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 : Prof. Dr. Alexander Versch

Submission Date : 05/10/2024

DECLARATION

I declare that the work in this thesis, **"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 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 thesis to any other examination board.

Ludwigsburg, 05.10.2024

Place, 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 ..... Yet Another Markup Language

Abstract

System developers at Bosch create modules in MATLAB and Python to address specific use cases. These modules are integrated into Optislang, a software tool used for optimization analysis. For every new feature in the modules, developers must generate a new version and test it locally, a process that is both time-consuming and error-prone. This thesis proposes a solution to automate the testing of these modules. The modules are executed using a Python-based framework, which is integrated into a Continuous Integration (CI) pipeline on GitHub. This pipeline is triggered whenever a new commit is pushed to the repository, running the tests and providing feedback to the developers. By implementing this solution, developers receive immediate feedback and can focus more on developing new features.

Chapter 1

Introduction

1.1 Overview

At our department, system developers are responsible for creating modules considering specific use cases like power loss, thermal calculation and lifetime assessments. These modules are created in MATLAB and Python. To optimize the performance of these modules, optiSLang is used. optiSLang is a software tool that is used for optimization and robustness analysis. It contains algorithms to optimize the performance of the modules, verify the robustness of the modules and to perform sensitivity analysis.

To add new features to the modules, the developers need to update the current module, test it locally and then push the changes to the repository. This process is time-consuming and error-prone. To address this issue, this thesis proposes a framework which tests the module when a developer pushes the changes to the repository and notifies the developer if the module is not working as expected.

1.2 Objective

The objective of this thesis is to create a framework that test the modules created by the developers. The framework should be able to test the modules which are created in MATLAB and Python. Since the framework needs to be automated, it should run the module in the command line and not using the GUI. The framework should provide feedback to the developer regarding their latest changes to the module. Finally, the framework has to verify if the output files generated by the modules are present and are generated as expected.

1.3 Outline

In Chapter 2, we will discuss the basics of Multi Objective Optimization, its applications in the organization and the current problems faced by the developers.

Chapter 3 will discuss the methods and best practices implemented while creating the framework. These methods will help in creating a robust framework.

Chapter 4 delves into the creation of the framework. It will discuss the introduction and importance of the framework, a brief overview of the framework, the architecture and the execution. This chapter also explains the tools used to get the input files for the framework.

Chapter 5 will talk about the implementation of automated testing of the modules using a Continuous Integration pipeline.

Chapter 6 contains the results of the framework. It will discuss the results of the framework by running the tests on the modules.

Finally in Chapter 7, the thesis is concluded with a summary of the work done and the future work that can be done to improve the framework.

Chapter 2

Background

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. A detailed understanding of MOO is provided in the following sections.

Example: A car consists of many components like engine, body, wheels and many more which can be tweaked. In this example, 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

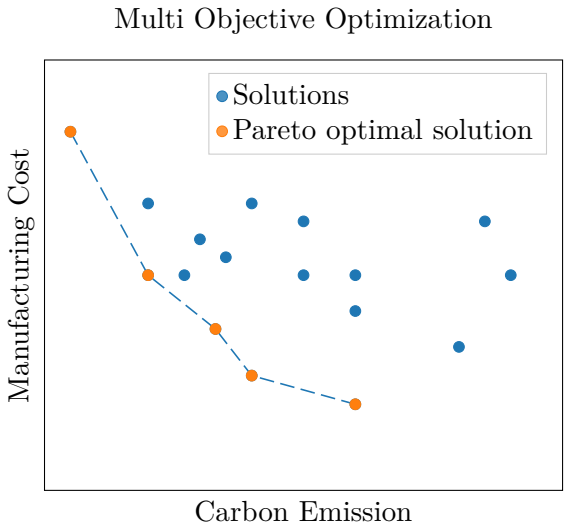

Figure 2.1: Example of MOO

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[12].

2.2 Difference between Multi Objective Optimization and Single Objective Optimization

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.

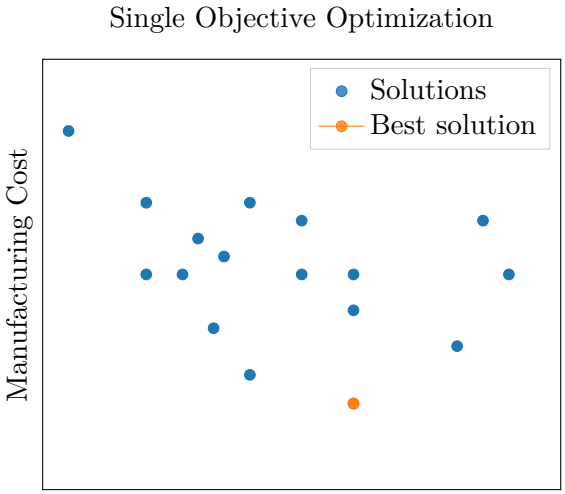

Figure 2.2: Example of SOO

In Figure 2.2, we have considered the same example given in Section 2.1. But, here, we are considering only **one objective**, minimizing the manufacturing cost. This is shown in the y-axis. The best solution is highlighted 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 Softwares used for Multi Objective Optimization

To calculate MOO, we need a software platform that can handle the complexity of the problem. Ansys Optislang [1] is one such software, 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 it. System developers design the modules and later integrate them into Optislang as a parametric system. Parametric system refers to a simulation model is defined using parameters which can be varied. By adjusting the parameters, the simulation model can be used to explore the behavior of the system under different conditions. Apart from MOO, Optislang can also be used for sensitivity analysis, robustness analysis, reliability analysis, and many more.

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 a specific issue. To document and collaborate with other system developers, every module is versioned and stored in a specific manner in a repository in GitHub Enterprise. Figure 2.3 shows us the structure of how each module is maintained in GitHub organization.

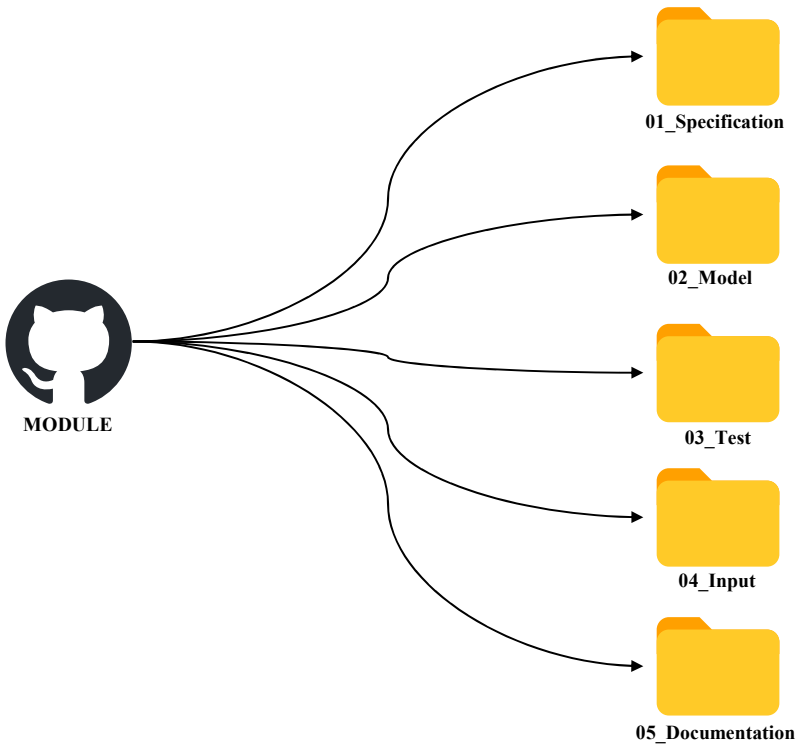

Figure 2.3: Structure of a module in GitHub Enterprise

- `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

System developers at Bosch are responsible for developing automatized workflows for power electronics products considering functional loads, reliability indications and many more. 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 shows an example of a workflow implemented in Optislang.

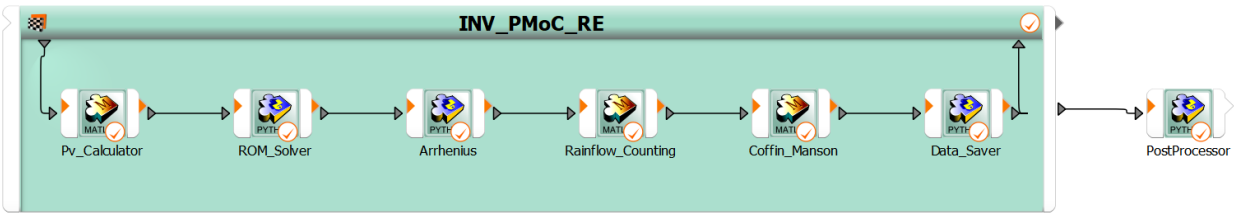

Figure 2.4: Example of a workflow in Optislang

In the above Figure 2.4, the workflow `INV_PMoC_RE` is responsible for providing fast, automatized and modular electro-thermal based reliability assessment and optimization for a Power Module on Cooler (PMoC). The workflow consists of several modules like `ARRHENIUS`, `ROM SOLVER`, `PV CALCULATOR` and many more.

For workflows and modules, our department follows a quality assurance process called **Model Readiness Level (MRL)**. Higher the readiness level, higher the maturity of the module or workflow. This quality assurance process ensures that the modules and workflows are not developed for a small development circle, but for a broader audience. It also ensures that the modules and workflows are well documented, tested and maintained.

2.5 Current Problems

There are several modules and workflows developed by the system developers at Bosch. While introducing a new feature or fixing a bug, the developer needs to test their newly implemented code locally. Later, they need to test their feature in Optislang to ensure the functionality of the module

or workflow. This process is time demanding. Some of the module or workflow tests are complex considering the number of parameters, which makes the testing process error prone. Currently, developers are spending a significant amount of time testing the modules and workflows. This is not a sustainable solution considering the fast-paced development in the automotive industry.

Every system developer or a team of system developers is responsible for developing the modules and workflows. Each module or workflow is developed in a specific way. Some of the older modules are maintained with different standards than the newer ones. This leads to a lack of standardization in the modules and workflows. This lack of standardization makes it difficult to maintain the modules and workflows effectively.

Considering the above problems, developing a digitalization strategy for automated development of a module or workflow is a key point to keep Bosch competitive in a constantly evolving market where a fast, reliable and sustainable product development is crucial.

Chapter 3

Related Work

3.1 Continuous Integration in Software Development

Continuous Integration (CI) is a software development practice that integrates code changes frequently and tests them automatically to detect issues early in the development process. The concept of CI was popularized by Martin Fowler in the early 2000s, emphasizing the importance of automating the build and test process to ensure code stability[14] . Jenkins[1] and Travis CI[2] are two of the most widely adopted CI tools that automate build processes and testing in a wide range of programming environments. These tools have laid the groundwork for integrating code quality checks, reducing effort of manual testing.

More recently, GitHub Actions[5] has gained popularity as a CI tool integrated directly within GitHub repositories, enabling developers to automate workflows triggered by events like code pushes or pull requests. Similar to these approaches, the framework developed in this thesis leverages GitHub Actions to automate testing of OptiSlang modules, ensuring real-time feedback to developers[18].

3.2 Automation in Simulation based Optimization Workflows

Automation of simulation-based optimization workflows has a significant focus in engineering domains. Industries like automotive, aerospace, and manufacturing rely heavily on simulation tools like MATLAB and Ansys to optimize designs. These tools are often integrated with optimization algorithms to automate the design process. The optimization algorithms are used to find the best design parameters that satisfy the design constraints and objectives. An article by Bucher et al.[13] discusses the use of Optislang for optimization of a non-linear system. In this study, the authors use design of experiments to generate a response surface model and later optimize the model using a non-linear optimization algorithm. But the study does not discuss the testing phase nor the automation of the workflow, which this thesis aims to address by automating the creation and testing of Optislang modules.

Piero Pezze et al.[24] have developed a pipeline for SBpipe[3], a software tool for automating repetitive tasks in model development and analysis. By using this pipeline, productivity and reliability during model development are increased. A case study by [15] introduced an approach to automate unit testing in SCADA[4] software. The study shows that by implementing automated

[1] https://www.jenkins.io/
[2] https://www.travis-ci.com
[3] https://sbpipe.readthedocs.io/en/latest/
[4] https://scada-international.com

testing framework, time was saved and the quality of the software was improved. [23], [25] and [26] have also discussed the importance of automated testing framework in their respective studies and their advantages in improving software quality. This thesis aims to extend these ideas by proposing a Python based framework for automated creating and testing of Optislang modules. By doing so, the framework will address the issues like modularization, automation and scalability in simulation-based optimization workflows.

3.3 Continuous Integration for Optimization tools

Optislang is a software tool developed by Dynardo GmbH which is widely used in the industry for optimization, robustness analysis and many more. It also allows integration with other simulation tools like MATLAB, Ansys, etc. to automate parametric studies and optimization workflows. However, few works have been done to automate the testing of Optislang modules.

An article by Mathworks [16] shows importance of using version control and automated testing for simulations. Another article by dSPACE [8] highlights the advantages of using CI for Hardware-in-the-loop (HIL) simulations. These studies emphasize the implementation of DevOps practices like version control, automated testing and CI in simulation-based workflows and the benefits of using these practices[9].

A similar approach is proposed in this thesis to automate the creation and testing modules in a CI pipeline.

3.4 Challenges and Advancements in Workflow Automation

Automation in CI pipelines, especially in simulation-based optimization workflows, poses several challenges. These include handling of input and parameters for simulations, managing dependencies across modules and providing real-time feedback to the developers. Many of the studies discussed above have addressed these challenges in a general DevOps concept, yet there is a limited focus on simulation specific tools like Optislang.

The framework developed in this thesis aims to address these challenges by providing a modular approach to create and test Optislang modules. The framework provides a way to automate the creation of Optislang modules and test them using GitHub Actions. The framework also provides a way to verify the generated files by the pipeline. This feature ensures that the generated files are correct and can be used for further analysis.

Chapter 4

Methods used for CI Framework

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 based framework is created to test the modules in an according manner. To run the framework, 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 provides feedback to the developer about the failure. The developer can then look into the issue and resolve it.

Many modern software engineering methods and practices are applied in this thesis. The following sections discuss these best practices and their implementation in the thesis.

4.1 DevOps

4.1.1 Introduction

In former days, 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 [20]. 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 immediate feedback.

4.1.2 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 4.1 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.

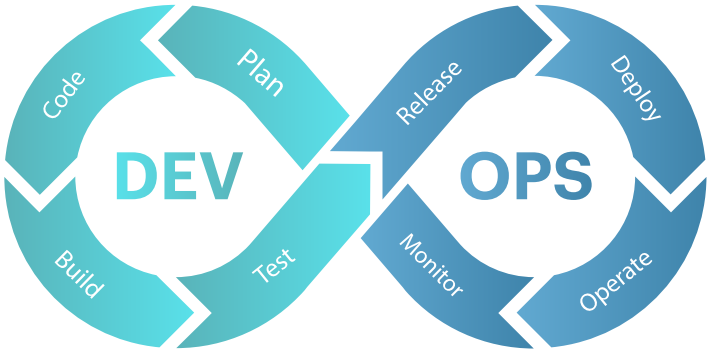

Figure 4.1: DevOps Lifecycle [2]

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 [22], 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.

4.2 Continuous Integration

Continuous Integration (CI) is the practice in software development where developers frequently commit their code changes to a shared repository. A CI pipeline can be applied to automate and streamline the testing process, leading to more robust, efficient and reliable software solutions [10].

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 and GitHub. A crucial practice of CI is to commit the code changes to the repository frequently [19]. By implementing this practice, the codebase is always up-to-date and the developers can easily identify and fix issues early in the development process. The main task of a CI pipeline to inform the team whether the changes made to the codebase are correct or not. There are several tools which provide this functionality. Some of the popular tools are Jenkins, Travis CI, Circle CI, GitHub Actions.

Issues can arise while configuring the CI pipeline. For example, the CI pipeline can fail due to incorrect configuration, or the changes made to the codebase can impact negatively to the end user. To overcome these issues, modern CI pipelines are equipped with features like logging, alerts, metrics, and notifications.

In this thesis, GitHub is used as the primary platform for storing the files and GitHub Actions is used to implement a CI pipeline. After the code changes are committed, the CI pipeline is triggered automatically. A detailed implementation of the CI pipeline is discussed in Chapter 6.

4.3 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, Mercurial 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.

4.3.1 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.

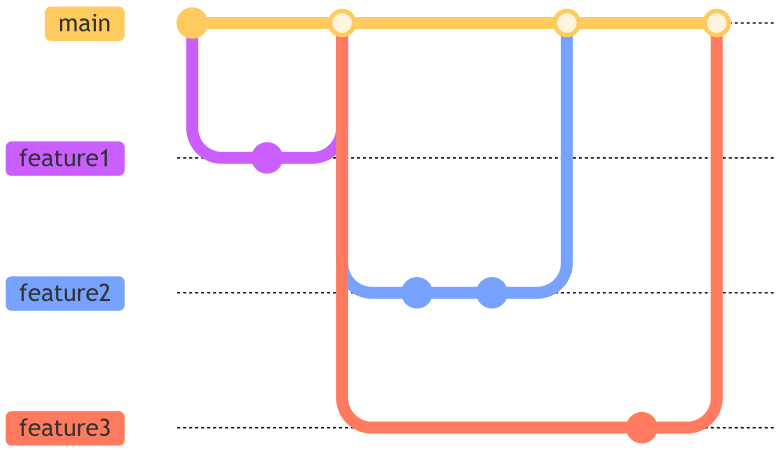

Figure 4.2: Example of GitHub Flow

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

- **Main Branch**:
  The 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 main branch. This is practiced to avoid conflicts with the code in the main branch. The developer works 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 4.2, `feature1`, `feature2` and `feature3` are the feature branches.

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

4.3.2 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.

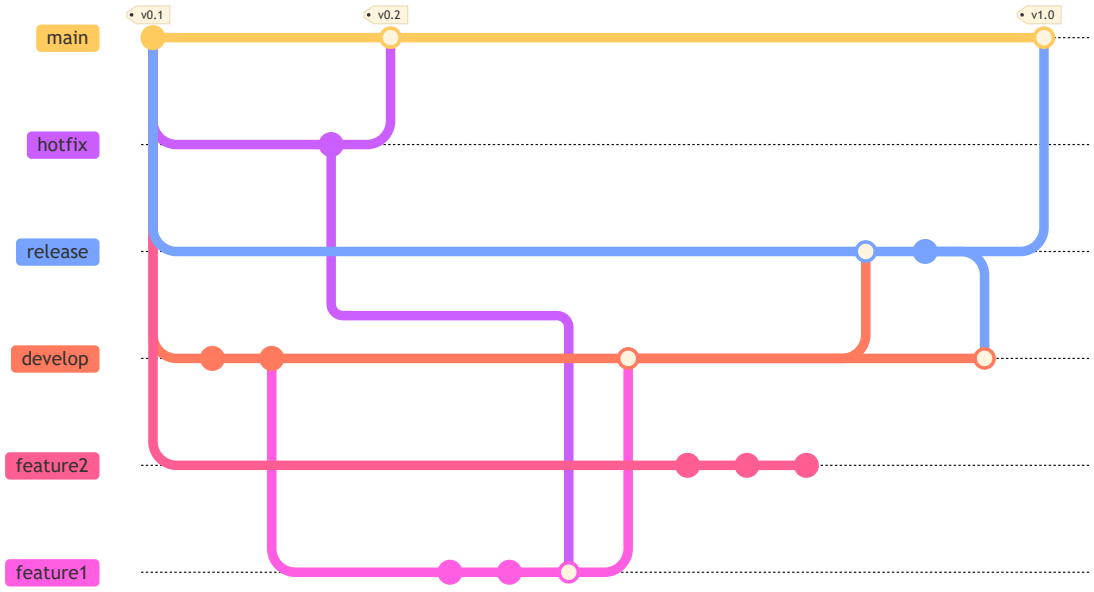

Figure 4.3: Example of Git Flow

- **Main Branch**:
  Main branch serves the same purpose as the main branch in GitHub Flow. It contains production ready code that can be released. Here, in figure 4.3, 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 main 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.

4.4 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 [11]. 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. CI/CD pipelines ensures the code quality is maintained throughout the development process, which is essential for the long-term success of the software.

- **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.

4.5 Frameworks

4.5.1 What is a Framework?

A framework is a pre-built structure that provides a foundation for developing applications [7]. 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.

4.5.2 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.

4.5.3 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.

Chapter 5

Creation of Framework

5.1 Concept of CI 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 5.1 provides an overview of the structure and functionality of the framework.

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

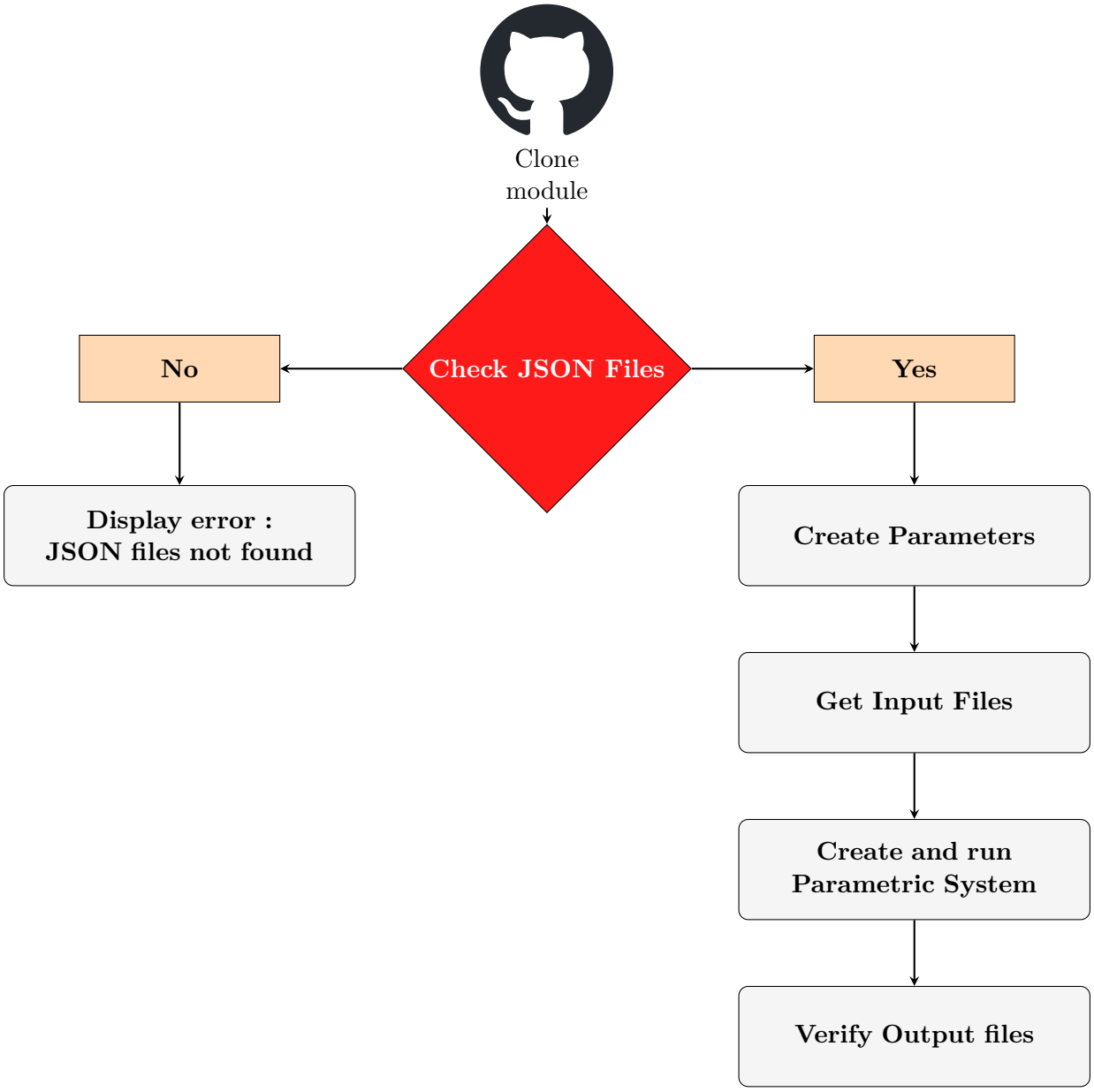

Figure 5.1: Flowchart of the framework

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 the module present. Therefore, the first step is to clone the required module from the specific repository and branch from GitHub. These serve as the main arguments needed to run the framework. After cloning the module, 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. Code snippet 5.1 shows the implementation 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 5.1: 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 5.5.

After the input files are retrieved, the framework proceeds to create the parametric system. This is achieved without the user's intervention, i.e, automatically. At this stage, the Python interpreter provided by Optislang is executed as it includes the necessary libraries and functions to create and run the parametric system. Another terminal emerges, displaying the progress of the module creation.

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`.

A high level overview of the framework is provided in Section A.1 which provides the functions and classes used in it.

5.2 Directory Structure

The directory structure is depicted in Figure 5.2. 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.

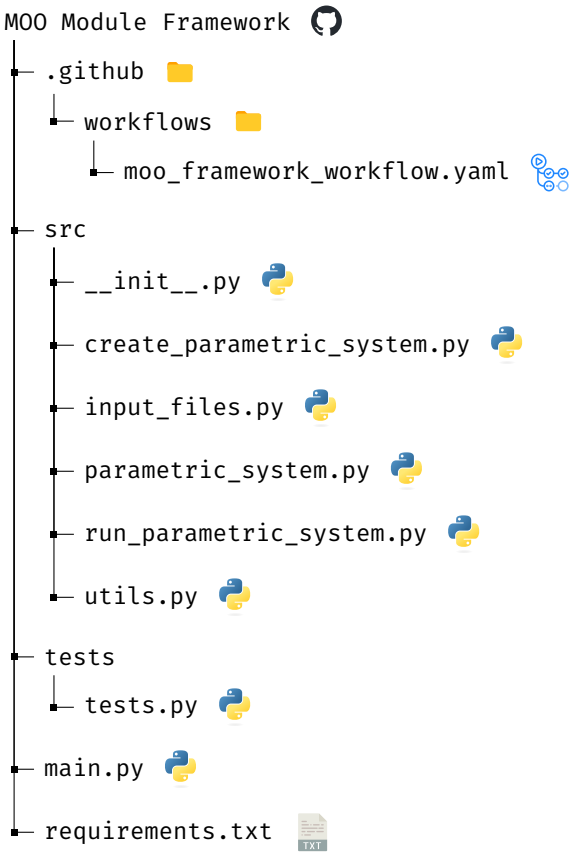

Figure 5.2: Directory structure of the framework

5.3 Testing of CI 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`. In this thesis,`unittest` is used 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. Boundary tests, negative tests, and unit tests are implemented in the framework. Boundary tests are implemented to check the edge cases of the functions. Here, a test function is implemented to verify whether the generated parameters for the system is empty or not. Negative tests are implemented to check if the function handles incorrect input properly. To verify the data types generated in the output file, a test case is implemented. This test verifies if the data types in the columns of the output file are generated correctly.

Code snippet 5.2 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 5.2: Example of a unit test

First, a class `TestParametricSystem` is created, which inherits from `unittest.TestCase`. To avoid initialization of the same variables in each test case, **`setUp()`** method is used. Here, an object is created to 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 5.2,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 or not. To ensure this, `assertEqual` is used, 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.

5.4 Execution of CI Framework

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

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. Execution of the framework is provided in detail with an example in Chapter 7.

5.5 Tools used in the CI Framework

Input files which are required by the module can be mocked using the data in `module_config.json`. But, in some modules, the input files required are very complex. For example, the module, `ROM`

SOLVER, needs input files in the form of matrices, containing 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.

To store the input files, OpenShift is used. OpenShift[6] is a Kubernetes platform which helps in deploying, scaling and managing containerized applications. OpenShift is used in this thesis as it is easy to build, deploy and maintain in the cloud. The application in OpenShift needs to be containerized and deployed.

Docker[3] is a platform 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. Docker is preferred for development and deployment as it is lightweight, portable and scalable. This can run anywhere, regardless of the operating system.

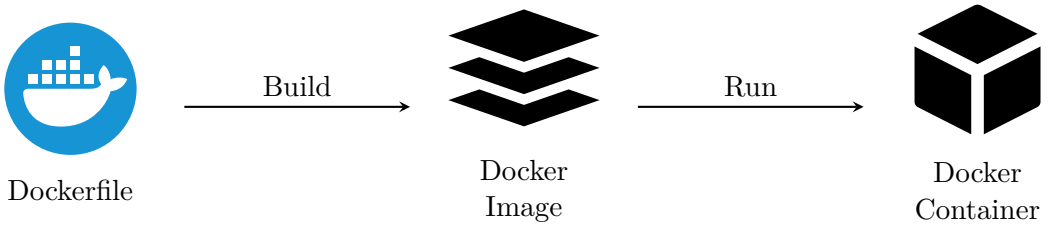

Figure 5.3: Creation and deployment of Docker image

To build an image in Docker, a Dockerfile is required. The Dockerfile contains the instructions to build the Docker image. Figure 5.3 shows the creation of a container. This container contains input files required for modules to run the parametric system in Optislang.

But, the input files are not directly accessible from the container. To transfer the files, a communication medium is required. This is achieved by using API calls. API is a collection of protocols and tools which allows different software applications to communicate with each other. Figure 5.4 shows the working of an API.

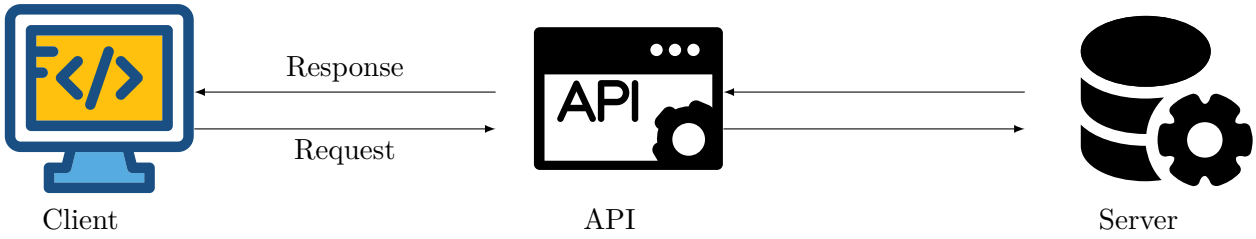

Figure 5.4: Working of an API

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 information back to the client. This thesis leverages the use of API to request for input files from the cloud storage. To build the API, FastAPI is used. FastAPI[4] is a modern, fast (high-performance), web framework for building API with Python. FastAPI is preferred as it is easy to use, fast to develop, high performance and easy to deploy. To secure the API calls and to access the input files, a basic HTTP authentication is implemented in this thesis. Figure 5.5 shows the file structure of the input files storage.

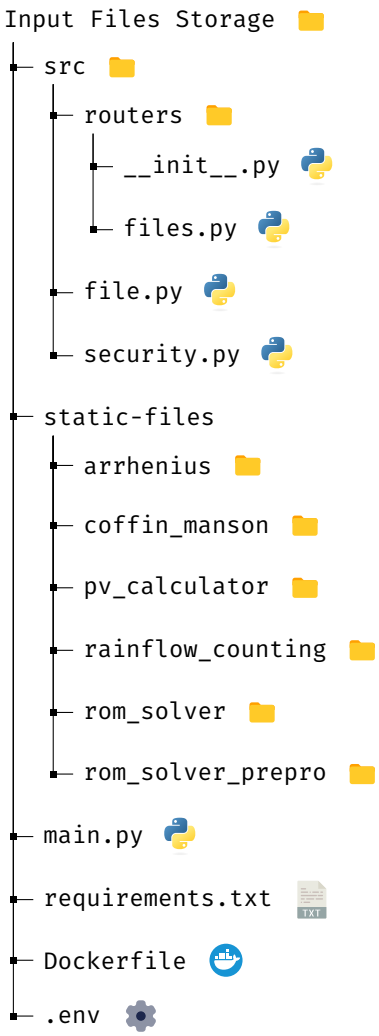

Figure 5.5: Directory structure of the input files storage

`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-value pair 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. Code snippet 5.3 shows the creation of an image using 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
```

```
10
11 RUN mkdir -p ./temp && \
12     chgrp -R 0 ./temp && \
13     chmod -R g=u ./temp
14
15 COPY main.py /code/
16 COPY ./src/ /code/src/
17 COPY .env /code/.env
18 ENTRYPOINT ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "7000"]
```

Code 5.3: 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 in OpenShift. This is later used during the first step of executing the parametric system.

The `.env` file contains the environment variables consisting of authentication details to access the API.

Chapter 6

Implementation of CI Pipeline

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 [21]. 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, a CI pipeline is implemented using GitHub Actions. This pipeline is responsible for running the framework on every commit to the repository. A detailed explanation of the pipeline is provided in the following sections.

6.1 Version Control Strategy

As discussed in Section 4.3, 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.

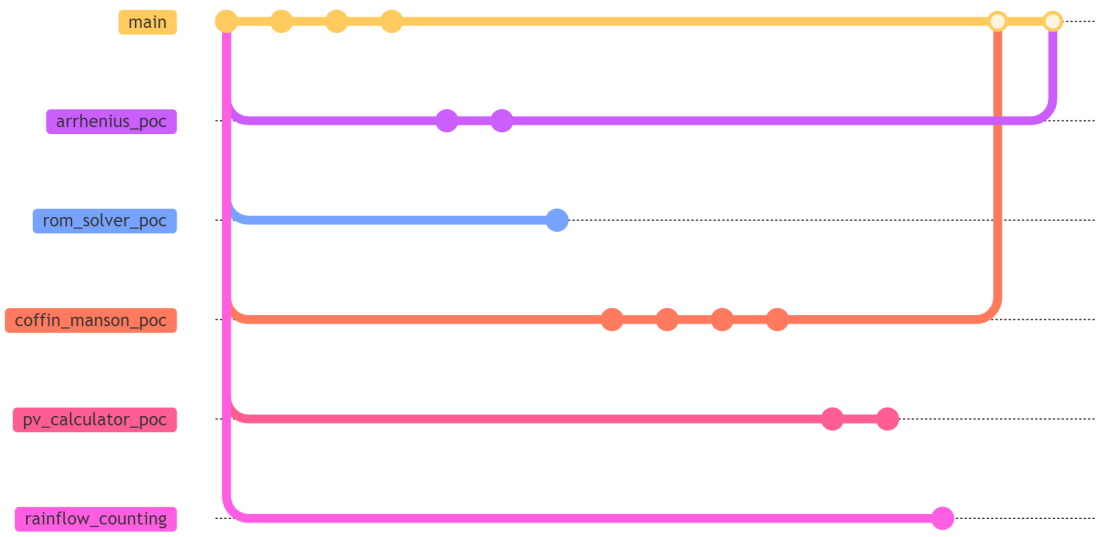

Figure 6.1: Branching strategy for the framework

From the figure 6.1, 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.

6.2 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[17]. 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 6.2 shown below explains the software development lifecycle and the usage of CI/CD in the process.

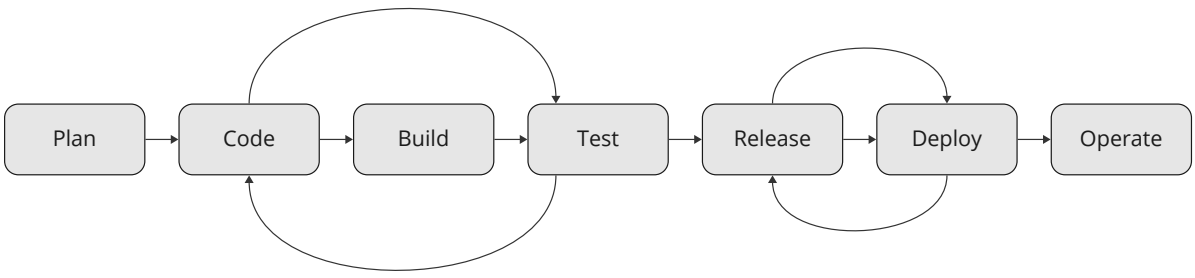

Figure 6.2: Software development lifecycle

GitHub Actions is implemented after step 2 in the figure 6.2. 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.

6.2.1 Components of GitHub Actions

Workflows

This pipeline is implemented in a `.yml` or a `.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 6.1: 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 code snippet 6.1, 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 code snippet 6.1, 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 code 6.1, 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 Code 6.1, 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.

6.3 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 7

Results

This chapter presents the results of the experiments conducted in the previous chapters. The results are present with an example of execution of a standalone module in Optislang.

`ARRHENIUS` module is responsible for calculating the field load equivalent hours according to Arrhenius lifetime model for each relevant monitor point in the design. To run this module, the developer needs to execute `main.py`, providing the module name and name of the branch where the module is present. Code snippet 7.1 provides the code for executing the framework using `main.py`.

```python
from src import ParametricSystem

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

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 7.1: Execution of framework using `main.py`

`main.py` calls the `ParametricSystem` class containing the framework. After cloning the module, the framework verifies the existence of the `JSON` files. If the files are not present, the framework provides a log message to the user.

```
c:\Users\BIS4SI\Desktop\MOO_Module_Framework\MOO_M_ARRHENIUS\01_Specification\module_config.json does not
exist
Please ensure c:\Users\BIS4SI\Desktop\MOO_Module_Framework\MOO_M_ARRHENIUS\01_Specification\module_config.
json exists and re-run
```

Code 7.2: Error message when JSON files are not present

In code snippet 7.2, the framework provides an error message when the JSON file, `module_config.json`, is not present.

After verifying the existence of the JSON files, the framework continues further to create the required parameters and runs the parametric system.

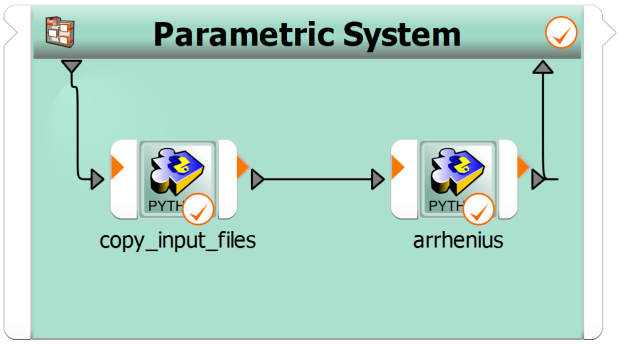

Figure 7.1: Execution of ARRHENIUS module in Optislang

Figure 7.1 shows the parametric system created by the framework. The system consists of a python actor, **copy_input_files**, which contains the algorithm to retrieve the input files from OpenShift via API calls. These input files are stored in the working directory of the parametric system. Later, the system continues to run the actor, `arrhenius`, containing the module. The tick mark on the actors indicate the successful execution in Optislang. Successful execution of the parametric system is also displayed in the console as shown in Code snippet 7.3.

```
Attribute Qt::AA_UseSoftwareOpenGL must be set before QCoreApplication is created.
INFO : Saving project "parametric_system"
INFO : New project "parametric_system". Working dir : "c:\Users\BIS4SI\Desktop\MOO_Module_Framework\MOO_M_
ARRHENIUS\Module\02_Model\parametric_system.opd".
INFO : working directory of "parametric_system" set to "c:\Users\BIS4SI\Desktop\MOO_Module_Framework\MOO_M
_ARRHENIUS\Module\02_Model\parametric_system.opd"
INFO_DETAIL : License feature checkout requested (first of): optislang_level2 [in 281 milliseconds]
INFO_DETAIL : 2024-Sep-30 09:14:07.340325 : License feature checkout requested (first of): optislang_level
2 [in 1 millisecond]
INFO : 2024-Sep-30 09:14:07.340325 : Auto-saving project "parametric_system"
INFO : 2024-Sep-30 09:14:07.409259 : parametric_system : Current iteration successfully prepared
INFO : 2024-Sep-30 09:14:07.409259 : Parametric System : Sent Design 1
INFO : 2024-Sep-30 09:14:07.432607 : Parametric System : Current iteration successfully prepared
INFO : 2024-Sep-30 09:14:11.214374 : copy_input_files : copy_input_files processed successfully [Design 1]
INFO : 2024-Sep-30 09:14:12.848899 : arrhenius : arrhenius processed successfully [Design 1]
INFO : 2024-Sep-30 09:14:12.859403 : Parametric System : Collected Design 1
INFO : 2024-Sep-30 09:14:12.863430 : Parametric System :  Parametric System processed successfully
INFO : 2024-Sep-30 09:14:12.863430 : Auto-saving project "parametric_system"
INFO : 2024-Sep-30 09:14:12.024235 : Auto-saving project "parametric_system"
INFO : 2024-Sep-30 09:14:12.976036 : Auto-saving project "parametric_system"
INFO : 2024-Sep-30 09:14:13.042652 : Total execution time : 6 seconds
INFO : 2024-Sep-30 09:14:13.059409 : Saving project "parametric_system"
```

Code 7.3: Message providing status of parametric system execution

This whole framework is implemented in the CI pipeline using GitHub Actions. The pipeline is triggered when a developer pushes the code to the repository. If all of the steps in the pipeline are successful, new commits are pushed to the branch. If the pipeline fails, the developer is notified via email about the failure. This is shown in Figure 7.2.

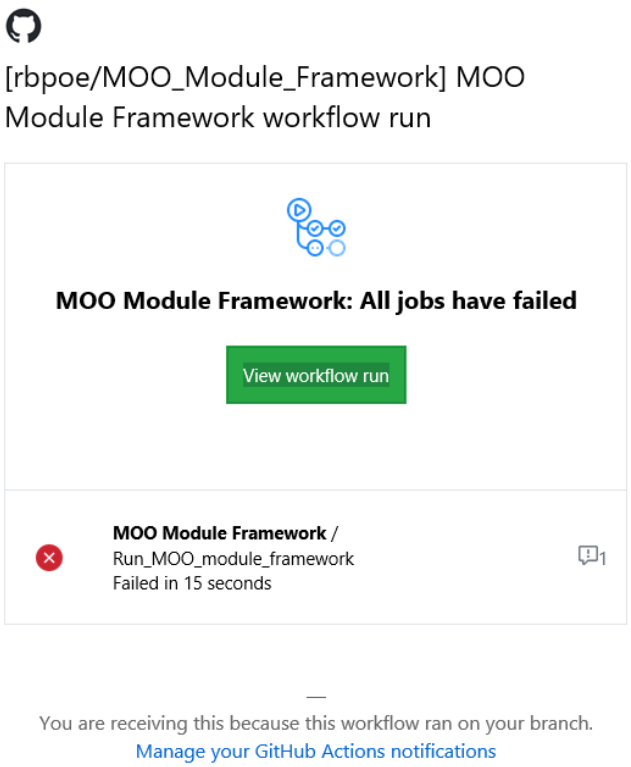

Figure 7.2: Notification of pipeline failure in GitHub Actions

After the successful execution of the system in the pipeline, the framework checks the output file generated by the module. Here, the algorithm checks if the output file generated is as expected, which is provided in `module_config.json`. A detailed explanation verification of output files is provided in Section A.1.3. A detailed log message is provided to the user regarding the output file. Code snippet 7.4 shows the log message provided to the user.

```
arrhenius_results.csv present
{'monitoring_point': 'str', 'time': 'float'}
Column 'monitoring_point' not found in arrhenius_results.csv
Column 'time' not found in arrhenius_results.csv
```

Code 7.4: Log message for output file verification

In Code snippet 7.4, the algorithm confirms the presence of the output file, `arrhenius_results.csv`. But, the column names `montoring_point` and `time` are not present. If the column names are present, it verifies if the data types of the columns are as expected.

The framework is also responsible for creating and excuting MATLAB modules in Optislang. The developer needs to provide the module name and the branch name where the module is present. The framework takes care of the rest of the process. Figure 7.3 shows the successful execution of the MATLAB module, `COFFIN MANSON`, in Optislang.

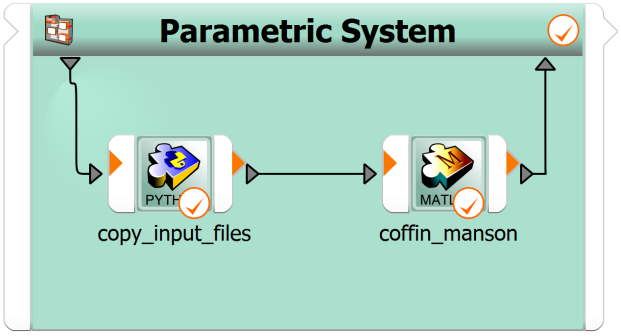

Figure 7.3: Execution of MATLAB module in Optislang

The execution of the MATLAB modules takes a bit longer compared to the Python module. This is due to execution of MATLAB in the background. Currently, the framework is capable of running Python and MATLAB modules. The framework is built in such a way that it can be extended to run other modules as well. To run other modules in the framework, the developer only needs to provide the module name and branch of the module. The framework and the CI pipeline take care of the rest of the process.

Earlier, the developer had to manually test their new commits to the module by running the module in Optislang. This process was taking around 30 minutes to complete. With the help of the framework, the developer can now run the module in Optislang in seconds. In Code snippet 7.3, the framework took around 6 seconds to run the module in Optislang. This is a significant improvement in the time taken to run the module in Optislang compared to the manual process.

Chapter 8

Summary

8.1 Conclusion

The goal of this thesis was to develop a Continuous Integration framework for Optislang workflows that tests the modules and provides immediate feedback to the developer without the user intervention. By implementing this framework, modules can be tested faster and reduce errors during the development process. Through the implementation of a Python-based framework, integrated with GitHub Actions, the workflow has been successfully automated, simplifying the development process for system developers at Bosch

Throughout the project, key efforts were directed toward automating the execution and testing of these modules, which were previously handled manually. By incorporating DevOps best practices into the framework, the process has become much more efficient, offering immediate feedback whenever changes are pushed to the repository. This not only saves time but also helps minimize human error during development.

The work presented here is particularly relevant as industries increasingly depend on automation to stay competitive. With the growing complexity of modern software and optimization tools like OptiSlang, automating repetitive tasks like testing ensures that companies can move faster and deliver more reliable products, which is crucial for companies like Bosch.

Ultimately, the takeaway from this thesis is clear: integrating CI practices into simulation-based environments can greatly enhance productivity, reliability, and scalability. This work demonstrates how automation can play a vital role in ensuring the success of complex workflows, enabling developers to focus more on innovation and less on routine tasks.

8.2 Future Work

The framework developed in this thesis is a starting point for automating Optislang workflows. Future work could focus on expanding the framework to support more modules and tools. For example, the framework could be extended to support for workflows that involve multiple modules, or to integrate with other optimization tools. Additionally, the framework could be enhanced to support more advanced testing. This changes can be implemented easily by extending the functionality of the framework.

Some of the older modules in the framework do not consist the data for verifying output files. This is a crucial step in the framework to verify the output files generated by the module. By standardizing the files in every module, the framework can be used for all the modules in the repository.

Storage of input files in this thesis is done through OpenShift and FastAPI for a proof of concept. This can be improved by using a more secure and scalable solution. One possible solution is to store the files in a Blob storage in Azure[1], securing the files and making it easier to access the files.

[1] https://azure.microsoft.com/en-us/products/storage/blobs/

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.

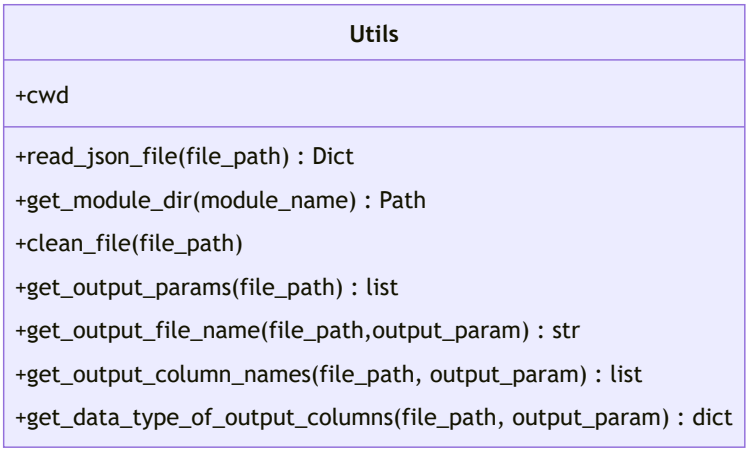

Figure A.1: Overview of functions in `utils.py`

`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.

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. After providing the input files, the parametric system is created.

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.2 shows the functions present in the class `ParametricSystem`.

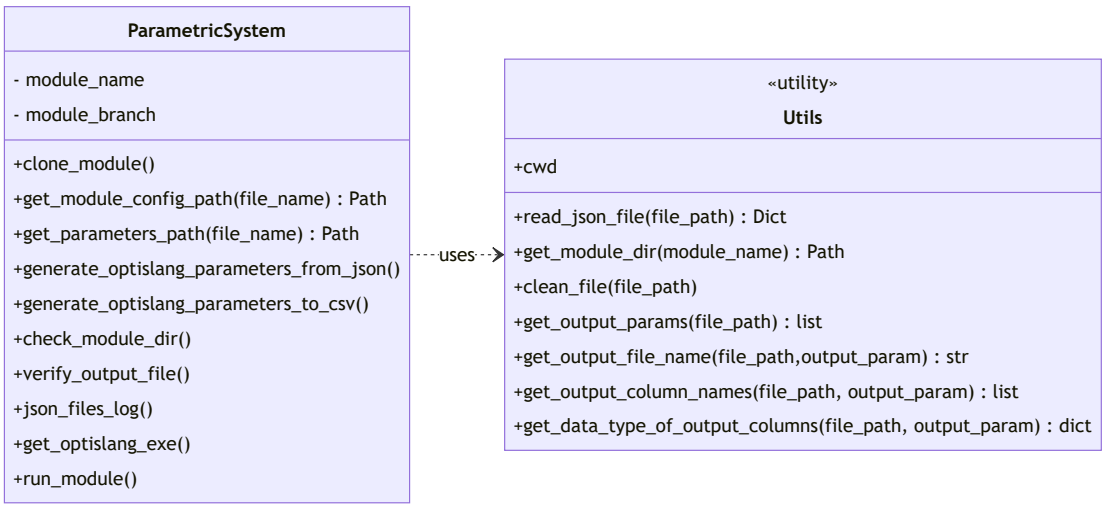

Figure A.2: Overview of the class `ParametricSystem`

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 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.

  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. Figure A.3 shows the working of `verify_output_files()` function.

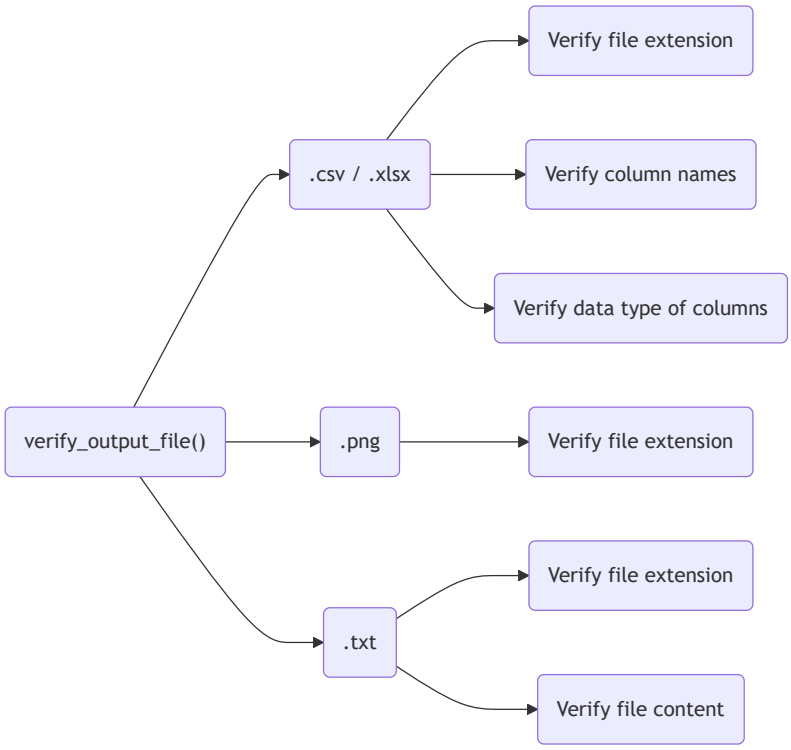

Figure A.3: Working of `verify_output_files()` function

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] Docker. https://www.docker.com/. Online; Accessed on 15/09/2024.

[4] FastAPI. https://fastapi.tiangolo.com/. Online; Accessed on 20/09/2024.

[5] GitHub Actions. https://docs.github.com/en/actions. Online; Accessed on 25/09/2024.

[6] Red Hat OpenShift. https://www.redhat.com/en/technologies/cloud-computing/openshift. Online; Accessed on 15/09/2024.

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

[8] Successfully Applying Continuous Integration for HIL Simulation. *dSPACE*, June 2021.

[9] Accelerate DevOps with Continuous Integration and Simulation. *Windriver*, May 2022.

[10] 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.

[11] 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.

[12] Harold P. Benson. *Multi-objective optimization: pareto optimal solutions, propertiesMulti-objective Optimization: Pareto Optimal Solutions, Properties*, pages 2478–2481. Springer US, Boston, MA, 2009.

[13] Christian Bucher, Johannes Will, and Jörg Riedel. Multidisciplinary non-linear optimization with Optimizing Structural Language OptiSLang.

[14] Martin Fowler and Matthew Foemmel. Continuous Integration, 2006.

[15] Shahnewaz Amin Jolly, Vahid Garousi, and Matt M. Eskandar. Automated unit testing of a scada control software: An industrial case study based on action research. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 400–409, April 2012.

[16] Puneet Khetarpal, Marco Dragic, and Govind Malleichervu. Accelerating Simulink Simulations in Continuous Integration Workflows with Simulink Cache Files. *MathWorks*, 2022.

[17] 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.

[18] Enrique Larios Vargas, Joseph Hejderup, Maria Kechagia, Magiel Bruntink, and Georgios Gousios. Enabling real-time feedback in software engineering. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 21–24, May 2018.

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

[20] 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.

[21] 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.

[22] 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.

[23] L. Felipe Perrone, Christopher S. Main, and Bryan C. Ward. Safe: Simulation automation framework for experiments. In *Proceedings of the 2012 Winter Simulation Conference (WSC)*, pages 1–12, Dec 2012.

[24] Piero Pezze and Nicolas Gambardella. Sbpipe: A collection of pipelines for automating repetitive simulation and analysis tasks. *BMC Systems Biology*, 11, 12 2017.

[25] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, Nov 1996.

[26] Tanja E.J. Vos, Beatriz Marín, Maria Jose Escalona, and Alessandro Marchetto. A methodological framework for evaluating software testing techniques and tools. In *2012 12th International Conference on Quality Software*, pages 230–239, Aug 2012.