Bachelor Thesis

for the Bachelor of Engineering degree program in Mechatronics

Development of a Continuous Integration Framework for OptiSlang Workflows

In co-operation with

Robert Bosch GmbH

Author : Sathvick Bindinganavale Srinath
Matriculation Number : 4020025

Supervisor : Mr. André Haeitmann Dutra
1st Examiner : Prof. Dr.-Ing. Andreas Schiffler
2nd Examiner :

Submission Date : 06/10/2024

DECLARATION

I declare that the work in this dissertation, **"Development of a Continuous Integration Framework for OptiSlang Workflows"** was carried out in accordance with the regulations of the Technical University of Applied Sciences Würzburg-Schweinfurt. I declare that I have acknowledged all the sources I have used in writing this dissertation. I have clearly marked and acknowledged all direct quotations and all information obtained from other sources. I have not used any other sources or resources than those indicated. I have not submitted this dissertation to any other examination board. I am aware that any violation of these rules will result in the revocation of my degree. I am also aware that the dissertation may be checked for plagiarism using appropriate software.

_____ _____

Date Signature

Contents

List of Figures

Listings

Abbreviations

API ..... Application Programming Interface

CAE ..... Computer-Aided Engineering

CD ..... Continuous Deployment

CI ..... Continuous Integration

GUI ..... Graphical User Interface

HTTP ..... Hypertext Transfer Protocol

JSON ..... JavaScript Object Notation

MOO ..... Multi Objective Optimization

PIDO ..... Process Integration and Design Optimization

SOO ..... Single Objective Optimization

YAML ..... YAML Ain't Markup Language

Abstract

lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc

Chapter 1

Introduction

1.1 Overview

This thesis explains about the creation and development of a framework for Optislang workflows.

1.2 Objective

The objective of this thesis are as follows:

- Development of a method to create standalone modules in Optislang, based in MATLAB and Python.(only code based, without using GUI).
- Implementation of modules and workflows without the help of GUI.
- Creation of a framework in Python to create and test modules and workflows in Optislang.
- Establishment of a strategy for automated integration testing in Github for modules based in Python and MATLAB.

1.3 Outline

- In Chapter 2, we are going to discuss about the MOO project, its role in the company and the modules created in Python and Optislang.
- In Chapter 3, we will discuss the motivation and the principles used in the development of the framework.
- In Chapter 4, we understand the creation of a framework to create standalone modules and workflows in Python.
- In Chapter 5, we will look into the automated integration testing in Github.
- In Chapter 6, we will discuss the results and evaluate the impact of the solution.
- Finally, Chapter 7 provides us the summary of the work, highlighting the achievements and feedback for development in potential areas.

Chapter 2

Multi Objective Optimization

2.1 Introduction

In today's increasingly complex world, decision-makers often face the challenge of optimizing several conflicting objectives simultaneously. Multi Objective Optimization (MOO) is an optimization that deals with such problems, where multiple objective functions are optimized simultaneously. To understand MOO better, let us consider an example.

Example: Let us consider an example of a car manufacturer. The car consists of many components like engine, body, wheels, etc which can be tweaked. In our case, the manufacturer wants to optimize the car for two objectives: lower manufacturing cost of the car and lower carbon emissions. With considering the input parameters and the objectives, we get many solutions as shown in Figure 2.1

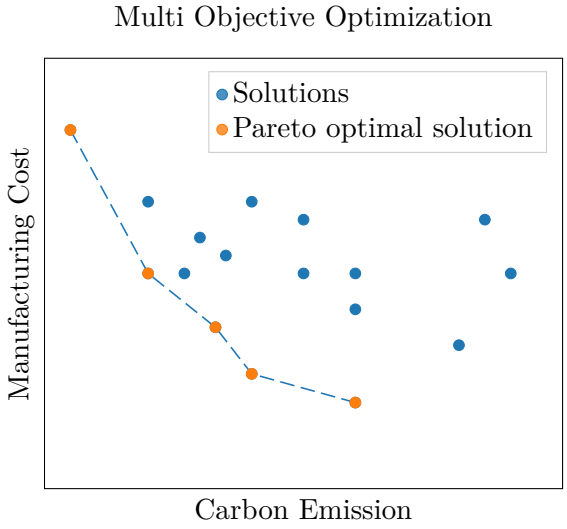

Figure 2.1: Example of MOO

In an MOO problem, there typically is no single best solution. Rather, the *goal* is to identify a set of solutions that are optimal in terms of all objectives. In Figure 2.1, the best solutions for the given objectives is indicated in orange known as pareto optimal solutions. A solution is said to be pareto optimal if no other solution can improve on any of the objectives without worsening at least one of the other objectives.

2.2 Difference between MOO and SOO

Optimization problems, whether single-objective or multi-objective, have the same goal: to find the best solution(s) to a given problem. However, the approach to solving these problems is different.

In Single Objective Optimization (SOO), the goal is to optimize a single objective function, which can either be maximized or minimized. The problem is simpler to define and solve because it involves only one objective. To calculate SOO, we can use methods like gradient descent, linear programming, etc.

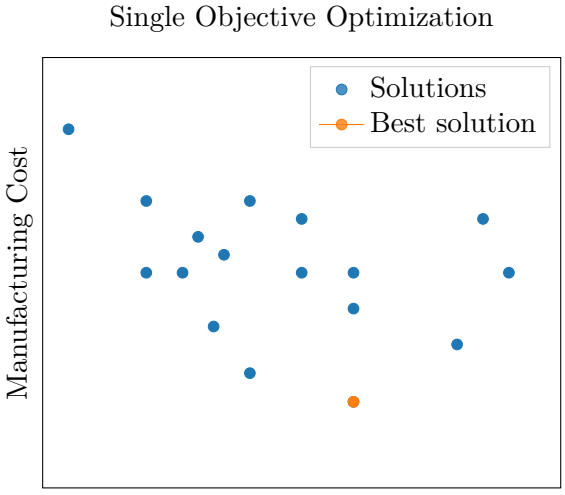

Figure 2.2: Example of SOO

In Figure 2.2, we have considered the same example given in section 2.1. But, here, we are considering only one objective, which is to minimize the manufacturing cost. The best solution is indicated in orange.

In MOO, the optimization involves two or more objective functions simultaneously. The problem is more complex because the objectives are often conflicting. Unlike SOO, where we have a single best solution, in MOO, we have pareto optimal solutions. To calculate MOO, we can use methods like pareto optimization, scalarization method, weighted sum method, ϵ-constraint method, etc.

While SOO focuses on finding the best solution according to a single criterion, MOO addresses the more complex task of balancing multiple, often conflicting objectives. The choice between SOO and MOO depends on the nature of the problem at hand and the goals of the decision-maker. Understanding the differences between these approaches is crucial for selecting the appropriate optimization technique and achieving the desired outcomes.

2.3 Optislang

To calculate MOO, we need a software platform that can handle the complexity of the problem. Ansys Optislang [1] is such a software platform, which is used for design exploration, Computer-Aided Engineering (CAE) based sensitivity analysis and optimization in conjunction with any product development tool. It is a Process Integration and Design Optimization tool or in short, a PIDO tool. Process Integration refers to automate and orchestrate manual simulation processes and

to realize complex workflows. Design Optimization aims for better understanding of your design, optimizing the product, identify an improved design which has the desired qualities and resulting in a best design by reliability analysis and statistical analysis.

Optislang uses several solvers to look into aspects like mechanical, technical, mathematical and any other problems. This is easier in Optislang as it provides integration to create toolchains of many external programs like ANSYS, MATLAB, Excel, Python, CATIA and many more.

Our department utilizes Optislang for solving MOO problems, as it includes algorithms specifically designed for MOO.

2.4 Modules and Workflows

2.4.1 Modules

Modules are created by the system developers. Modules include a simulation model as a calculation with defined interfaces for coupling with other modules. These modules are either defined in MATLAB or Python. Each module is designed to tackle/improve a specific issue. To document and collaborate with other system developers, each module is versioned and stored in a specific manner in a repository in GitHub Enterprise.

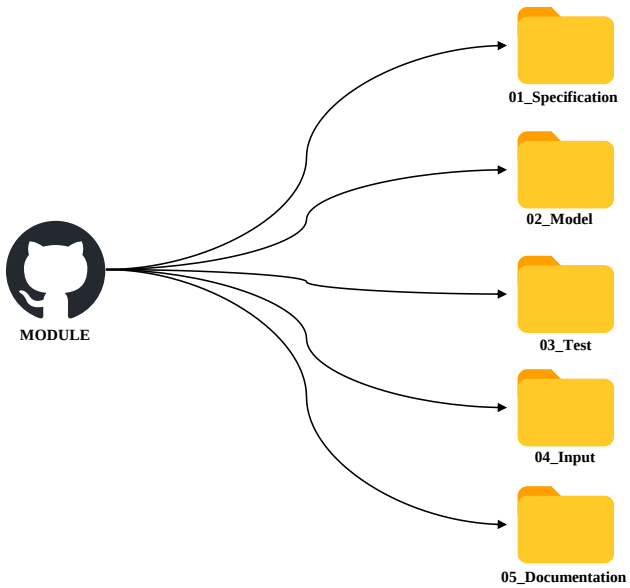

Figure 2.3: Example of a module's GitHub repository structure

Figure 2.3 shows us an example of how each module is maintained in our GitHub organization.

- `01_Specification` has all the requirements for the module to run.

- `02_Model` contains all parts to run the model. This can also be used as a playground for the development of a model.
- `03_Test` contains all the unit tests for the module. This is to ensure that the module is working as expected.
- `04_Input` carries all the initial parameters or functions to be defined at the start of a module.
- `05_Documentation`contains documents explaining functioning and usage of the module.

2.4.2 Workflows

Our department develop automatized workflows for power electronics products considering functional loads to reliability indications. Workflows are a sequence of modules designed for a fast calculation performance characteristic like temperature or reliability indication. To develop a workflow, Optislang is used.

Every architectural workflow has a GitHub repository that is maintained in a similar way to how modules are being maintained.

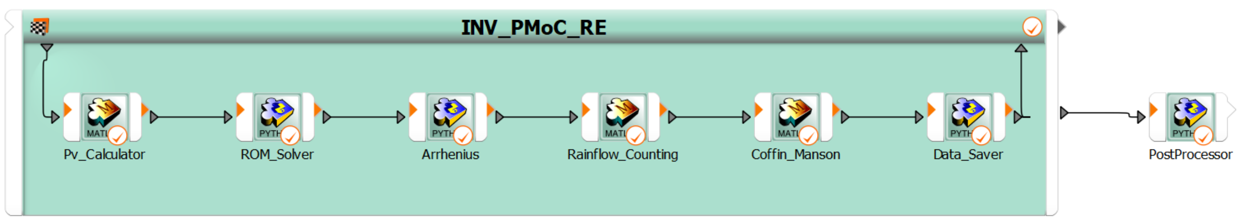

Figure 2.4: Example of a workflow in Optislang

2.5 Current Problem

lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc

Chapter 3

Practical Application of the Thesis

To overcome the problems discussed in section 2.5, this thesis proposes a solution to automate the process of testing standalone modules in Optislang. Since, the process is automated, the testing of modules needs to be done without the help of the GUI. To achieve this, a Python framework is created to test the modules in an according manner. To use the framework in an automated manner, a CI pipeline is created using GitHub Actions. The pipeline is triggered whenever a new commit is pushed to the repository. The pipeline runs the tests on the modules in a virtual machine and checks if the results are as expected. If the tests fail, the pipeline notifies developer about the failure. The developer can then look into the issue and resolve it.

To achieve this, the following sections explains the concepts and practices like DevOps, Continuous Integration, Version Control and code quality in the context of this thesis.

3.1 DevOps

3.1.1 Introduction

Earlier, software development was divided into two separate phases: development and operations. The development team was responsible for writing the code and building the software, while the operations team was responsible for deploying and maintaining the software. If the software failed in production, the operations team would send the software back to the development team for fixing. This division of responsibilities often led to many issues

- Developers needed to wait for the feedback from the operations team, which slowed down the development process.
- Developers passed the software to the operations team without knowing how it would perform in production, which led to many issues.
- Collaboration between the development and operations teams was limited, which made it difficult to resolve issues quickly and efficiently.

To overcome these challenges, DevOps was introduced. **DevOps** is a software development approach which encourages the collaboration between development(Dev) and operations(Ops) team [8]. From this collaboration, the software development lifecycle is faster, more efficient and more reliable. These teams implement processes and tools to automate labour intensive, manual tasks which were slowing down the software development process. This also eliminates the need to depend on the other team for feedback.

3.1.2 DevOps Lifecycle

DevOps lifecycle is a set of phases, which brings in the development and operations group together to manage the entire software development lifecycle. During each of the phases, different practices and tools are used to automate and streamline the software development process. Figure 3.1 shows the DevOps lifecycle. The development phase is depicted on the left side of the loop, whereas the operations phase is depicted on the right side of the loop. The loop represents the continuous nature of the DevOps lifecycle.

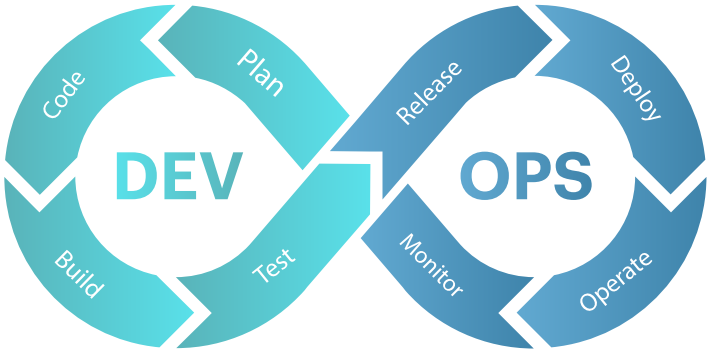

Figure 3.1: DevOps Lifecycle [2]

The DevOps lifecycle consists of the following phases:

- **Plan**:
  The initial phase of any software development process is planning. During this stage, teams outline and prioritize the features to be developed. This involves defining requirements, setting objectives, and creating a project roadmap.
- **Code**:
  After the requirements are defined, developers start implementing the code. Developers must adhere to the coding standards, versioning and collaborating with other developers to ensure code quality.
- **Build**:
  The build phase involves compiling the code and packaging the software. This phase ensures that the code is error-free and ready for deployment across different environments.
- **Test**:
  The testing phase involves running various automated tests to ensure that the software meets the requirements and is free of bugs. This phase includes unit tests, integration tests, and system tests. This helps to identify and fix issues early in the development process.
- **Release**:
  After the code has been tested and verified, it is released to the production environment. After verifying the code from bugs and errors, the code is passed on the next phase.

- **Deploy**:
  The code is automatically deployed to the production environment. This phase involves setting up the infrastructure, configuring the servers, and deploying the software.

- **Operate**:
  The operations team is responsible for verifying that the software is operating smoothly in the production environment.

- **Monitor**:
  To verify if the software is running smoothly, the operations team monitors the software. This phase involves collecting and analyzing data to identify the health, performance, and security of the software. This data is used to identify and fix issues before they impact the end-users.

DevOps practices play a crucial role in the development of the automation process described in this thesis. By integrating Continuous Integration (CI) and Continuous Deployment (CD) pipelines, we ensure that the testing of modules is efficient and reliable. This helps us to improve productivity and reduce human error. According to [10], DevOps is not only helping to bridge the gap between development and operations teams, but also helping to improve the quality of the software. The DevOps approach allows for seamless collaboration between development and operations teams, ensuring that the testing framework and the modules it tests are consistently maintained and updated. This integration of DevOps practices not only enhances the quality of the software but also accelerates the development lifecycle, enabling faster delivery of new updates and features.

In summary, the application of DevOps in this thesis demonstrates how modern software engineering practices can be applied to automate and streamline the testing process, leading to more robust, efficient and reliable software solutions.

3.2 Continuous Integration

Continuous Integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. This practice allows to frequently merge code changes into a central repository where builds and tests are ran later on. A CI pipeline can be applied to automate and streamline the testing process, leading to more robust, efficient and reliable software solutions [4].

For the CI pipeline to work, usually the code changes are stored in a version control system which can be collaborated by multiple developers. Examples of such version control systems are Bitbucket, GitLab, GitHub. A crucial practice of CI is to commit the code changes to the repository frequently [7]. In this thesis, GitHub is used as the primary version control system. After the code changes are committed, the CI pipeline is triggered automatically.

A detailed implementation of the CI pipeline is discussed in Chapter 5.

3.3 Version Control Strategies

To track changes in the codebase, version control is used. It acts as a safety net for the developers, allowing them to revert to the previous version of the codebase if something goes wrong. Git, Mermaid and many other distributed version control systems provides the same functionality. In this thesis, Git[1] is used as the primary version control system. This method of tracking is setup locally. To collaborate with other developers, the codebase needs to be pushed to a remote repository. In our organization, GitHub is used as the main source for hosting the codebase. By hosting a remote+ repository, developers can collaborate with each other, track changes, and manage the codebase effectively. Version control is considered to be an important practice in DevOps, as they reduce the development time of a product and increase the quality of the software by frequent deployment.

While working on a feature, or a bug fix, developers tend to create a new branch in the repository, so that the main branch is not affected. There are two main branching strategies : long-lived branches and short-lived branches.

3.3.1 GitHub Flow

This branching strategy is a simplified version of the Git Flow. It is a lightweight, branch-based workflow that is designed around deploying code to production. It uses the concepts of branches and pull requests, which is ideal for working in teams and frequent deployments.

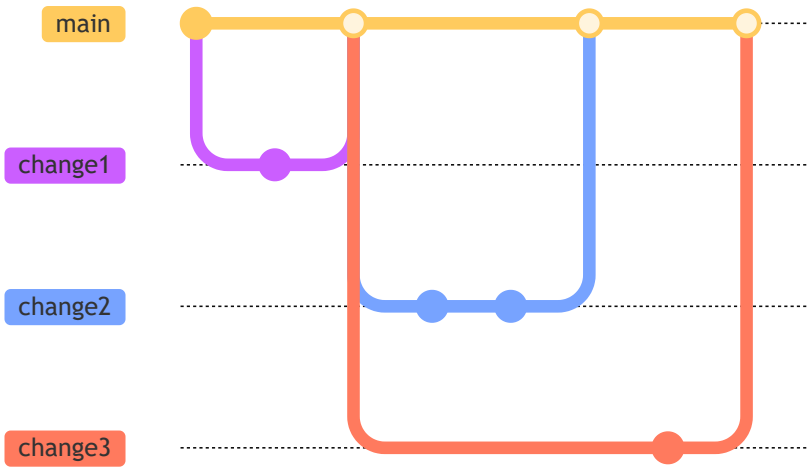

Figure 3.2: Example of GitHub Flow

- **Master Branch**:
  The master/main branch is the default branch in the repository. This branch is always deployable and contains the latest stable version of the code.

- **Feature Branch**:
  When a new feature is to be implemented, a new branch is created from the master branch. This is practiced to avoid conflicts with the code in the main branch. The developer works

[1] `https://git-scm.com/URL`

on the feature branch until the feature is completed. After the feature is completed, reviewed and tested, the branch is merged back to the main branch. In the figure 3.2, `change1`, `change2` and `change3` are the feature branches.

GitHub Flow is simple to learn and implement and provides faster feedback to developers.

3.3.2 Git Flow

Git Flow is a more complex branching model compared to GitHub Flow. It is a branching model that helps to manage the codebase in a more organized manner for managing large projects with scheduled releases. It uses two long lived branches, master and develop, along with several short-lived branches.

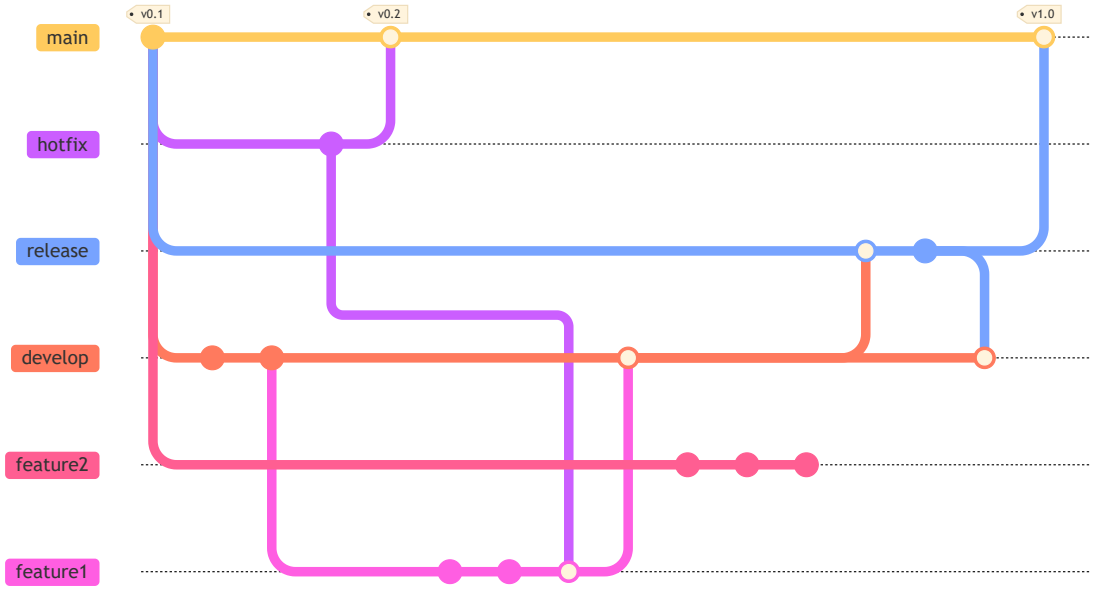

Figure 3.3: Example of Git Flow

- **Main Branch**:
  Main branch serves the same purpose as the master branch in GitHub Flow. It contains production ready code that can be released. Here, in figure 3.3, the main branch commits are tagged with their respective version numbers.

- **Develop Branch**:
  This branch is created at the start of a project and contains pre-production code. This branch is maintained throughout the project and is used to merge feature branches. The develop branch is merged to the release branch when the code is ready for release.

- **Feature Branch**:
  This branch has the same responsibility as the feature branch in GitHub Flow. After the feature is implemented, the branch is merged with the develop branch.

- **Release Branch**:
  This branch contains the code that is ready for release. It is created from the develop branch and is merged back to the main branch after the release.
- **Hotfix Branch**:
  This branch is used to quickly address critical bugs in the production code. It is created from the master branch.

Git Flow is a more complex branching model compared to GitHub Flow. It is suitable for large projects with scheduled releases. It provides a more organized approach to managing the codebase and ensures that the code is stable and reliable before it is released to production. Due to the creation of several branches, developers can work parallel on different features without affecting the main branch.

3.4 Code Quality

Code quality is a critical aspect of software development that directly impacts the maintainability, reliability, and performance of the software. High-quality code is easier to understand, test, and modify, which is essential for the long-term success of any software project [5]. In the context of this thesis, ensuring code quality is particularly important for several reasons:

- **Maintainability**:
  High-quality code is well-structured and well-documented, making it easier for developers to understand and maintain. This is crucial for the continuous integration and continuous deployment (CI/CD) pipelines, as it ensures that the codebase remains manageable and scalable over time.
- **Reliability**:
  Code quality directly affects the reliability of the software. Well-written code is less prone to bugs and errors, which reduces the likelihood of failures during the testing and deployment phases. This is particularly important in a DevOps environment, where the goal is to deliver reliable software quickly and efficiently.
- **Efficiency**:
  High-quality code is optimized for performance, which can lead to faster execution times and more efficient use of resources. This is important for the automation processes described in this thesis, as it ensures that the testing and deployment pipelines run smoothly and efficiently.
- **Collaboration**:
  In a collaborative environment, such as GitHub, high-quality code is essential for effective teamwork. Clear, well-documented code allows multiple contributors to work on the same project without confusion or conflicts, which is a key aspect of successful CI practices.

In summary, maintaining high code quality is essential for the success of the automation processes and the overall framework described in this thesis. It ensures that the software is maintainable, reliable, efficient, and conducive to collaboration, all of which are critical for achieving the goals of Continuous Integration and Continuous Deployment.

Chapter 4

Creation of Framework

4.1 Introduction

4.1.1 What is a Framework?

A framework is a pre-built structure that provides a foundation for developing applications [3]. It includes libraries, tools, and best practices that accelerate the development process. Frameworks serve as templates that can be customized to meet project requirements. They allow developers to work on the core, i.e, the application of logic, rather than worrying about the underlying structure. Many frameworks are open-source and are easily available. Developers can also contribute to the framework by adding new features or fixing bugs.

It is crucial to first understand the project requirements and determine which programming language and corresponding framework best suit those needs. Each framework is designed for a specific purpose and offers unique features. Having a fundamental understanding of the chosen programming language is essential for effectively working with the framework. Popular frameworks used today include Django, Flask, Angular, React, PyTorch, TensorFlow, and more. These frameworks empower developers to create robust and feature-rich applications.

4.1.2 Why is a Framework used?

Developing code from scratch can be a tedious and error-prone task. Clean, well-tested, and bug-free code is essential, but achieving this can be challenging. Additionally, developers must adhere to coding standards and best practices to ensure code quality. Therefore, using frameworks that meet the requirements is a better choice. Frameworks simplify the development process, reduce errors and provide a general template that can be customized as needed. They also make it easier for others to understand your code, as they are likely familiar with the frameworks used. Frameworks offer several advantages, including:

- Simplified testing and debugging of code.
- Clean and understandable code.
- Reduced code redundancy within the project.
- Decreased project time and cost.
- Modifiable and extendable features and functionalities provided by the framework.

4.1.3 Libraries vs Frameworks

It is a common misconception that libraries and frameworks are the same. But they serve different purposes and have distinct characteristics.

Libraries:

- A library is a collection of pre-written code that developers can use to optimize tasks.
- It provides specific functionality that can be called upon when needed.
- Developers have control over the flow of the application and decide when to use the library.
- Some of the popular libraries include NumPy, Pandas, Matplotlib, and more.

Frameworks:

- A framework is a pre-built structure that provides a foundation for developing applications.
- It dictates the architecture and flow of the application.
- Developers must adhere to the structure and guidelines set by the framework.
- Examples of popular frameworks include Django, Flask, Angular, React, and more.

4.2 Overview of the Framework

Before, building a framework, it is essential to understand the requirements and objectives of the project. The main objective of this framework is to automate the process of running parametric simulations in Optislang, standardize and to verify the output files generated. The framework should be user-friendly, easy to use, and provide detailed error logs in case of any issues.

After understanding the requirements, the next step is to design the framework. The framework should be designed in such a way that it is scalable, modular, and easy to maintain. It should also be flexible enough to accommodate future changes and updates.

This framework is constructed using classes, functions, and libraries such as `Pandas`[1], `NumPy`[2], and other built-in Python modules. Figure 4.1 provides an overview of the structure and functionality of the framework.

[1] `https://pandas.pydata.org`
[2] `https://numpy.org`

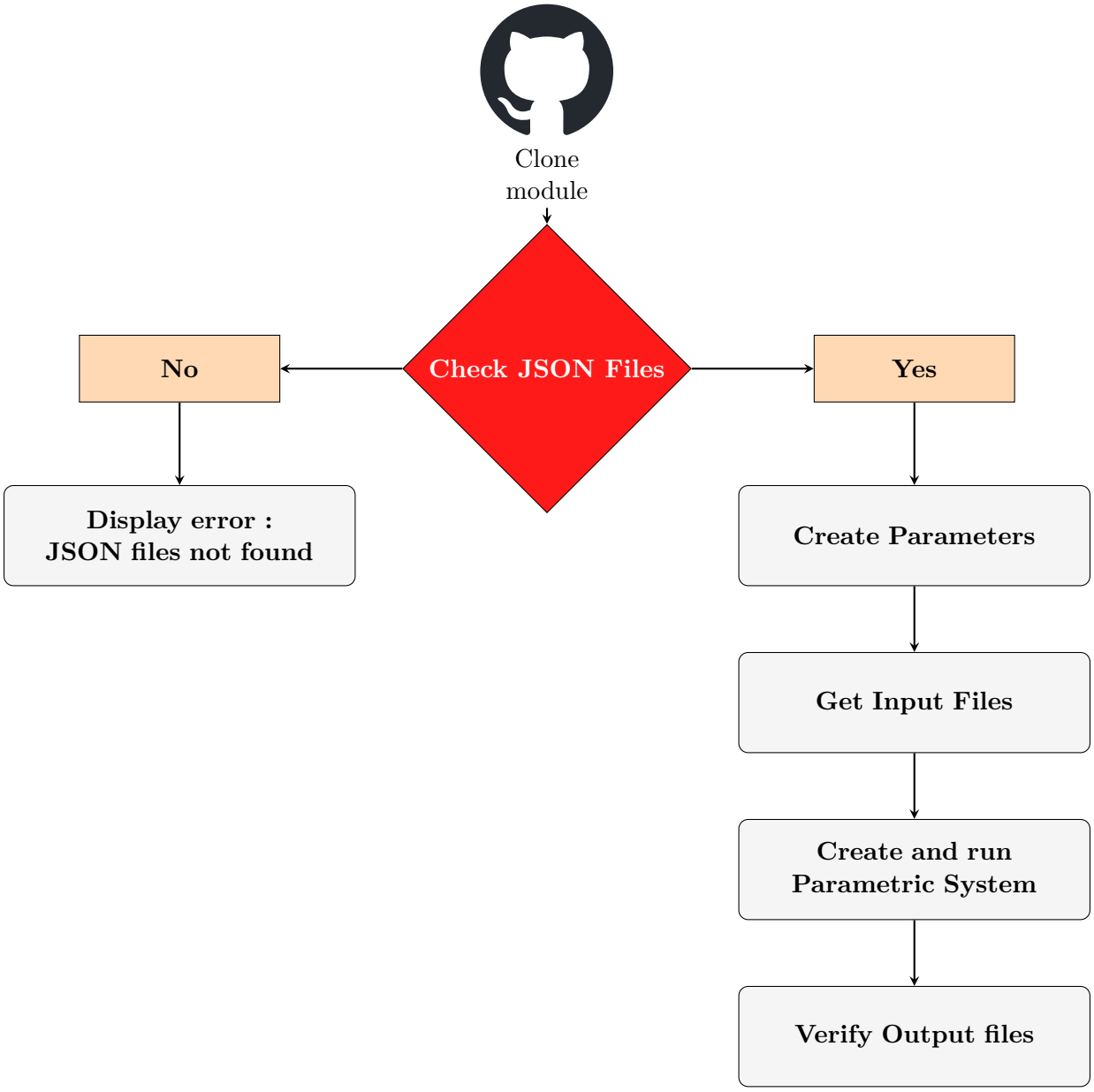

Figure 4.1: Flowchart of the framework

Let us understand the working of the framework in detail. The framework is mainly built using Python and uses Optislang's Python API to create and run the parametric simulations. The primary requirement for the framework to run is to have module present. Therefore, the first step is to clone the required module from the specific repository and branch from GitHub. These serve us as the main arguments needed to run the framework. After cloning, the framework checks for `module_config.json` and `parameters.json` files. These JSON files are crucial to be present in the module as they contain the information required to run the parametric simulations automatically. The `module_config.json` file contains information of the module like description of the module,

name of the main script containing the algorithm, type of framework, input and output files and their properties.These data are important to create, run and verify the parametric system generated. The `parameters.json` file contains information about the parameters required to be set as input in the parametric system.

At this stage, a decision is implemented. If the JSON files are present, the framework proceeds to further steps. If the JSON files are not found, the framework comes to a halt and displays an error message being shown to the user. Figure 4.1 shows the implemetation of the error message. This function is present inside the class `ParametricSystem`.

```
def json_files_log(self):
    try:
        if not self.get_module_config_path().exists():
            raise FileNotFoundError(
                f"{self.get_module_config_path()} does not exist"
            )
    except FileNotFoundError as e:
        print(
            f"{e} \nPlease ensure {self.get_module_config_path()} exists and re-run"
        )
    except Exception as e:
        print(e)
    try:
        if not self.get_parameters_path().exists():
            raise FileNotFoundError(f"{self.get_parameters_path()} does not exist")
    except FileNotFoundError as e:
        print(f"{e} \nPlease ensure {self.get_parameters_path} exists and re-run")
    except Exception as e:
        print(e)
```

Code 4.1: Function to verify existance of JSON files

If the framework identifies the JSON files, it proceeds to the next step of generating the necessary parameters for running the parametric system. These parameters are generated in the form of a `csv` file and a Python file. These files are fed as input to the parametric system.

The next step is to provide input files which are required by the parametric system in order to execute the simulations. These input files are retrieved from a cloud storage using API calls. A more detailed explanation of this process is provided in Section 4.6.

After the input files are retrieved, the framework then needs to create the parametric system. This should be achieved without the user's intervention, i.e, automatically. At this stage, we will be using the Python interpreter provided by Optislang as it includes the necessary libraries and functions to create and run the parametric system. Another terminal pops up which displays the progress of the execution of the system.

Since the whole process in the framework is automated, we need to ensure that the files generated by the parametric system are correct and are produced as expected. This is done by the function `verify_output_files` present in the class `ParametricSystem`.

4.3 Directory Structure

Before discussing the implementation of the framework, let us first understand how the files are structured within the framework. The directory structure is depicted in Figure 4.2. The framework

consists of the following files and directories:

- **`moo_framework_workflow.yaml`**:
  This file contains the workflow for running the framework automatically. It is written in YAML and is later used inside GitHub Actions.

- **`src`**:
  This directory contains the source code of the framework. It consists of files which are used to create and run the parametric system in Optislang. It also consists of some helper functions which are used to build complex functions and classes for the creation of framework. We will discuss more about these files in section .

- **`tests`**:
  This directory contains test cases for the framework. Section explains the test cases in detail.

- **`main.py`**:
  This python file calls the files which are responsible for the framework creation from `src` directory . It is the main file for running the framework.

- **`requirements.txt`**:
  This file contains the list of libraries which helps in running the framework. It is important to install these libraries before running the framework.

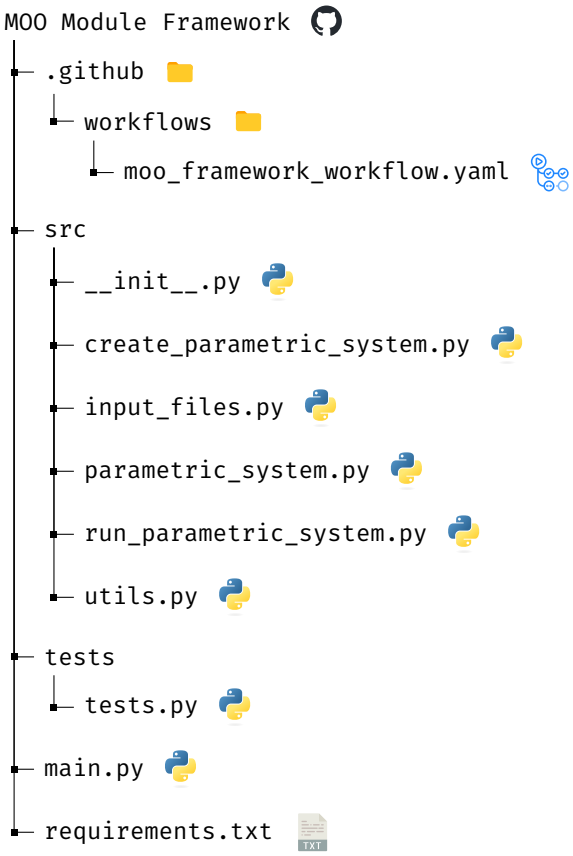

Figure 4.2: Directory structure of the framework

4.4 Testing of Framework

While building the framework, it is also essential to test the framework to ensure that it is working as expected. One way to do is to include breakpoints, add print statements, and debug the code. However, this method is not efficient when the codebase is huge. Therefore, another way is to write unit cases for the framework.

Unit testing is a software testing method that involves testing a small unit of code, typically a function or method. They are crucial part of the development process as they help in identifying bugs and errors early in the development cycle. Python has two frameworks for unit testing, `unittest` and `pytest`. I have implemented `unittest` for testing since it is part of the Python's standard library. Here, the unit tests can be found in the `tests` directory. The file `tests.py` contains the test cases for the framework. Unit tests generally should cover the following aspects:

- **Unit tests**:
  This is used to test the functionalities of each individual methods and functions.
- **Integration tests**:
  These tests are implemented to verify the interaction of units, modules or components of an application .
- **Boundary tests**:

This ensures to check the edge cases of the functions. For example, if the provided input is an empty string, the function should return an error message.

- **Negative tests**:
  To check if the function handles incorrect input properly, negative tests are implemented. An example for this would be to handle if the argument is of a different data type than expected.

Figure 4.2 shows an example of a unit test implemented in Python to test the existence of JSON files.

```python
class TestParametricSystem(unittest.TestCase):
    def setUp(self) -> None:
        self.parametric_system = ParametricSystem('MOO_M_ARRHENIUS','MOO-1355_py_framework_poc')
        self.cwd = os.getcwd()

    def get_module_name(self):
        for file_name in os.listdir(self.cwd):
            if file_name.startswith('MOO'):
                return Path(self.cwd,file_name)
        return None

    def test_get_module_config_path(self):
        self.assertIsNotNone(self.get_module_name(), 'Module folder not found.')
        expected_path = (Path(self.get_module_name(),'01_Specification', 'module_config.json'))
        actual_path = self.parametric_system.get_module_config_path()
        self.assertEqual(expected_path, actual_path)
```

Code 4.2: Example of a unit test

Firstly, we create a class `TestParametricSystem` which inherits from `unittest.TestCase`. To avoid initialization of the same variables in each test case, we take advantage of the **`setUp()`** method. Here, we create an object an initialize the arguments for the class `ParametricSystem`. The functions in the unit tests needs to start with the prefix `test`. This convention is used to identify the function which the test cases. For example, in Figure 4.2,the function `test_get_module_config_path` recognizes that it is a test case whereas the function `get_module_name` is a helper function and not a test case. In this example the function `test_get_module_config_path` is responsible to check if the path of the folder containing the JSON file is correct. To ensure this, we use the function `assertEqual` which checks if the expected path is equal to the actual path. If the paths are equal, the test case passes, else it fails. `unittest` provides several other functions to test the code.

4.5 Execution of Framework

To execute the framework, the user needs to run the file `main.py`. This file is the main file to run the framework. This script calls the functions from the class `ParametricSystem` and runs the module. Figure 4.3 shows the execution of the framework.

```python
from src import ParametricSystem

module_name = "MOO_M_ARRHENIUS"
module_branch_name = 'MOO-1355_py_framework_poc'

```

```python
def main():
    system = ParametricSystem(module_name, module_branch_name)
    system.clone_module()
    if (
        system.get_module_config_path().exists() & system.get_parameters_path().exists()
    ) == True:
        system.run_module()
        system.verify_output_file()

    else:
        system.json_files_log()

if __name__ == "__main__":
    main()
```

Code 4.3: Execution of framework using `main.py`

In code snippet 4.3, an example of cloning the module `MOO_M_ARRHENIUS` is shown. An instance of the class `ParametricSystem` is created with the arguments `module_name` and `module_branch_name`. Then, the check for the `JSON` files is done. The verifying of the files is done outside the function `run_module()` as we do need to verify the existence of the files before running the module. Once the existence of the files are verified, the module is being run and after the successful execution in Optislang, it displays the status of the output files.

This framework also contains some external libraries which are required for the functioning of the framework. Therefore, `requirements.txt` file contains the list of libraries required to install before running the framework in the virtual machine.

4.6 Retrieve input files for Framework

4.6.1 Motivation

Initially, the idea was to mock the input files required for the parametric system. The data required for mocking the input files were present in the `module_config.json`. But, in some modules. the input files were very complex. For example, in the module `ROM SOLVER`, there are input files which are matrix files which have 250,000 lines of data. Mocking these huge files is time consuming, inefficient and not a way to standardize the framework. Therefore, the idea was to save these files in a cloud storage and retrieve them later when required. This improves the efficiency of the framework and also helps in standardizing it.

In the following section, we will discuss the practices, tools and implementation to retrieve the input files from OpenShift using FastAPI.

4.6.2 Docker

Docker is platform used for developing, shipping and running applications effectively. It is a containerization platform which packages the application and all its dependencies together in the form of containers. To run Docker in a local machine or in a cloud, we need to install it. Docker is preferred during development and deployment as it is lightweight, portable and scalable. This can run anywhere, regardless of the operating system. A Dockerfile contains instructions to build a

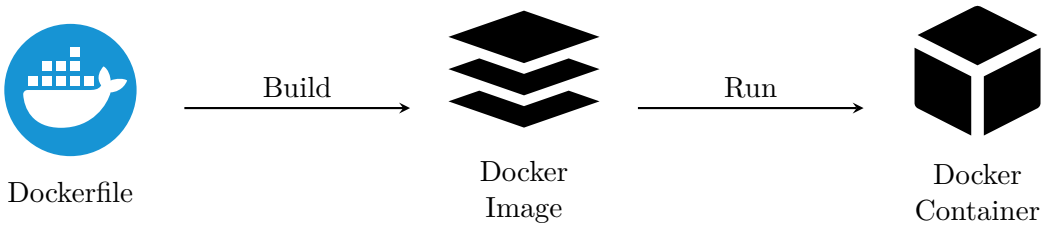

Figure 4.3: Creation and deployment of Docker image

Docker image. It is basically a blueprint to built the Docker image. A Docker image is a lightweight, standalone, executable package that includes everything needed to run the application. WHen the image is run, it becomes a container. Figure 4.3 shows the creation and deployment of a Docker image.

The main difference between a docker container and a virtual machine is that a container shares the host's kernel which makes it more lightweight and faster. Whereas, a virtual machine reserves some place in the host's memory for the guest operating system, libraries and applications which makes it slower and heavier.

4.6.3 OpenShift

OpenShift is a Kubernetes platform which is used to deploy and manage containerized applications. It allows developers to build, deploy and scale applications on the cloud. OpenShift is preferred as it is an effective and flexible platform for container orchestration and deployment of applications. The advantage of using OpenShift is that it provides a web console and a command line interface to manage the applications. It also provides key metrics and logs for monitoring CPU, memory, bandwidth usage, latency of network, container health and many more.

4.6.4 FastAPI

What is API?

API is a collection of protocols and tools which allows different software applications to communicate with each other. Let us consider an scenario to understand the concept of API. Suppose, you are in a restaurant and you want to order food. You call the waiter and tell him the food you want to order. The waiter takes the order to the kitchen, gives it to the chef and the chef prepares the food. Once the food is ready, the waiter brings the food to you.

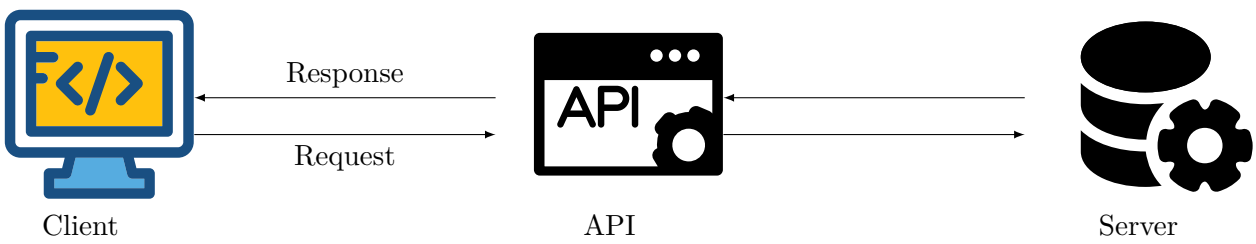

Figure 4.4: Working of an API

Similar to the example, an API takes the request from the client, processes it and sends the

response to the server. The server processes the request and sends the required response back to the client. To retrieve the required data from the server, the client needs to send the request to the server via the required API endpoint.

To communicate to the server, API uses HTTP methods like `GET`, `POST`, `PUT`, `DELETE` etc. The `GET` method is used to retrieve data from the server, `POST` is used to send data to the server, `PUT` is used to update the data and `DELETE` is used to delete the data from the server.

FastAPI

FastAPI is a modern, fast web framework which is relatively fast and used for building APIs using Python.

4.6.5 Implementation

After understanding the practices and tools, we will discuss the implementation of retrieving the input files from OpenShift using FastAPI. To access the input files remotely, we need to upload a docker image to OpenShift. The docker image contains the implementation of FastAPI and the input files required for the parametric system. To access the input files, we need to create endpoints in FastAPI. Figure 4.5 shows the directory structure of the input files storage.

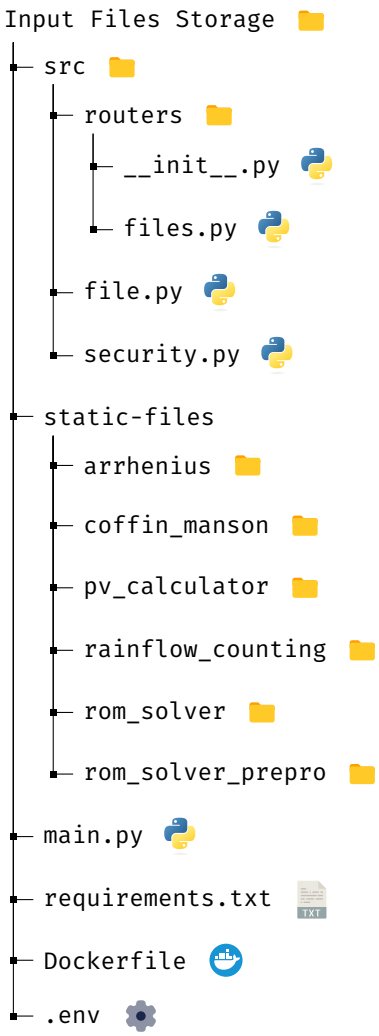

Figure 4.5: Overview of implementation of retrieving input files

`src` contains the scripts needed to establish a connection with FastAPI. FastAPI contains two `GET` endpoints which helps in the retrieval of input files. Endpoint `getInputFiles` returns a dictionary containing the key-pair value of module name and the corresponding location of the input files. The directory `static-files` contains the input files which is used to retrieve the input files. The endpoint `download` takes the location of a file as an argument and returns the file to the user.

To access the input files in a cloud storage, we build a docker image consisting of the FastAPI implementation and the input files. Figure 4.4 shows the implementation of the Dockerfile.

```
FROM python:3.10
WORKDIR /code

COPY ./requirements.txt /code/
COPY ./static-files/ /code/static-files/

RUN ["python", "-m", "pip", "install", "-r", "requirements.txt"]

EXPOSE 7000

```

```
RUN mkdir -p ./temp && \
    chgrp -R 0 ./temp && \
    chmod -R g=u ./temp

COPY main.py /code/
COPY ./src/ /code/src/
COPY .env /code/.env
ENTRYPOINT ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "7000"]
```

Code 4.4: Implementation of Dockerfile

The Dockerfile contains the instructions to build the Docker image. Here, the Dockerfile copies the folder containing the input files and FastAPI. It installs the required libraries using the `requirements.txt` file. Later, it exposes the port 7000 and runs the FastAPI using the uvicorn server. The `.env` file contains the environment variables consisting of username and password to access the API.

After building the Docker image, it is deployed in a pod in OpenShift. Using OpenShift, we can access the input files remotely via API calls. This is later called during the first step of creating the parametric system.

Chapter 5

Implementation of CI Pipeline

5.1 Introduction

CI/CD is all about delivering frequent, incremental changes to the application so that you can get regular feedback. But faster delivery of the code should not affect the quality of the code. Therefore, a reliable and efficient automated testing system is required to ensure that the code is working as expected.

Automated testing/Continuous Testing is one of the cornerstone of CI/CD. It is the process of executing automated tests as part of the software delivery pipeline to obtain immediate feedback on the software and its quality [9]. These scripts eliminate the need for regular manual testing and validate the source code sequentially. By using continuous testing, developers can identify and fix bugs early in the development process, ensuring the quality of the code. During the development of the framework, I have implemented a CI pipeline using GitHub Actions. This pipeline is responsible for running the framework on every commit to the repository. A detailed explanation of the pipeline is explained in the following sections.

5.2 Version Control Strategy

As discussed in Section 3.3, different strategies can be implemented for organizing a project. The modules present in our organization uses the Git Flow strategy. Here, the two long running branches, `main` and `develop` are used and feature branches are created from the `develop` to tackle the new features. While implementing the framework, I have used the GitHub Flow strategy as it is simple and easy for a single developer.

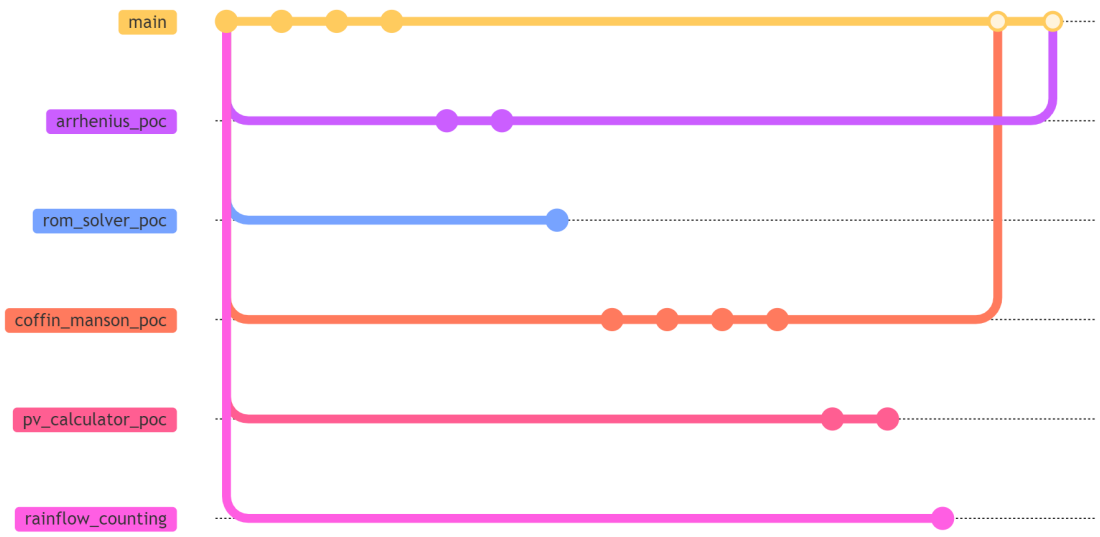

Figure 5.1: Branching strategy for the framework

From the figure 5.1, the `main` branch contains the stable code required for running the modules. Since, the framework needs to be tested and implemented for all the modules, develop branches are created from the `main` branch. Each of the develop branch contains the code for a specific module. This is beneficial as the code for each module is isolated and the main source code is not affected. Once the code is tested and implemented, the code is merged to the `main` branch. During any commit to the develop branch, the CI pipeline is triggered which tests the framework.

5.3 GitHub Actions

Since the modules created by the developers and the framework are being hosted on our GitHub organization, the ideal choice for implementing the CI pipeline was GitHub Actions. GitHub Actions is a CI/CD service that makes it easy to automate tasks based on events that occur in the GitHub repository[6]. For example, a trigger can be set up to run the run the pipeline based on the events like push, merge, pull request, etc. Figure 5.2 shown below explains the software development lifecycle and the usage of CI/CD in the process.

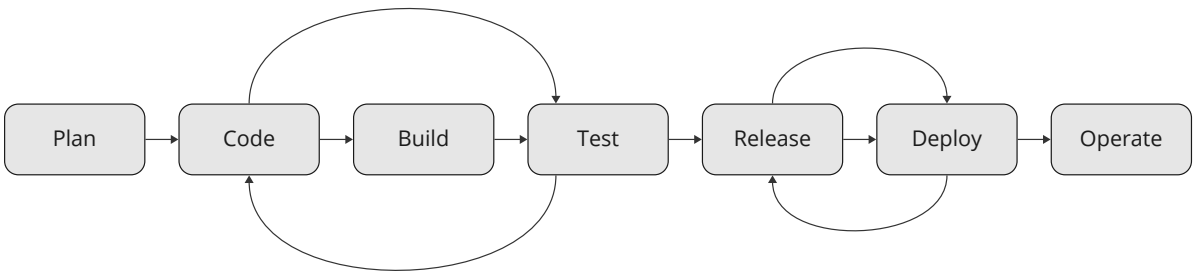

Figure 5.2: Software development lifecycle

GitHub Actions is implemented after step 2 in the figure 5.2. After updating the codebase,

the developers pushes the code to the code to the repository which triggers the CI pipeline. This pipeline executes a script which validates the developer's code. If any exception is raised, the pipeline fails, preventing the code from being merged and sends a notification to the developer. Otherwise, the build is successful and the code is merged to the main branch.

5.3.1 Components of GitHub Actions

Workflows

This pipeline is implemented in a `.yml`/`.yaml` file which is a superset of JSON. This workflow file contains the configuration of the pipeline. This file is usually stored in the `.github/workflows` directory in the repository.

```yaml
name: MOO Module Framework

on:
  push:
    branches:
      - arrhenius_poc

jobs:
  Run_MOO_module_framework:
    runs-on: [self-hosted, MOO_WINDOWS]

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: "3.12"

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          python install -r requirements.txt

      - name: Run main.py
        run: python main.py
```

Code 5.1: Example of a GitHub Actions workflow file

Events

Events are the triggers which starts the workflow. We can assign events from GitHub like push, pull to start the workflow. We can also assign the trigger to a specific branch or tag. In the example shown in Figure 5.1, the workflow is setup to trigger on every push to the `arrhenius_poc` branch.

Jobs

Jobs are a set of defined tasks which are executed when the pipeline is triggered. Each job is assigned to a runner which is a virtual machine that runs the job. Jobs contains a set of steps which are executed sequentially. If any step fails, the job is marked as failed and the pipeline comes to a halt. In the example shown in Figure 5.1, the job `Run_MOO_module_framework` is assigned to a self-hosted runner with the label `MOO_WINDOWS`. This job contains 4 steps which are executed sequentially.

Actions

Actions is a GitHub Actions feature that contains specific custom program performing a task. These actions are reusable and can be used in multiple workflows. It helps in reducing the number of lines of code in the workflow file. From the figure 5.1, the actions `actions/checkout@v2` and `actions/setup-python@v2` are used to checkout the repository and setup the python environment respectively in the runner. There are several actions available in the GitHub marketplace which can be used in the workflow.

Runners

Runners are servers which assist in the running the workflow. GitHub provides Ubuntu Linux, Microsoft Windows and Ubuntu runners which are hosted by GitHub. Apart from these, we can host our own runners which can be used to run the workflow. In our organization, the workflow is usually ran in one of our lab computers which is setup as a self-hosted runner. To identify the runner, a label is assigned to the runner, so that the workflow can be run in a desired runner. In the example shown in Figure 5.1, the job is assigned to a self-hosted runner with the label `MOO_WINDOWS`. The disadvantage of runners in GitHub Actions is that the runners can only run one job at a time. Additionally, needs to be online all the time to execute the workflow.

5.4 Summary

The implementation of the CI pipeline is a crucial step in the testing of modules using the framework. The pipeline is set to trigger on every push to a specific branch. On triggering, the pipeline gets the files from the repository, sets up the python environment, installs the dependencies and runs the `main.py`, which calls the framework. If any exception is raised during steps, the pipeline fails and sends a notification to the developer. Otherwise, the pipeline is marked as successful and the code is merged to the branch.

Chapter 6

Summary

6.1 Conclusion

6.2 Future Work

Appendix A

Appendix

A.1 Implementation of the Framework

In this section, we will briefly discuss the working and functioning of each of the files present in the framework. Each of the files are crucial and responsible for the proper functioning and execution of the framework.

A.1.1 Utility files

During creation of a framework, it is common to repeat functionalities in different parts of the codebase. But, it is frowned upon in software development as it leads to code redundancy and makes the codebase difficult to maintain. Therefore, it is ideal to store the repeated functionalities in a separate file and import them when required. A utility file refers to a Python source file which contains utility functions or helper functions. These functions are built in a way that they can provide common functionalities or operate specific tasks which can be reused in different parts of the codebase.

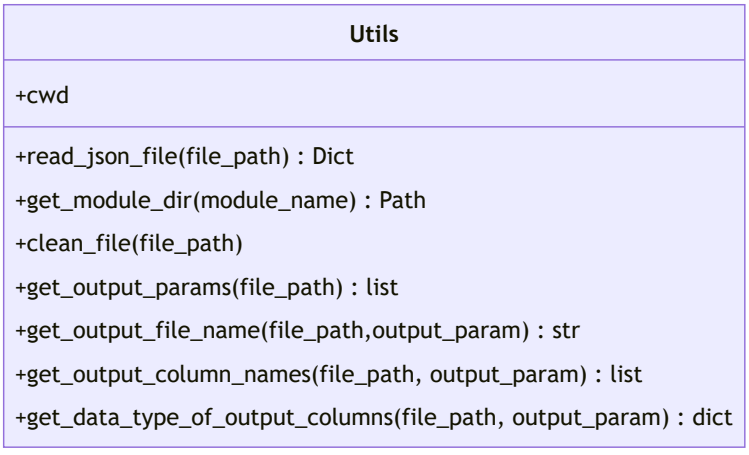

Figure A.1: Overview of functions in `utils.py`

`utils.py` is a helper file which mainly consists of functions which are later used to build complex functions for the creation of the framework. It consists of functions like `read_json_file` which reads a JSON file and returns the data in the form of dictionary, get the path of a module, get column names of a `csv` file, get data type of a column in a `csv` file and many more. Figure A.1 shows the functions present in the `utils.py` file.

A.1.2 Creation and Execution of Parametric System

Since the main objective of the framework is to automate the process of running parametric simulations in Optislang, it is essential to create the parametric system automatically. The script `create_parametric_system.py` assists in creating the parametric system without the help of Optislang's GUI. The parametric system is created based on the information present in `module_config.json`. The JSON file consists of key-value pairs containing crucial information for creating the system, like name of the main script containing the algorithm, type of module to build, i.e, Python or MATLAB.

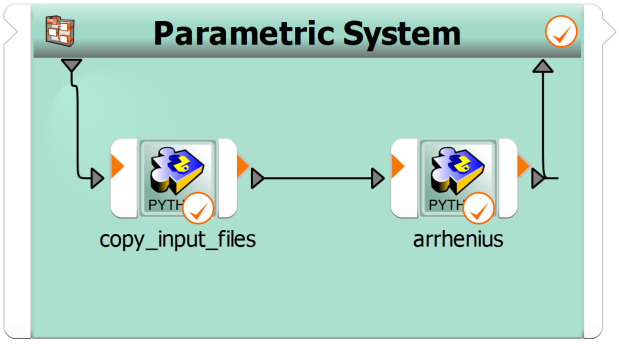

Figure A.2: Example of a parametric system with copy actor

Optislang takes in different functions to create a system depending on the type of actor. Therefore, type of module is required. Since the framework is automated, the process of inputting the requried files also needs to be automated. The script `input_files.py` is responsible for retrieving the input files from OpenShift. A detailed explanation of retrieving input files from OpenShift is provided in Section After providing the input files, the parametric system is created and is shown in Figure A.2.

The next step is to run the created parametric system. `run_parametric_system.py` runs the newly created parametric system. This system is being run inside the Python interpreter provided by Optislang. These scripts are being called and ran in an orchestrated manner inside the class `ParametricSystem`.

A.1.3 Framework Orchestration

For better code organization, reusability and encapsulation, the framework is built using classes. Classes are the pillar of object-oriented programming. There are many advantages of using classes.

- Code is maintained and organized in a better way.
- Data can be encapsulated and hidden from the user.
- Code can be reused and extended easily.

Creation of framework is possible by using the class `ParametricSystem`. Since the creation of framework is tedious and complex, we make use of `utils.py` which assists in building the class. Figure A.3 shows the functions present in the class `ParametricSystem`.

The class takes in the arguments `module_name` and `module_branch` which are required to clone the module from GitHub. The following provides a high-level overview of the key functions used

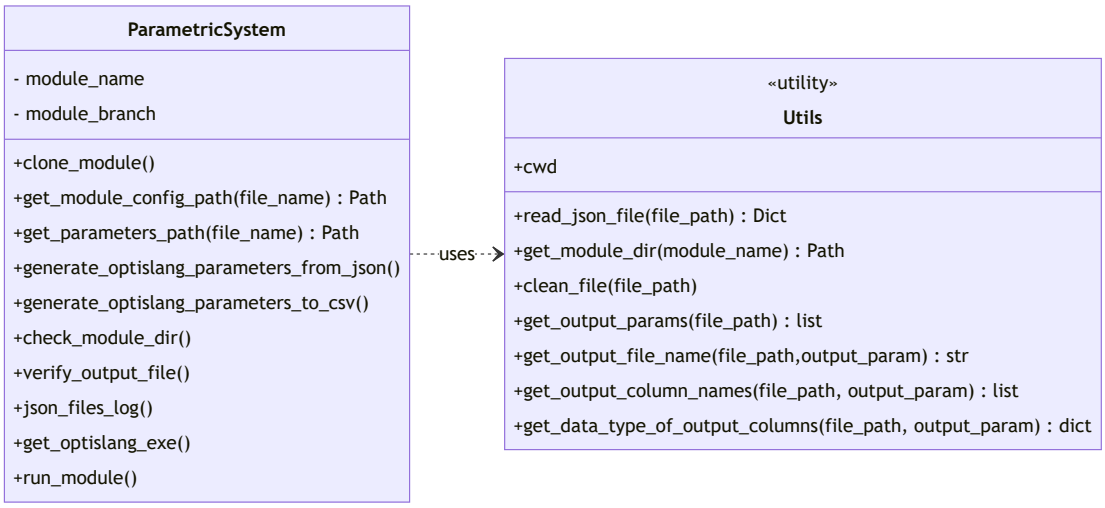

Figure A.3: Overview of the class `ParametricSystem`

in the framework. These functions are essential for automating the process of testing standalone modules in Optislang.

- **`clone_module()`**: This function is responsible for cloning a module from a GitHub repository. It ensures that any existing module is deleted before cloning the new one, thereby maintaining a clean working environment.
- **`get_module_config_path()`**: This function retrieves the absolute path of the `module_config.json` file. It takes the file name as an argument and returns its absolute path, facilitating easy access to the configuration file.
- **`get_parameters_path()`**: Similar to `get_module_config_path()`, this function returns the absolute path of the `parameters.json` file. It ensures that the framework can locate and use the parameters file efficiently.
- **`generate_optislang_parameters_to_csv()`**: This function converts the parameters from the `parameters.json` file into a CSV format required by the parametric system. The resulting CSV file is saved in the current working directory as `optislang_actor_parameters.csv`.
- **`generate_optislang_parameters_from_json()`**: This function generates the parameters needed for creating the parametric system from the `parameters.json` file. The parameters are saved in a Python file, `optislang_parameters.py`, which is later used in the system creation process.
- **`check_module_dir()`**: This function creates a mock directory structure to accommodate hardcoded paths within the modules. It ensures that the necessary directories and files are in place, allowing the modules to function correctly.
- **`verify_output_files()`**: This function verifies the existence and correctness of the output files generated by the parametric system. It is used after the system is created to ensure that the output meets the expected criteria.

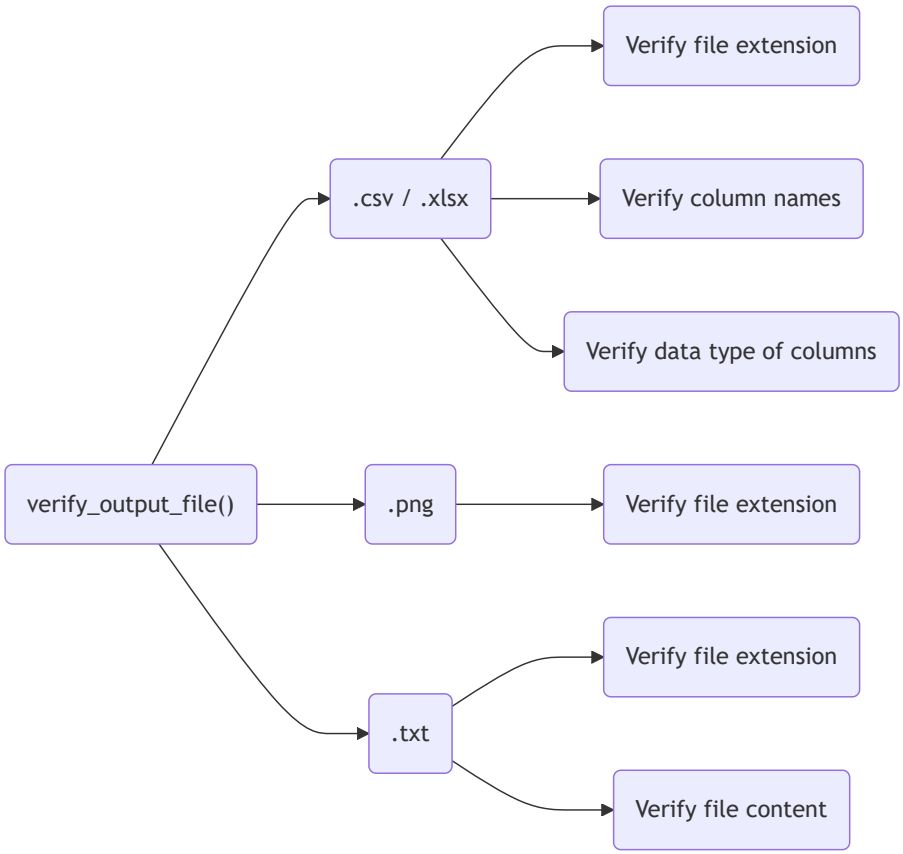

Figure A.4: Working of `verify_output_files()` function

To verify the output, we first retrieve the verification data from `module_config.json`, which includes properties like column names, file names, formats, and data types. The function iterates through the output folder to check the presence of all output files. Using the Pandas library, it reads `csv` files to verify column names and data types. If all checks pass, a success message is displayed; otherwise, an error message specifies the issue.

For non-`csv` files, the function also verifies the presence and correctness of `.txt` and `.png` files. For `.txt` files, it additionally checks if the file is not empty.

Bibliography

[1] Ansys optislang. https://www.ansys.com/products/connect/ansys-optislang. Online; Accessed on 10/05/2024.

[2] Devops - the complete guide for 2023. https://www.openxcell.com/devops. Online; Accessed on 15/08/2024.

[3] What is a framework? https://www.geeksforgeeks.org/what-is-a-framework/. Online; Accessed on 13/09/2024.

[4] S.A.I.B.S. Arachchi and Indika Perera. Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)*, pages 156–161, May 2018.

[5] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, Nov 2014.

[6] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 420–431, May 2021.

[7] Mathias Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014.

[8] E Naresh, S V N Murthy, N. Sreenivasa, Seshaiah Merikapudi, and C R Rakhi Krishna. Continuous integration, testing deployment and delivery in devops. In *2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*, volume 1, pages 1–4, April 2024.

[9] S. Nithin, Hamsalekha S J, and Parvathy S. Enhancing the automotive software test environment using continuous integration and validation pipeline. In *2023 Innovations in Power and Advanced Computing Technologies (i-PACT)*, pages 1–6, Dec 2023.

[10] Pulasthi Perera, Roshali Silva, and Indika Perera. Improve software quality through practicing devops. In *2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)*, pages 1–6, Sep. 2017.