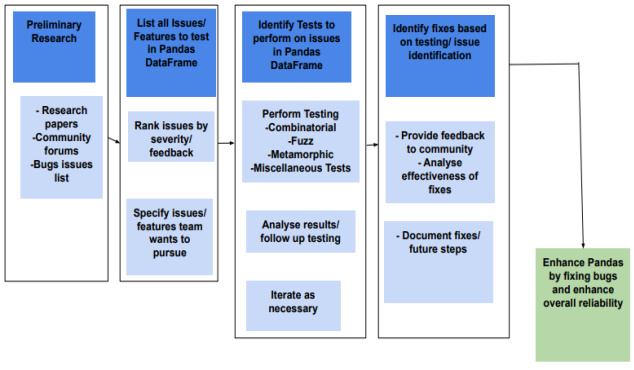

Figure 15: Proposed work diagram taken from the validation study proposal. The diagram depicts an early formulation of the

work to be completed for the project. Each block represents a stage within the project with associated work to be completed.

Comparing the project proposal to our final outcomes, the team learned from the proposal early on, that our goal of identifying new issues and solving them should not be our main scope. After getting feedback, our team adjusted our project to focus on the types of testing techniques used and instead creating strong tests backed with results, rather than going out and finding a specific type of issue. The overall goal was to quantify the robustness of the DataFrame class in the Pandas library. The team was able to accomplish that by applying advanced software validation techniques, including combinatorial testing, fuzz testing, and metamorphic testing to potentially address issues within Pandas. The process that we took to accomplish this was a success. We started off by familiarizing ourselves with the Pandas library, setting up a test suite, and knowing the simple functions of DataFrame through small scale tests. By doing this first, it allowed the team to develop more focused and well-rounded tests. Although the team did not find any specific issues, the overall goal of evaluating robustness was a success from the perspective of the DataFrame Class.

Throughout our validation testing, numerous aspects unfolded positively, enriching our project execution and fostering collaborative success. Firstly, we achieved significant milestones by effectively implementing and executing three distinct testing methodologies: fuzz testing, combinatorial testing, and metamorphic testing. With thousands of runs conducted, we meticulously analyzed the Pandas DataFrame class, garnering insightful results crucial for assessing its robustness. Secondly, our commitment to thoroughness extended beyond testing as we dedicated ample time to code refinement and documentation, ensuring clarity and reproducibility of our findings. Moreover, our team demonstrated remarkable adaptability, seamlessly adjusting strategies and timelines to navigate unexpected challenges and optimize project outcomes. Thirdly, our adherence to the project schedule was exemplary, facilitated by open communication and proactive problem-solving, enabling us to address issues promptly and maintain project momentum. Lastly, our adept utilization of available resources, including documentation and community support, augmented our testing efforts, underscoring our commitment to maximizing efficiency and effectiveness. These collective achievements underscore the collaborative spirit and dedication that defined our project journey.

While our project encountered no major setbacks, several areas could have benefited from improvement. Firstly, despite our thorough testing efforts, we found it challenging to identify significant issues within the Pandas codebase. While this underscores the robustness of the DataFrame class, it also suggests the need for more comprehensive testing to uncover potential vulnerabilities or edge cases. This testing might involve further advanced techniques beyond the scope of our project. Additionally, our attempts to engage with developers and the broader Pandas community regarding potential issues were somewhat limited. This may indicate a lack of prominent issues or the exceptionally robust nature of Pandas, thereby reducing the urgency for community involvement. Furthermore, we struggled to find open issues within the project scope that aligned with our objectives. While we anticipated contributing to issue resolution, we encountered primarily small-scale issues that did not require significant intervention. Moving forward, addressing these areas could enhance our project's impact and foster deeper engagement with the Pandas community.

Two failures are evident in our project postmortem. Firstly, we underestimated the time required for learning and implementing advanced testing techniques, resulting in an inability to conduct all planned tests thoroughly. The root cause of this is a lack of accurate assessment of the complexity involved and inadequate allocation of time for exploration and execution. The team was able to write tests and get results showing robustness of Pandas, however, our approach to testing changed from one focusing on issues, to the broader nature of Pandas. Secondly, as mentioned, we encountered difficulty in identifying specific issues within the Pandas DataFrame class, highlighting the challenge posed by the library's robustness and maturity as an open-source project. The root cause of this is the extensive codebase and diverse use cases of Pandas, making it challenging to pinpoint critical vulnerabilities or defects. Additionally, the fact that many issues within Pandas are relatively small due to its robustness, making them harder to detect. These factors contributed to limited issue identification during our validation campaign.

In conclusion, our project postmortem provides valuable insights into our journey of validating the robustness of the DataFrame class in the Pandas library. While we achieved significant successes in implementing advanced testing techniques and analyzing the DataFrame class, we also encountered challenges and limitations that shaped our learnings moving forward. Two key lessons emerge from our experience. Firstly, accurate time estimation and allocation are crucial for effectively executing complex testing endeavors, especially when exploring large-scale open-source projects like Pandas. Our underestimation of the time required for learning and implementing advanced testing techniques underscores the importance of thorough planning and realistic expectations. Secondly, the nature of Pandas as a robust and mature open-source project presents unique challenges in identifying specific issues within its codebase. While our focus on the DataFrame class yielded valuable insights, going forward in the future, we will continue to adjust the current testing methodologies we used, as well as explore more classes within Pandas.