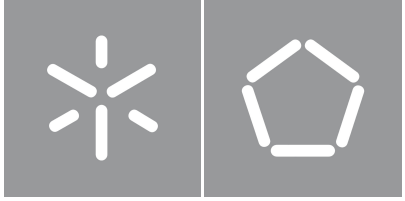


University of Minho
School of Engineering

Luís Manuel Gonçalves da Silva

Methodology for Validation and Performance Analysis of Compilation Toolchains

January 2024



University of Minho
School of Engineering

Luís Manuel Gonçalves da Silva

Methodology for Validation and Performance Analysis of Compilation Toolchains

Master's Dissertation
Master in Informatics Engineering

Work carried out under the supervision of
André Martins Pereira
Nuno Filipe Gomes Cardoso

January 2024

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.
This work can thereafter be used under the terms established in the license below.
Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY-NC

https://creativecommons.org/licenses/by-nc/4.0/

Acknowledgements

I would like to express my sincere gratitude to my university mentor, Doctor André Pereira, whose guidance and support have been instrumental in shaping the direction and quality of my work.

A special mention goes to Doctor Nuno Cardoso, who served as both a mentor and a friend during my time at Synopsys where this project was undertaken. Nuno's insights, encouragement, and collaborative spirit have significantly enriched the project experience.

I would also like to express my gratitude to Doctor Cupertino Miranda for his vital contribution in introducing me to Synopsys, presenting the project's idea, and providing valuable assistance. His guidance and support were crucial to the success of this endeavor.

I extend my appreciation to my colleagues and teammates for their collaboration, insights, and encouragement, contributing to the success of this project.

I am also grateful to Synopsys for their support and provision of resources, without which this endeavor would not have been possible.

To my family and friends, thank you for your understanding, patience, and unwavering support during challenging times.

Lastly, I want to acknowledge the countless individuals who may not be named here but have played crucial roles in various capacities, contributing to the success of this project.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.
I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, January 2024

Luís Manuel Gonçalves da Silva

Abstract

A Toolchain is a fundamental element of software development and provides developers with the resources necessary for the development, testing, and maintenance of applications. Unreliable test environments can cause major problems for the successful completion of tests. resulting in a decrease in performance and quality of the application. The disparity between tests executed locally and remotely by different development teams can exacerbate this issue, leading to inconsistency in results. In order to guarantee reliable and comparable results it is essential that all developers use the same environment and configuration when conducting tests. This consistency is particularly important to obtain an accurate depiction of the application's capabilities, revealing any potential issues that may not be prevalent in one environment but exist in another.

This research aims to create a streamlined and efficient testing methodology for Toolchain developers, developed in collaboration with Synopsys, a leader in the semiconductor industry, provides a range of ARC RISC processors supported by various commercial and open-source operating systems and middleware. This work proposes a tool designed to simplify the testing process for different tools in the Toolchain, allowing developers to easily execute, compare, profile, explore, and report tests with confidence. The tool will enable direct local comparisons, reducing the time and effort required for manual testing and reporting. By using this tool, developers can save valuable time while ensuring their tests are accurate and reliable. An improved testing methodology will be beneficial to all Toolchain developers, as it will allow them to focus less on manual testing processes and more on developing innovative solutions with confidence. Tests are an essential component of any Toolchain development process, so having an efficient testing methodology in place is key to ensuring the highest standards of quality code. With the proposed tool, developers can rest assured that their tests are accurate, reliable and replicable.

Keywords Continuous Integration, Compare Past Executed Iterations, Web API

Resumo

A Toolchain é um elemento fundamental no desenvolvimento de software que fornece aos desenvolvedores os recursos necessários para o desenvolvimento, teste e manutenção de aplicações. Ambientes de teste pouco confiáveis podem causar grandes problemas para a conclusão bem-sucedida dos testes, resultando numa diminuição de desempenho e qualidade da aplicação. A discrepância entre testes executados localmente e remotamente por diferentes equipas de desenvolvimento pode agravar esse problema, levando a inconsistências nos resultados. Para garantir resultados confiáveis e comparáveis, é essencial que todos os desenvolvedores usem o mesmo ambiente e configuração ao realizar testes. Essa consistência é especialmente importante para obter uma representação precisa das capacidades da aplicação, revelando quaisquer problemas potenciais que possam não ser evidentes em um ambiente, mas que existam em outro.

Esta pesquisa tem como objetivo criar uma metodologia de teste simplificada e eficiente para os desenvolvedores da Toolchain, desenvolvida em colaboração com a Synopsys, líder na indústria de semicondutores, que fornece uma variedade de processadores RISC ARC suportados por vários sistemas operativos comerciais e de código aberto, além de middleware. Este trabalho propõe uma ferramenta projetada para simplificar o processo de teste para diferentes ferramentas na Toolchain, permitindo que os desenvolvedores executem, comparem, explorem e reportem testes com confiança. A ferramenta possibilitará comparações locais diretas, reduzindo o tempo e esforço necessários para testes manuais e relatórios. Ao usar essa ferramenta, os desenvolvedores podem economizar tempo valioso, garantindo que seus testes sejam precisos e confiáveis. Uma metodologia de teste aprimorada será benéfica para todos os desenvolvedores da Toolchain, permitindo que se concentrem menos nos processos manuais de teste e mais no desenvolvimento de soluções inovadoras com confiança. Testes são um componente essencial de qualquer processo de desenvolvimento de

Toolchain, portanto, ter uma metodologia de teste eficiente é fundamental para garantir os mais altos padrões de código de qualidade. Com a ferramenta proposta, os desenvolvedores podem ter certeza de que seus testes são precisos, confiáveis e replicáveis.

Palavras-chave Integração Contínua, Comparar Iterações Anteriores Executadas, Web API

Contents

I Introductory material 1

1 Introduction 3

1.1 Challenges and Goals . . . 5

1.2 Document structure . . . 6

2 State of the Art 7

2.1 Continuous Integration . . . 7

2.1.1 Continuous Integration Tools . . . 9

2.2 Version Control Systems . . . 13

2.2.1 Version Control System Types . . . 13

2.2.2 Git Version Control . . . 16

2.2.3 GitHub Webhooks . . . 17

2.3 Toolchain . . . 18

2.3.1 Toolchain Testing Infrastructures . . . 18

II Core of the Dissertation 21

3 BSF 23

3.1 The Proposed Framework Workflow . . . 24

3.2 Requirement Analysis . . . 24

3.2.1 Functional Requirements . . . 25

3.2.2 Non-Functional Requirements . . . 27

3.3 Architecture . . . 29

3.3.1 BSF . . . 30

3.3.2 Lock . . . 30
3.3.3 OptionParser . . . 31
3.3.4 Config . . . 33
3.3.5 Build . . . 34
3.3.6 Status . . . 35
3.3.7 Exceptions . . . 36
3.3.8 Clean . . . 36
3.3.9 Log . . . 36
3.3.10 Cat . . . 36
3.3.11 Ls . . . 37
3.3.12 VarManager . . . 37
3.3.13 DirManager . . . 37
3.3.14 Helper . . . 37
3.3.15 Publish . . . 38
3.3.16 Diff . . . 38
3.3.17 Compare . . . 38
3.3.18 GitManager . . . 39
3.4 BSF Structure . . . 39
3.4.1 File Hierarchy . . . 39
3.4.2 Configuration Folder Structure . . . 41
3.4.3 Configuration File Structure . . . 42
3.5 BSF Features . . . 47
3.5.1 Initialisation Flowchart . . . 48
3.5.2 Finalisation Flowchart . . . 49
3.6 Auxiliary Scripts . . . 58
3.6.1 Get Versions Script . . . 58
3.6.2 Check Versions Script . . . 60
3.6.3 testsuite-filter Script . . . 61
3.6.4 Compare Script . . . 62
3.7 Validating BSF . . . 65
3.7.1 Software Tests . . . 71
3.8 CI with GitHub Webhooks and BSF External API . . . 75

3.8.1 External API Implementation and Integration with Webhooks 75
3.8.2 Requirements Analysis 77
3.9 External API Architecture 83
3.9.1 Overall System Architecture 84
3.9.2 Database 84
3.10 Webhook Workflow 90
3.10.1 Task Execution Middleware 94

4 Conclusions and future work **97**
4.1 Prospect for future work 98

III Appendices **103**

A BSF Use Cases **105**
A.1 Clone Use Case 105
A.2 Status Use Case 105
A.3 Tasks Use Case 106
A.4 Set Use Case 106
A.5 Vars Use Case 106
A.6 Sources Get Use Case 106
A.7 Sources Delete Use Case 107
A.8 Sources List Use Case 107
A.9 Sources Show Use Case 107
A.10 Git Use Case 108
A.11 Search Use Case 108
A.12 Diff Use Case 108
A.13 Ls Use Case 109
A.14 Cat Use Case 109
A.15 Cp Use Case 109
A.16 Compare Use Case 110
A.17 Compareall Use Case 110
A.18 Report Use Case 110
A.19 Save Config Use Case 110

B BSF Features **113**

B.1 Save Config Feature 114

B.2 Clone Feature 115

B.3 Vars Feature 116

B.4 Tasks Feature 117

B.5 Set Feature 118

B.6 Diff Feature 120

B.7 Git Feature 121

B.8 Search Feature 123

B.9 Status Feature 124

B.10 Ls Feature 127

B.11 Cat Feature 128

B.12 Sources Get Feature 130

B.13 Sources Delete Feature 132

B.14 Sources List Feature 133

List of Figures

1 Continuous Integration Process 8
2 Branches Concept 15
3 Proposed framework workflow diagram 24
4 BSF Class Diagram Architecture 29
5 BSF Environment File Hierarchy 40
6 Configuration Folder Hierarchy Example 41
7 Initialisation flowchart 48
8 Finalisation flowchart 49
9 The flowchart outlines the workflow for the init feature. 50
10 The flowchart represents a execute command flow. 52
11 The flowchart represents the compare feature flow. 54
12 The flowchart outlines the workflow for an publish command. 56
13 Output from Compare Script. 63
14 Output from Compare Script with Verbose. 63
15 BSF Output for Compare Feature between HEAD and LOCAL with verbose. . . 71
16 BSF Output for Compare Feature between two releases: arc-2023.03 and arc-2023.09. 71
17 External API Architecture. 83
18 External API Database Structure. 85
19 Webhook Sequence Diagram. 90
20 The flowchart outlines the workflow for the *saveconfig* feature. 114
21 The flowchart outlines the workflow for the *clone* feature. 115
22 The flowchart outlines the workflow for the *vars* feature. 116
23 The flowchart outlines the workflow for the *tasks* feature. 117

24 The flowchart outlines the workflow for the *set* feature. 118
25 The flowchart outlines the workflow for the *diff* feature. 120
26 The flowchart outlines the workflow for the *git* feature. 122
27 The flowchart outlines the workflow for the *search* feature. 123
28 The flowchart outlines the workflow for the *status* feature. 125
29 The flowchart outlines the workflow for the *ls* feature. 127
30 The flowchart outlines the workflow for the *cat* feature. 129
31 The flowchart outlines the workflow for the *sources get* feature. 131
32 The flowchart outlines the workflow for the *sources delete* feature. 132
33 The flowchart outlines the workflow for the *sources list* feature. 133

List of Tables

1 Use Case representing the Initialisation of BSF using a configuration folder. . 25
2 Use Case representing the Execution of a single task or multiple tasks. 26
3 Use Case representing the publish feature of BSF's internal repository. 26
4 Use Case representing the display of the log of a task. 26
5 Internal Property Values Description. 43
6 Defining the significance of every available user-defined property value. . . . 66
7 Use Case API representing the *login* POST request type. 77
8 Use Case API representing the *register* POST request type. 78
9 Use Case API representing the *change_permissions* PUT request type. 78
10 Use Case API representing the *kill* POST request type. 79
11 Use Case API representing the *start_task* POST request type. 79
12 Use Case API representing the *tasks* GET request type. 80
13 Use Case API representing the *machines* GET request type. 80
14 Use Case API representing the *ci_tasks* PUT request type. 81
15 Use Case API representing the *ci_tasks* GET request type. 81
16 Use Case API representing the *ci_update_process_id* PUT request type. 82
17 Example for the Bitmap Permissions for GCC and QEMU modules. 88

18 Use Case representing the initialisation of BSF by repository. 105
19 Use Case representing the display of the status of each task. 105
20 Use Case representing the display of the name and description of each task. 106
21 Use Case representing the set of an User-Defined Property Value. 106
22 Use Case representing the display of all User-Defined Property Values. . . . 106
23 Use Case representing the clone feature of repositories. 107
24 Use Case representing the delete feature of repositories. 107

25 Use Case representing the list feature of repositories. 107
26 Use Case representing the show feature of repositories. 108
27 Use Case representing the Git feature of the framework's internal repository. 108
28 Use Case representing the search feature of the framework. 108
29 Use Case representing the diff feature of the framework. 109
30 Use Case representing the ls feature of the framework. 109
31 Use Case representing the cat feature of the framework's internal repository. 109
32 Use Case representing the cp feature of the framework. 109
33 Use Case representing the compare feature of the framework. 110
34 Use Case representing the compareall feature of the framework. 110
35 Use Case representing the report feature of the framework. 110
36 Use Case representing the saveconfig feature of the framework. 111

Part I

Introductory material

Chapter 1

Introduction

A Toolchain is an integral part of the software development process, providing developers with the necessary resources to create, test, and maintain applications. It is composed of various components such as compilers, linkers, debuggers, libraries, and other tools that are used to convert source code into an executable form. Collaboratively, these tools allow for the successful development and deployment of software applications. Lack of consistency in test environments is one of the most important factors that can lead to issues in the successful execution of tests. These issues can have a significant impact on both the performance and quality of the application. To ensure the correct development of an application, it is essential for any company to establish and maintain a consistent and trustworthy testing environment. This requires frequent tests to be conducted to properly measure the status of the software product. This way, the company can be confident that all tests are being conducted accurately and consistently. Without this, the risk of encountering errors or inconsistencies within the software increases significantly, leading to issues which may not be detected until it is far too late.

Code quality inspection and assessment is a vital part of software development that helps teams improve the overall quality of their applications and ensures that each feature will be implemented with the highest quality, capabilities, and performance. Developing and maintaining compilation tools is an extremely labour-intensive process, requiring a strict and organised procedure to be done correctly. This process involves executing and analysing the results of multiple verification tools and benchmarks to ensure the accuracy of the compilation tools. This is an incredibly time-consuming task and has the potential to result in errors due to its complexity.

Synopsys, a technology leader in the semiconductor industry, offers a range of ARC RISC processors supplied by a wide range of commercial and open-source operating systems and middleware. The ARC family of processors, which includes 32-bit and 64-bit variants, is built on an advanced RISC CPU architecture. The GNU Toolchain tools like Binutils, GCC, GDB, Newlib, uClibC, GlibC, Linux, and the QEMU simulator support these processors. As a result of the extensive ecosystem provided by Synopsys and the GNU Toolchain Team, these processors can be deployed in numerous embedded designs, including data storage devices, multi-media players, automotive infotainment systems and smart home appliances.

Synopsys has implemented Continuous Integration (CI) to ensure that software applications remain stable and functional. Through CI, code changes are tested automatically and integrated into the main code base regularly. This not only reduces the risk of integration issues, but it also ensures that quality standards are maintained throughout the development process. While it is a powerful way to ensure a successful product, the process can be quite time-consuming and requires a dedicated person or team to handle it properly. This type of structure demands an additional level of maintenance and efficiency, which can be challenging.

Given the complexity involved in setting up and maintaining CI, it needs to be done in a systematic manner. It can be broken down into several stages, such as setting up the server, configuring the repository, installing the CI tool, and writing scripts for integration tests. A team that is well versed in all these areas needs to be employed to ensure that everything runs smoothly. An experienced team with expertise in software development and testing is essential to guarantee that the software works effectively and efficiently across all relevant architectures.

This thesis aims to establish a comprehensive methodology that will eliminate the need for a specialized team to manage the testing process for all tools within a Toolchain to be employed at Synopsys. The methodology will ensure that the software is thoroughly tested and any issues are identified and resolved in a timely and effective manner, resulting in a high-quality end product.

This will provide a reliable and effective methodology for Toolchain developers that simplifies the execution, comparison, profiling, exploration, and reporting of tests. The project is being developed in collaboration with Synopsys, considering their needs and preferences. The methodology should be easy to use and abstract the deployment of testing and reporting for different tools in a Toolchain. This way, Toolchain developers will be capable of applying the method with confidence to produce more consistent results. Additionally, the tool should allow developers to directly use it to produce comparisons locally.

This type of methodology could be beneficial to Toolchain developers, as they will have access to a dependable tool that will help them reduce time spent on manual executions and comparisons. Additionally, they won't have to worry about complex deployments and reporting for the various tools in the ARC Toolchain. The potential of such an innovation is exciting and could improve the way Toolchain developers approach tests and their results at Synopsys, making their jobs easier and more efficient.

1.1 Challenges and Goals

The automation in a CI environment needs a widely accepted standard within the community. To ensure the accuracy of test results, maintaining consistent test configurations and environments is of utmost importance. Variations in test execution environments can compromise result validity, emphasizing the need for standardisation to achieve uniformity in test execution. However, this standardisation is yet to be universally implemented, leading to discrepancies between locally executed tests (on individual machines or development environments) and remotely executed tests (using the deployment system). To address this, clear guidelines and procedures for test execution must be established and adhered to by all developers, both in their local development environments and when deploying tests remotely.

The complexity of testing tools necessitates the involvement of expert support teams for managing and maintaining them, incurring significant expenses. A uniform execution methodology is vital to control test results effectively. Precise control over test environments and configurations ensures the legitimacy of result comparisons. In the new methodology, rather

than relying on dedicated teams, the toolchain developers take on the responsibility of contributing new configurations.

A user-friendly and easily configurable tool is essential, enabling developers to utilise it both locally and in a CI environment while employing the same configuration. A key goal is to empower all developers to contribute new recipes to the system, each operating within its designated execution environment to avoid conflicts with other tasks. To ensure widespread applicability across developers and accommodate diverse scenarios, the tool's configuration file will comprehensively describe its functionality and specify the files to be stored in the internal repository. Adopting a version control system will efficiently manage and preserve all relevant data for seamless collaboration and retention.

1.2 Document structure

The document is organized as follows: Chapter 2 presents a examination of the current state-of-the-art in the fields of verification and testing. It covers topics such as the concept of Continuous Integration and its various implementations, along with techniques for testing compilation tools. Chapter 3 details the methodology intended to address limitations in current testing systems. Furthermore, it outlines the development of the framework, covering its architecture, structure, validation, and results. It also discusses the development of an external API architecture. Chapter 4 is dedicated to conclusions and future work.

Appendices A and B, included in this document, complement the framework Use Cases and Features not addressed in the core document, respectively.

Chapter 2

State of the Art

In this chapter, will be discussed the fundamentals of Continuous Integration and its importance in modern software development. Delving into the world of version control systems that are available and widely used across the globe. Additionally, the discussion will analyse the concept of a Toolchain, its components, and establish the standard testing framework for all Toolchain components.

2.1 Continuous Integration

Continuous Integration (CI) is a fundamental software development practice that streamlines the process of tests, builds, and releases through automation. The primary goal of CI is to ensure the reliability of software releases and facilitate prompt bug-fixing by seamlessly integrating new code changes into the project's source code repository. Facilitating teams to build and release software, CI significantly reduces time to market and elevates overall quality. Belmont (2018)

In traditional software development approaches, a substantial amount of manual effort is invested in building and releasing software. After each modification to the source code, developers must dedicate hours to manually compile new versions of the code-base and run tests on those versions. This manual process can extend to several days or weeks, leading to missed deadlines and the potential shipment of buggy or incomplete products. The lag time between making code changes and delivering those changes to customers can have adverse consequences.

CI resolves these challenges by automating these manual tasks, ensuring swift execution

without impeding developers' workflow or causing missed deadlines. When implemented effectively, CI empowers teams to produce superior, more reliable software at accelerated speeds, eliminating the need for unnecessary tasks like constant code compilation that can hinder productivity.

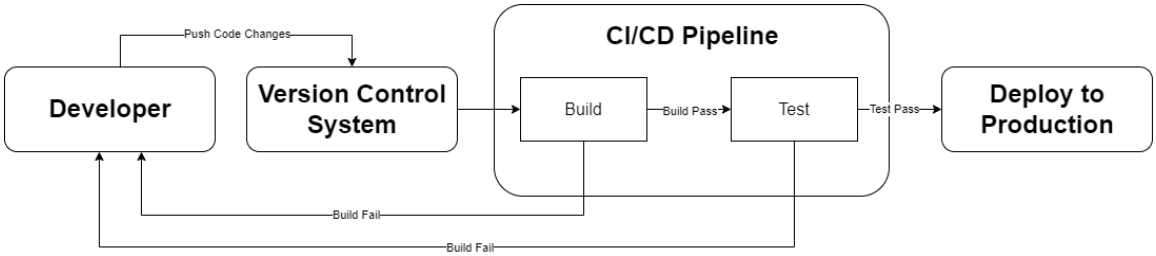


Figure 1: Continuous Integration Process

Figure 1 illustrates the CI workflow, beginning with the developer writing and implementing software code. After making modifications to the source code, the developer proceeds to upload their changes to a communal code repository, which serves as a version control system for storing, sharing, and tracking code changes over time. Subsequently, the CI server detects these changes and proceeds to build the most recent version of the software. It also executes automated tests to ensure the code's functionality and adherence to quality standards. In the event of any test failures, developers are promptly notified, alerting them to address and rectify the issues, thereby initiating the process anew. Conversely, if all tests pass successfully, the CI system integrates the code changes into the main code base, rendering it production-ready.

According to Hilton et al. (2016), companies that embrace CI undoubtedly reap numerous benefits that significantly enhance their software development processes. Among these advantages are a reduced risk of integration problems, improved collaboration among developers, and the swift delivery of new features and bug fixes to users. These benefits represent just a fraction of CI's transformative potential for companies. Let's delve into more of these advantages in the following paragraphs.

Keeping software up-to-date is a critical aspect of the development process. Not only does it ensure access to the latest version, but it also fosters more efficient coding practices. With

modern development tools, incremental changes can be made to the code without requiring a complete build every time. This empowers developers to focus on making minor, continuous improvements and adding new features without enduring long build cycles. Additionally, continuously updating software helps prevent conflicts arising from working with outdated versions, ensuring smooth and efficient operation across all environments. Ultimately, maintaining up-to-date software streamlines the development process and facilitates the creation of high-quality products with minimal hassle.

Maintaining code integrity is paramount in software development, and automated tests are a powerful tool to achieve this goal. Manual testing becomes redundant by running automated tests against code, reducing the testing budget. Automated testing allows developers to check their code quickly and efficiently, verifying that any changes made do not compromise the existing code's integrity. Automated testing saves valuable time and resources that can be better invested elsewhere in the development process.

Having a clear understanding of issues and quickly identifying what needs fixing during development is crucial minimising wasted time and keeping teams focused on their objectives. This becomes particularly vital for projects involving multiple teams working on different components, as coordination can become challenging when one team awaits the completion of another part. CI is an invaluable tool to help teams maintain focus and stay on track. By adopting CI, teams can readily detect and address issues during development, preventing unnecessary delays and keeping the project momentum intact. As a result, teams can concentrate on their objectives rather than being preoccupied with error correction or complications that may arise.

2.1.1 Continuous Integration Tools

Continuous Integration tools have revolutionised the software engineering landscape by automating building, testing, and deployment processes. They provide rapid feedback to developers, enabling them to assess the impact of code changes on the overall application. Examples of popular CI tools include Jenkins, Travis CI, Circle CI, and BuildBot. Each offering unique advantages and integration capabilities.

- Jenkins (2011): As a Java-based open-source platform, Jenkins is one of the most widely used CI systems. Its popularity can be attributed to the vast array of plugins it offers, allowing integration with various tools and technologies. This extensibility makes Jenkins a versatile and customisable choice for developers.

- TravisCI (2011): A hosted CI service, Travis CI provides developers with a user-friendly and reliable solution for building, testing, and deploying software projects. It supports a wide range of programming languages and offers robust GitHub integration. Additional features, such as parallel builds, build matrixes, code coverage, code quality assessment, and deployment tools, enhance its appeal to developers seeking a comprehensive CI solution.

- CircleCi (2011): Regarded as one of the most advanced cloud-based CI and delivery platforms, Circle CI impresses users with its powerful and efficient capabilities in builds, tests, and deployments. Leveraging parallelism features, it offers a seamless experience for building projects in multiple programming languages. Its integration with popular services like GitHub, BitBucket, and GitLab further streamlines the development workflow and promotes team collaboration.

- BuildBot (2003): Designed as a robust open-source CI and delivery platform, BuildBot caters to organisations seeking highly customisable and flexible CI solutions. With its web-based interface, Buildbot enables users to monitor builds and seamlessly integrate with version control systems, build tools, and test frameworks. This adaptability makes it suitable for diverse project requirements.

However, relying solely on CI tools for testing has limitations. When developers perform local testing, they may have specific configurations, dependencies, or environmental factors that differ from the standardised environment maintained by the CI system. These differences can result in varying testing results. For instance, a developer might use a different version of a library, a specific system setup, or even different hardware specifications. These disparities can lead to cases where the code appears to work correctly on the developer's machine but fails during CI testing.

Developers can ensure consistency and reliability in their testing efforts by having a tool that supports both individual testing and CI with the same configuration. When developers run tests locally with the same configuration as the CI environment, they can catch potential issues early, reducing changes of integration problems later. This alignment between local and CI testing environments minimises discrepancies and enhances the overall quality of software.

Some advantages and disadvantages of CI tools are summarised below, according to Ionos (2020), Longbottom (2021), Small (2018), GitLab :

Advantages

- Early Issue Detection: CI enables early detection and resolution of issues during development, leading to time and resource savings in the long run.
- Increased Efficiency and Productivity: Automating the integration process and CI tools help teams work more efficiently and improve their development workflows' speed.
- Enhanced Code Quality: Frequent check-ins and automated testing ensure a stable and defect-free code base, improving overall code quality.
- Agile Responsiveness: CI facilitates quick response to changes in the operational environment, allowing teams to adapt rapidly to new requirements or challenges.
- Standardised Processes: CI promotes the creation of automated and standardized procedures for building, packing, and testing applications, ensuring consistency in the integration process.

Disadvantages

- Initial Adoption Challenges: For well-established teams, transitioning to CI can be challenging and may initially lead to increased time investments instead of immediate time savings.
- Monitoring and Reporting Overhead: CI necessitates continuous monitoring and clear communication of changes, especially those related to back-end databases, business processes, and user experience alterations.

- User Resistance to Constant Change: Users might not appreciate frequent changes to the User Experience, such as renamed functions, repositioned menu-bar items, or alterations to established steps.
- Complex Installation and Configuration: Implementing CI tools can be a complex and time-consuming process, often requiring a dedicated team or individual to maintain and configure the environment properly.
- Resource Intensive: Certain CI tools may demand more resources than others, potentially impacting the system's overall performance. This increased resource requirement could lead to higher infrastructure costs or slower build times.

2.2 Version Control Systems

Version Control Systems (VCS) form the foundation of the CI process, serving as essential tools for managing and tracking changes to various files, including source code. These systems allow developers to save multiple versions of files, enabling easy reversion to previous states in case of errors. Such capability is critical for project success and helps prevent costly mistakes. The integration of code into VCS marks the initiation of the CI process.

Beyond managing file versions, VCS offer an auditing trail that records changes and their respective authors and timestamps. This feature fosters collaboration and keeps all team members informed about ongoing developments. Additionally, the auditing trail proves invaluable in locating the resolving bugs and errors that may surface at later stages of the development process.

2.2.1 Version Control System Types

Various types of VCS exist to cater to different project scales and team sizes. Some are tailored for small teams, while others are robust enough to handle larger-scale projects with hundreds or thousands of users. The versatility of these systems allows development teams to choose the one that best aligns with their specific needs, ensuring efficient collaboration and streamlined version tracking.

In agreement with Serengetitech (2020), two commonly used types of VCS are Centralised Version Control Systems (CVCS) and Distributed Version Control Systems (DVCS).

Centralised Version Control Systems

In a CVCS, a central server stores the entire repository, including all versions of files and their history. Developers check out files from the central server to work on them and then check them back when done. This model allows for collaboration between team members, but it requires a constant connection to the central server for most operations.

Some examples of Centralized Version Control Systems:

- Subversion (2000): Subversion, often abbreviated as SVN, is one of the most well-known CVCS. It has been widely used in the past and is still used in some organisations. SVN operates on a client-server architecture, storing all the versioned files and history on a central server. Developers can check out files from the server, work on them, and then commit changes back to the central repository.

- Perforce (1995): Perforce, previously known as Helix Core, is another popular CVCS used in various industries, particularly in game development and large-scale enterprise projects. Similar to Subversion, Perforce uses a client-server model to manage versioned files.

- Team-Foundation-Version-Control (2005): Team Foundation Version Control is a version control system developed by Microsoft and integrated into their Azure DevOps platform. It is commonly used with Microsoft Team Foundation Server and allows teams to collaborate on software projects using a centralised repository.

- IBM-Rational-ClearCase (1992): ClearCase is an older but still used CVCS, particularly in enterprise environments. It enables developers to version control files and directories in a central repository and supports features like branching, merging, and access control.

- IBM-Rational-Synergy (1990): Rational Synergy, also from IBM, is a software configuration management tool that operates as a CVCS. It is designed to manage complex software projects, enabling teams to work together on large-scale projects with centralised control over versioning.

While CVCS were more prevalent in the past, many organisations have transitioned to Distributed Version Control Systems (DVCS) like Git due to their advantages, such as offline work, better collaboration, and fast branching and merging.

Distributed Version Control Systems

DVCS are version control systems that allow each developer to have their own complete copy of the entire repository, including the full history of the project, on their local machine.

This decentralisation and duplication of the repository enables developers to work independently and efficiently without requiring a constant connection to a central server. DVCS provides numerous benefits, such as better support for collaboration in distributed teams, the ability to work offline, faster branching and merging, and improved data redundancy.

In a DVCS, developers can commit changes to their local repository, create branches, merge changes, and review history without relying on a central server for most operations. Collaboration often happens peer-to-peer, where developers can exchange changes directly between their local repositories, making it possible to work efficiently in various workflows.

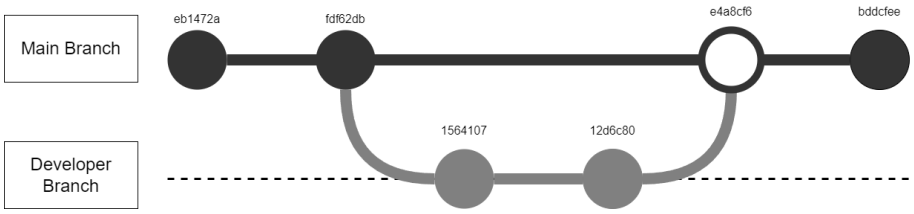


Figure 2: Branches Concept

One essential aspect of DVCS is the concept of branches, illustrated in Figure 2 which are separate lines of development within the repository. Developers can create branches to work on new features, bug fixes, or experiments without impacting the main project until they are ready to merge their changes into the main branch.

Examples of DVCS:

- Mercurial (2005): Mercurial provides similar functionalities to other VCS. It was created by Matt Mackall in 2005 and offers a more straightforward and user-friendly interface. Mercurial is often chosen by developers who prefer a simpler and more intuitive version control system.
- Bazaar (2005): Bazaar is a DVCS developed by Canonical, the company behind Ubuntu. It aims to provide an easy-to-use, flexible, and decentralised version control system suitable for small and large projects. Bazaar allows developers to work efficiently in distributed teams and offers a range of features for collaborative development.
- Git (2005): Git is one of the most widely used and popular DVCS systems. It was cre-

ated by Linus Torvalds in 2005 and has since become the standard for version control in the software development industry. Git holds immense value for software developers and professionals seeking efficient project management. it is a powerful tool to track changes in digital assets, enabling developers to access the code history, quickly locate errors, and promptly rectify them. Moreover, Git promotes seamless collaboration, allowing multiple individuals to work simultaneously on different aspects of a project and effortlessly merge their changes.

2.2.2 Git Version Control

One of Git's major advantages lies in its ability to showcase the evolution of code over time. This feature facilitates issue identification and correction without starting from scratch. In case of errors, developers can quickly revert to a previous version, ensuring the code remains up-to-date and functional. Git's extensive benefits cater to developers and non-developers alike, enabling efficient collaboration, change tracking, debugging, and much more. It has become an essential tool for professionals aiming to maintain their digital assets in top condition, significantly improving efficiency, accuracy, and overall effectiveness.

Some noteworthy benefits of using Git include:

- **Speed**: Git's exceptional speed is attributed to its use of SHA-1 hashes to identify revisions. By comparing hash values instead of internet files, Git achieves unmatched performance, empowering developers to manage their code base swiftly and efficiently. In today's complex software development landscape, Git's speed is a key advantage, enabling rapid progress and increased productivity.

- **Flexibility**: Git offers great flexibility by not requiring the entire project to be stored in its repository. This allows developers to use Git for specific parts of an application or individual features, making it an adaptable tool for various project scenarios. Developers can track changes, collaborate effectively, and easily switch between versions of their code, aiding debugging, and issue resolution.

- **Consistency and Reliability**: Git ensures each change made to a file has a unique identity, avoiding conflicts between multiple contributors. Each commit is independently tracked with a unique hash, guaranteeing code integrity, and preventing efforts from

being overwritten. This consistency and reliability enhance collaboration confidence and maintain a stable code base over time.

While Git primarily operates as a command-line tool, GUI clients are available, such as GitHub and GitLab, to ease its learning curve. Despite its powerful features, Git's complexity can be overwhelming for beginners, necessitating a learning curve to utilise it effectively.

Git's impact on modern software development cannot be overstated. Its ability to streamline collaboration, maintain code history, and promote agile development practices has revolutionised how teams manage projects. Speed, flexibility, and consistent tracking make Git an invaluable asset for any development team seeking a robust VCS. Its widespread adoption and integration into development workflows showcase its effectiveness and importance in today's software engineering landscape.

2.2.3 GitHub Webhooks

As mention in GitHub, GitHub Webhooks offer a powerful way to stay informed about the latest activities and events happening within your GitHub repositories. By utilising Webhooks, one can receive real-time notifications whenever specific actions occur, such as code commits, pull requests, issue updates, repository creations, and more.

When setting up a webhook for a repository, GitHub will send a payload of data to a specified external web server or application whenever the chosen events take place. This payload contains relevant information about the event, allowing you to integrate GitHub with other tools and automate various processes based on the received data.

Webhooks can be configured at the organisation, repository, or GitHub App level, giving the flexibility to apply them to different parts of the GitHub ecosystem. Additionally, one can choose which events to subscribe to, tailoring the Webhook notifications to your specific needs. Setting up a Webhook is a two-step process. First, define the events one wants the Webhook to listen for. Then, configure your server or application to receive and process the incoming payload data effectively. By doing so, one can take action immediately when important events occur in repositories, streamlining development workflows and enhancing collaboration within development teams.

2.3 Toolchain

As indicated by Saak and Köhler (2021), in software engineering, a Toolchain is a collection of tools used to develop complex programs. Typically consisting of multiple software development tools, the Toolchain is organised in a predetermined sequence and interconnected to form a comprehensive software development system. These tools encompass a build process automation tool, compilers for various programming languages, programs for creating and modifying binaries and libraries, a debugger for identifying and resolving faulty code, and a build system that enables the use of external dependencies. When utilized cohesively, these components can significantly reduce development time while ensuring high-quality results.

The importance of the Toolchain lies in its ability to facilitate a smooth and efficient development process. For example, the build automation tool expedites development by automating tasks such as compiling source code, linking object files, and generating executable files. Compilers allow developers to write programs in different languages, which can be seamlessly integrated using the build system. The debugger aids in identifying and quickly fixing errors, preventing unexpected issues during runtime. Additionally, the build system simplifies the accessibility of header and library files when needed.

In embedded development, the Toolchain, or cross-compiler, is crucial role in creating software that can run on multiple platforms. Cross-compiling involves using a single code base to build executable programs for various platforms, saving time and resources.

2.3.1 Toolchain Testing Infrastructures

Toolchain Testing Infrastructures have become an integral part of the test environment in any software development company. They are designed to provide an efficient and effective way of testing the software created by a company. Toolchain test environments can target specific tools included in the Toolchain, such as compilers, linkers, and even other tools. These testing infrastructures offer a significant advantage over traditional testing methods, as they can provide more accurate results and insights into the performance of the software.

Test Suite

As indicated by the findings in Pinto et al. (2012), a test suite is a collection of test cases that are used to test the functionality of a software program or system. In the context of compilers, for example, a test suite is used to test the correctness and performance of a compiler by running a set of test cases that cover various aspects of the language specification.

DejaGNU - GNU Test Framework

As stated in Free Software Foundation (1993), DejaGNU is a software framework for testing other programs. It provides a single front-end for all tests and offers several advantages, such as the ability to write tests in any language, to run tests on multiple platforms, and to easily add new tests.

According to information found in Free Software Foundation (-), DejaGNU is particularly well-suited for embedded system design, as it allows for testing to be done remotely on development boards, and separate initialisation files can be created for each operating system and board. It is thus popular with many GNU projects, universities, and private companies.

According to Free Software Foundation (2006), to use the GNU Test Framework, developers can write test scripts in any language, or use the existing ones, such as the test suite for GCC, and the *runtest* script to run the tests. The *runtest* script goes through a directory looking at configuration files and then runs some tests with given criteria.

DejaGNU utilises different codes to signify the status of individual test cases. These are some of the most frequently encountered codes and their meanings:

- **PASS**: The test case executed successfully without any errors or warnings.
- **FAIL**: The test case encountered an error or warning during execution and did not produce the expected result.
- **XPASS**: Contrary to expectations, the test case was expected to fail, but surprisingly

passed successfully. (This can happen in a case of validation of, for example, TLS (Thread local storage) where for a Baremetal application, it is expected to fail due to the missing operating system)

- **XFAIL**: As expected, the test case failed, aligning with the anticipated result.
- **UNRESOLVED**: The test case did not produce a definite result due to an error or other unresolved issue during execution.
- **UNSUPPORTED**: The test case is incompatible or unsupported on the current platform or configuration.

A source code compilation can also be considered as a test case, resulting in a simple test case having two outputs: the compilation output and the execution output. In scenarios where the source code needs to be compiled before execution, both stages are vital to ensure the overall correctness and functionality of the program. The compilation output reflects the success or failure of translating the source code into a machine-readable format. In contrast, the execution output indicates the behaviour and correctness of the program during runtime.

Part II

Core of the Dissertation

Chapter 3

BSF

BSF, as a framework, is designed to facilitate developers in using it locally and in a CI setting with consistent configurations and preventing conflicts with other execution files in a specified environment. To ensure broad usability and to accommodate various scenarios, the tool's configuration file provides a description of its functionality and specifies the files to be stored in the internal repository. The incorporation of a version control system efficiently manages and preserves all pertinent data. The central focus of the framework lies in simplicity and ease of installation. It employs a portable configuration stored in JSON format, enabling effortless sharing and generic task automation across diverse projects. With its user-friendly and intuitive command-line interface, developers can quickly and easily interact with the application.

BSF strives to address the limitations inherent in the tools discussed in earlier. It stands out for its versatility, supporting both local and remote executions using the same configuration. One of its standout features is the ability to store past execution iterations in local or remote repositories using the Git Version Control System, making it highly versatile. Developers can effortlessly compare executed tests for development purposes by referencing the commit ID of each iteration.

3.1 The Proposed Framework Workflow

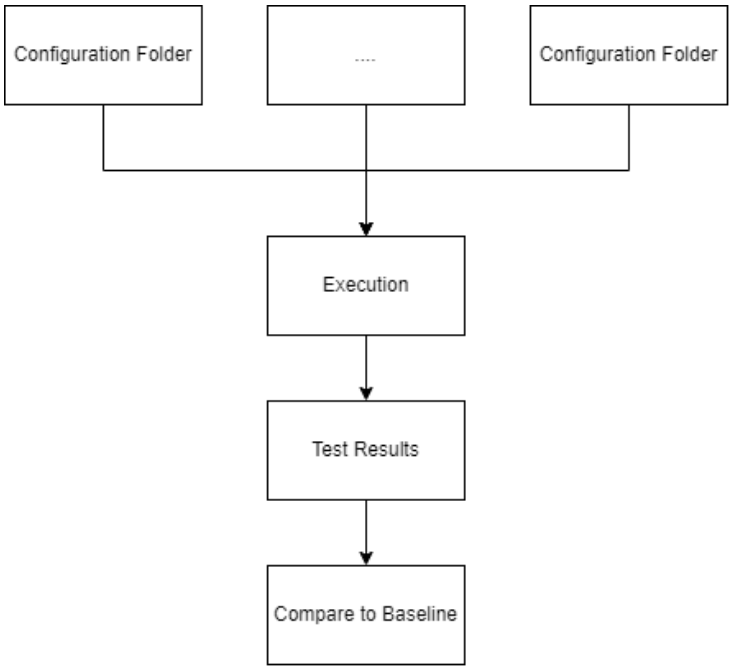


Figure 3: Proposed framework workflow diagram

As presented in Figure 3, the workflow starts with a configuration folder. The configuration file, as a mandatory JSON format file, is used to instruct the system what tasks are to be executed and how to conduct them. For a comprehensive method, it is important that all the necessary files, including the test results and relevant data, are properly stored within the framework using of the configuration file. This allows for easy access and tracking of all the information needed for a thorough and complete evaluation of current or past executed iterations. By properly organising and storing all the data in one place, it becomes easier to analyse and understand the results, ensuring a more accurate and effective testing process. Afterwards, it is possible to either compare the current results to a baseline, which is an existing set of results used as a reference point for further development or testing efforts or store the results to a Git Repository for later comparison.

3.2 Requirement Analysis

In any software development project, defining clear and comprehensive requirements are crucial for a successful implementation. Requirements can be broadly categorised into two

types: functional requirements and non-functional requirements.

3.2.1 Functional Requirements

Functional requirements specify what the software system should do and outline the services, features, and functionalities it must provide to its users. These requirements are typically described using use cases, which illustrate the interactions between the system and its users (Actors). Each use case represents a specific feature that the system offers to fulfill user needs.

In the context of the framework in development, this section will present its functional requirements using use cases. These use cases will outline the various services available to the system's Actors through BSF features.

Init Use Case

The *init* feature initialises a new BSF environment in each directory. When executed, this feature sets up all necessary directory structures and configuration files that are essential for the framework to begin execution.

Use Case	Use Case 01
Name	Initialisation
Objective	The Actor initialises the framework using the configuration folder.
Initialisation	Developer
Actors and Stakeholders	Developer
Frequency of Use	Frequently
Pre-conditions	The developer needs to have a configuration folder that includes a mandatory configuration file and all the necessary dependencies related to it.
Post-conditions in case of Success	The framework environment is initialised successfully.
Post-conditions in case of Failure	The framework environment is not initialised, and the Actor is alerted with an error message.
Main Scenario	The Actor initialises the framework with a configuration folder.

Table 1: Use Case representing the Initialisation of BSF using a configuration folder.

Execute Use Case

The *execute* feature executes all the provided tasks in the configuration file.

Use Case	Use Case 02
Name	Execute.
Objective	The Actor executes a single task or multiple tasks.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The Actor must have defined all user-defined property values and passed the pre-conditions of the task.
Post-conditions in case of Success	The task execution is successfully.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor executes a task or multiple tasks.

Table 2: Use Case representing the Execution of a single task or multiple tasks.

Publish Use Case

The *publish* feature creates a new entry within the internal repository of BSF's environment, containing details of the currently executed tasks.

Use Case	Use Case 03
Name	Publish.
Objective	Creates a new commit into the framework's internal repository.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The framework's internal repository must be initialised.
Post-conditions in case of Success	A new commit published into the framework's internal repository.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor publishes all the pretended data into the framework's internal repository.

Table 3: Use Case representing the publish feature of BSF's internal repository.

Log Use Case

The *log* feature grants developers the capability to review the individual task logs that are generated with each execution.

Use Case	Use Case 04
Name	Log.
Objective	The Actor displays the log of a certain task.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The Actor must have executed the task independently of the status.
Post-conditions in case of Success	The log is displayed to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor displays the log of a certain task that was generated upon execution.

Table 4: Use Case representing the display of the log of a task.

3.2.2 Non-Functional Requirements

While functional requirements address "what" the system should do, non-functional requirements focus on "how" the system should perform. These requirements are related to the system's performance, reliability, security, usability, and other quality attributes.

Programming Language

The choice of programming language for an open-source project carries profound significance as it directly shapes the project's accessibility, potential collaboration, and overall success within the open-source community. Opting for the most widely adopted scripting programming language prevalent in the open-source ecosystem aligns with the collaborative nature of these projects, offering numerous advantages.

In the case of this project, the decision to employ *Ruby* as the programming language was a clear directive from Synopsys. Nonetheless, I concur that *Python* would have been a more advantageous choice due to its broader utilisation within the open-source community. This could have potentially led to increased community engagement and support. It is important to note, however, that the selection of *Ruby* was based on the unique judgement and strategic considerations of the company.

Version Control as Database

Git and the open-source community have a deep and symbiotic relationship. Git's design, coupled with its open-source nature, has significantly influenced the open-source ecosystem and led to its widespread adoption. Although the company's directive to use Git as a repository for storing all past execution iterations was unequivocal, I wholeheartedly support this strategic decision. Git, in this context, not only serves as a reliable version control system but also aligns seamlessly with our working/testing workflow and objectives.

Within this context, each Git commit serves as a distinct test iteration, encompassing one or multiple tasks' execution. Git commits, beyond housing test results, encapsulate the precise testing environment used. This environment configuration can be effortlessly shared among

developers, ensuring uniform testing conditions across the team.

All test results find their home in Git's history log, serving as a centralised repository for all test-related data. This repository includes the environment configuration and logs generated by the framework at each iteration, along with all necessary files for each test iteration. The inherent accessibility of this approach means that comparisons with previous test iterations are not limited to the original machine where the tests were initially executed. Instead, they can be conducted on any other machine by cloning the BSF environment.

3.3 Architecture

The forthcoming section features the class diagram illustrating BSF's architecture and component relationships.

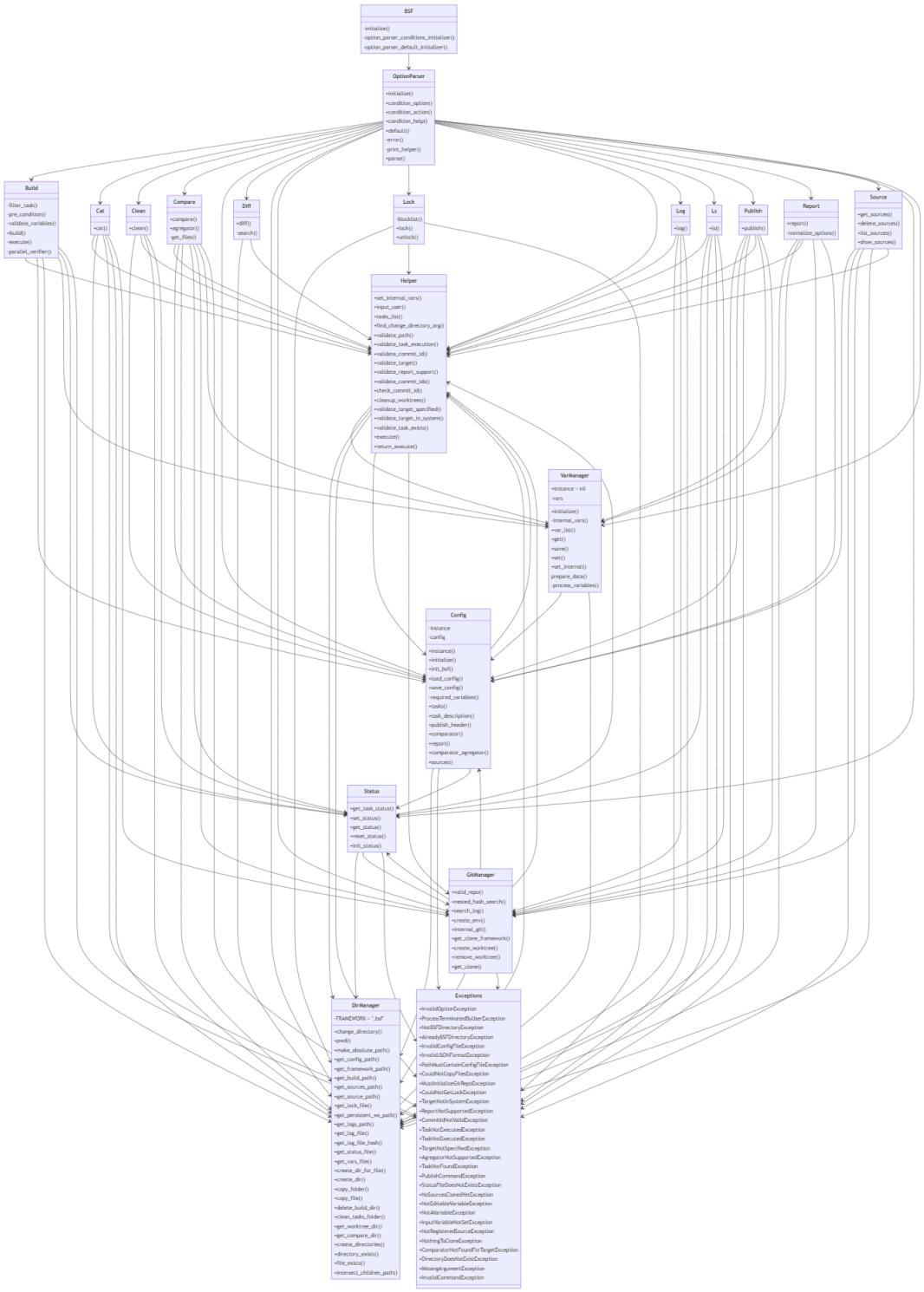


Figure 4: BSF Class Diagram Architecture

In Figure 4, this class diagram functions as a graphical depiction of the system's architecture, offering insights into the relationships and interactions between various components of the software system. It includes a set of components and their associations, illustrating how they collaborate to perform different tasks. Each component has specific methods and attributes that contribute to the functionality of the system.

BSF is designed around the utilisation of two fundamental *Ruby* components: Classes and Modules.

- Classes are not just responsible for encapsulating behaviours, but they also encapsulate state - data that defines the characteristics of the objects they represent. Attributes within classes store their state, and instances of these classes hold that state during program execution.

- Modules cannot create object instances or store state. Instead, they serve as hubs for methods, constants, and variables. This aligns well with components focused on executing functions or processes without data retention. Many components in this architectural design are modules, as they mainly coordinate actions and manage functionalities without persistent data storage.

Following that, a detailed presentation of each of the essential components will be provided.

3.3.1 BSF

The framework is anchored by the *BSF* component, serving as the entry point and responsible for managing the initial instance state. It not only defines features for developers but also integrates comprehensive documentation, providing a user-friendly guide to the framework's functionality and usability.

3.3.2 Lock

Embedded within the architecture, the *Lock* module emerges as a protector of operational integrity. It introduces a carefully coordinated mechanism to maintain order and exclusivity within the BSF ecosystem.

At its core, it employs a preventive approach to ensure the smooth and exclusive execution of critical features. It does so by identifying and intercepting sensitive features, preventing their unintended execution. This step is a proactive barrier, safeguarding the system from accidental disruptions.

This component uses a controlled locking system. This system enforces exclusivity, allowing only one instance of the framework to engage with specific operations at any given time by creating a lock that temporarily restricts access.

A helpful feature for these locks is the recording of the most recent initialisation time of the hosting machine. This approach is crucial in mitigating potential deadlocks that might arise during execution, especially when the machine undergoes a reboot. By recording the machine's initialisation time during the locking process, it proactively guards against undesirable results. During subsequent checks, if the recorded time matches the current machine's initialisation time, the framework is still operational. Conversely, in cases where the recorded initialisation time diverges from the current machine's time, a different scenario comes to light. This divergence signifies that the machine experienced a shutdown at some point between the lock and unlock stages. In such cases, the lock is released, prompting the initiation of a fresh BSF instance.

3.3.3 OptionParser

Embedded within the architecture is a component known as *OptionParser* designed as a module, which encapsulates the core logic for processing command-line options and arguments. It processes the command-line arguments provided by the developer, interprets their intent, and translates them into actionable tasks. This process involves carefully assessing predefined conditions, each associated with specific command patterns, taking full advantage of *closures*. A feature of the *OptionParser* component is its capacity to dynamically initiate predefined actions according to identified command patterns and their corresponding options. The module seamlessly guides the framework towards executing intended tasks by triggering suitable actions.

Drawing inspiration from the practices of numerous open-source applications, the option parser ensures a familiar and consistent experience for developers accustomed to working with similar software. By adhering to established conventions, the option parser facilitates a seamless application integration into the open-source ecosystem, fostering a user-friendly environment.

Understanding the Available Options

To begin, it is crucial to familiarise oneself with the available options and their intended purposes. Developers can gain insights into these options by referring to the documentation or executing the framework without providing any options. This provides developers with comprehensive information about the available options, their functionalities, and usage instructions.

Basic Usage

Developers interact with the framework via the command line by providing the necessary options and arguments. The general command format is as follows:

```
1    $ bsf [option] [arguments]
```

In this format, the options encompass the various features that execute specific actions within BSF. For example, the *execute* feature is an option that performs a task, while the *clean* feature is another option that handles the removal of task-related elements. On the other hand, arguments are the inputs required to execute these options. For instance, developers may need to provide one or multiple task names as arguments when executing a specific option. Additionally, options may also have associated short or long flags that further refine their behaviour and usage.

Flag Specification Options

Developers have the flexibility to specify flags in diverse formats, which are determined by the conventions established by the framework. Two formats used are:

- Short flags: denoted by single characters preceded by a hyphen (-). For instance, *-y* signifies a short flag.
- Long flags: consists of complete words preceded by two hyphens (–). For example, *–confirm* represents a long flag.

It is important to note that the examples above demonstrate the same action but are presented in distinct formats. This flexibility lets developers choose flags that align with their preferences and usage patterns.

Handling Positional Arguments

In addition to options, an application may also involve positional arguments. This design consideration plays a significant role in allowing developers to structure their commands as desired. Developers are not constrained by a rigid order or placement of options and arguments.

Developers can tailor the command to suit their specific needs by freely positioning options that enable or disable specific functionalities. The option parser within the framework takes responsibility for distinguishing between options and arguments, ensuring proper management of the command structure.

Help and Usage Information

Developers can conveniently access help information in two ways: by using the –help (or -h) option or by executing BSF without any arguments provided. Both methods provide developers with a comprehensive summary of the available options, their descriptions, and potential usage examples. The help information serves as a valuable resource to assist developers in understanding and utilising the framework effectively.

3.3.4 Config

Config assumes the responsibility of managing BSF's environment configuration data, and for this reason, this component is structured as a class. It follows the *Singleton* design pat-

tern that is a software design pattern that restricts the instantiation of a class to a singular instance. The Singleton pattern ensures that a single instance of a class exists throughout the framework's life-cycle. By adopting this approach, the potential for duplication and overlapping of data is effectively eliminated. During BSF's initialisation phase, this class is responsible for loading and retaining the configuration data in-memory. This approach ensures that the framework has swift and efficient access to the configuration information throughout its run-time.

When an instance of *Config* is created, it reads the JSON format configuration file used to initialise the BSF environment. This initialisation involves checking for the presence of a valid *config.json* file in the provided path, ensuring that BSF isn't already configured in the current directory, and copying the necessary configuration files to an internal configuration path. This component validates the configuration file, *config.json*, to ensure a successful initialisation of BSF. The class also offers a method to save the current configuration to a specified directory, excluding the *vars.json* file.

This component provides various methods to access specific configuration data, such as task descriptions, publish headers, comparators, reports, and source-related details. These methods allow other parts of BSF to retrieve relevant configuration settings as needed. It also includes a function that scans the configuration file for property values and returns a list of names. These property values are considered required for proper framework operation, ensuring that they are defined and accessible.

3.3.5 Build

The *Build* component was created to function as a module as it manages the execution of tasks within the system. It initiates the process by specifying which tasks are intended to be executed, either by providing specific task names or executing a group of them. It also takes into consideration any pre-conditions associated with these selected tasks. It prepares the necessary data and conditions from the framework's environment configuration file and executes these pre-conditions individually. Each pre-condition signifies all the essential dependencies that require validation before the execution of the corresponding task can com-

mence. For instance, a task may rely on a specific tool's existence within the machine. If any pre-condition is determined to be invalid, the execution will be terminated, and details regarding the problematic pre-condition will be communicated.

Following the pre-conditions checks, the component oversees the actual execution of tasks. It handles the setup of the workspace directories and log files, ensuring that each task's execution is performed within its scope. If a task comprises multiple commands, they are executed sequentially. Simultaneously, all command outputs are internally redirected to a log file within the framework's environment. This allows for easily identifying errors if any occur during the process.

Developers have the flexibility to choose between parallel or sequential execution of tasks based on their needs. In parallel execution, tasks run concurrently in separate processes in their own scope. Upon each task's execution conclusion, an internal status file receives an update. A successful execution results in the status being set as *Passed*, while any unsuccessful execution will mark it as *Failed* within the status file.

3.3.6 Status

The *Status* was designed as a module and is crucial to the framework's task management system. It is responsible for handling and reporting the status of the various tasks.

One of its primary functions is to provide a consolidated view of the task statuses. It can retrieve and summarise the status of each task, making it easier for developers to monitor the progress and results of these tasks.

This component utilises a mapping mechanism to translate numerical status codes into meaningful labels. This mapping is designed to make the status information more user-friendly and information. A status code of 0 is mapped to *Passed*, indicating a successful task execution. A status code of 1 corresponds to *Failed*, signifying that the task encountered issues or errors during execution. A status code of 9 symbolises *Not Executed*, suggesting that the task has not been executed yet.

The module also supports the resetting of task statuses. This can be useful for starting fresh or reevaluating task execution, allowing tasks to be marked as *Not Executed* returning to their default status.

3.3.7 Exceptions

The module *Exceptions* encapsulates a collection of custom exception classes. These exceptions are designed to handle specific error scenarios within BSF, providing informative error messages when particular issues or conditions arise.

3.3.8 Clean

The *Clean* is a module that is an essential part of BSF and is responsible for managing the cleaning process. Its primary function is to clean up tasks and associated data within BSF's environment. Ensuring a pristine workspace by cleaning up all workspace files and statuses is imperative before initiating a new execution. This proactive step is vital to prevent any interference from existing files that could potentially impact the accuracy and integrity of the execution results.

3.3.9 Log

The log file is one of the most essential resources for a developer, as it contains all the information regarding the commands executed and the error messages. The *Log* module is necessary to monitor the log files in real-time, so that any potential problems can be identified and addressed before they become a major issue.

3.3.10 Cat

The *Cat* module is structured to enable access to the contents of saved files within a persistent workspace. Developing a solution became necessary for this purpose. By supplying the task name and specifying the desired file, an internal cat command is executed, allowing developers to retrieve the relevant information.

3.3.11 Ls

The *Ls* module retrieves all the files stored within the persistent workspace. An internal ls command was developed to fulfill the requirement of obtaining information about all the internally stored files.

3.3.12 VarManager

The *VarManager* class in BSF oversees the management of internal and user-defined property values, loading data from the *vars.json* file on instantiation. It maintains internal variables, checks for variable presence and definition during certain commands, enforces rules, and allows modification of user-defined property values with persistence to the *vars.json* file. Additionally, the class supports processing variables within strings through parsing and follows the Singleton pattern to ensure a single instance throughout BSF.

3.3.13 DirManager

The *DirManager* module in BSF is designed to handle directory and file operations, ensuring proper functioning of the framework. It manages essential paths, including configuration files, workspace, source files, and more. The module facilitates directory creation, preventing duplication, and provides methods for constructing specific file paths. Additionally, it enables copying files and folders, supports directory cleanup for tasks and log files, and manages specialised directories like Git Worktree and compare directories.

3.3.14 Helper

Helper is a testament to the architectural foresight in optimising code organisation and reusability. It promotes a sense of coherence and consistency by offering a shared resource that can be leveraged without the need for replication. It also assists with validations, checks, and interactions offers a layer of abstraction that simplifies the integration of these functionalities across different modules.

3.3.15 Publish

A commit message in JSON format is required for each task to serve as the header for the stored iteration. The *Publish* module uses the *publish_header* property in the configuration file to obtain a mandatory JSON format. The framework automatically combines the obtained JSON format into a single file, consolidating all necessary information for storage.

3.3.16 Diff

The *Diff* module is crucial for comparing and identifying differences between two JSON-formatted messages associated with Git commit IDs.
When storing tool versions for each iteration in a Git commit ID message as a JSON formatted file, it becomes essential to establish a method for discerning differences between the two JSON formats. This contributes to identify which tool might be the source of discrepancies in the results.

3.3.17 Compare

The *Compare* Module is a vital feature enabling developers to compare previous iterations of tasks with versatility for various result comparisons. The strategy involves saving only differences between files stored in a Git repository to conserve disk space. To address unexecuted tasks, the status file of each task is utilised to determine success in each iteration. Tasks are considered for comparison if their status indicates successful execution. An alternative executes individual compare operations for each task's compare property in the configuration file, aggregating information into a single output.
The feature for comparisons is straightforward, using the compare feature with specified arguments for the task and iteration commit IDs. A concise format, *<commit_id_01>:<commit_id_02>*, allows developers to include additional information after the commit IDs, passed directly to comparison scripts. Internally, the framework converts the provided commit IDs syntax into paths, *-h <path_01>:<commit_id_01> -h <path_02>:<commit_id_02>*, creating Git Worktrees for each commit ID. Worktree enables the isolation of specific version of a code base, permitting them to coexist independently in separate directories. The modified syntax ensures comparison scripts can access stored files in the persistent workspace along with their corresponding commit ID or tag if provided.

3.3.18 GitManager

The *GitManager* module serves as a central component for handling interactions with Git repositories in BSF. It acts as a bridge for various Git-related operations, encapsulating Git-specific functionality. The module also facilitates searching for specific commit messages in the Git repository's history, allowing criteria-based filtering.

3.4 BSF Structure

To facilitate a comprehensive understanding of the developed framework, this section outlines the file hierarchy and structure of BSF. Furthermore, it provides insights into the configuration folder structure, the layout of the configuration file, and the properties encapsulated within it.

3.4.1 File Hierarchy

Just as in Git, where the directory intended for code management is created with the *.git* file, in BSF, a similar approach is followed. A hidden directory named *.bsf* is created within the directory.

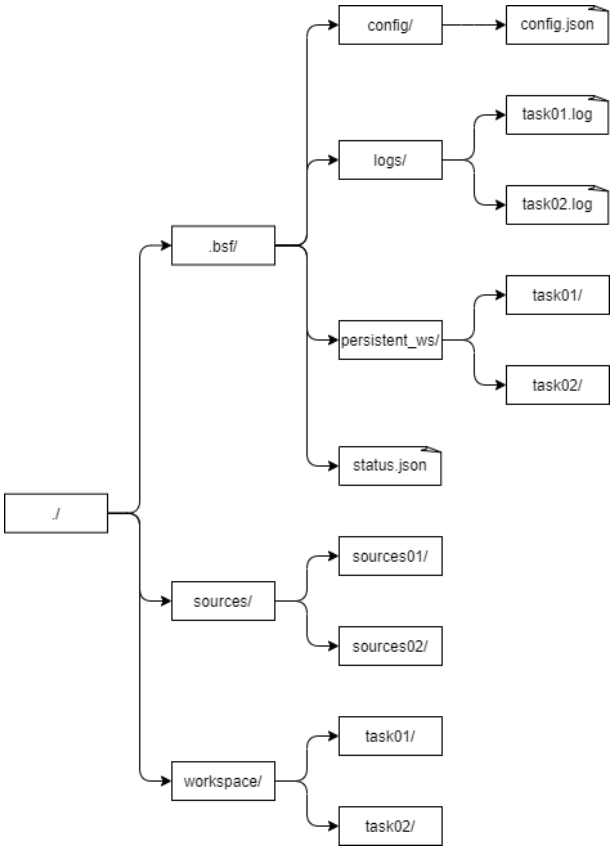


Figure 5: BSF Environment File Hierarchy

The Figure 5 illustrates an exemplary BSF environment structure, where, in this specific case, the configuration file specifies the management of two tasks (task01 and task02) and two sources (sources01 and sources02) within the BSF framework.

- **.bsf/**: The presence of this folder signifies a existence of a BSF environment.
- **.bsf/config/**: This folder contains the internal configuration that was copied to the BSF environment during it's initialisation.
- **.bsf/logs/**: This folder contains the log files generated during task execution organised by task name.
- **.bsf/persistent_ws/**: The persistent workspace is designed for storing files in the internal repository respectively to each task.
- **.bsf/status.json**: This file contains the statuses of each task in JSON-format.
- **sources/**: This folder contains the respective sources cloned by BSF.

- **workspace/**: This folder contains the respective temporary directories and files generated during each task execution.

3.4.2 Configuration Folder Structure

BSF initialisation process relies heavily on a folder that must invariably contain a JSON format configuration file, which essentially acts as the system's core. This JSON file encompasses all the critical information required for the system to operate, including the invocation of scripts and any other essential files.

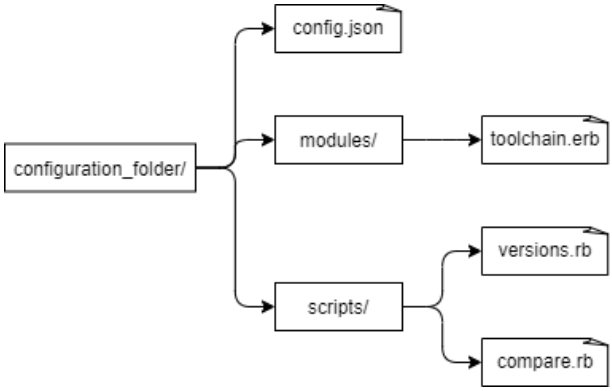


Figure 6: Configuration Folder Hierarchy Example

All these files, shown in Figure 6 must be housed within the initialisation folder to enable the framework to access them through the configuration file.

While configuration files are essential for system operation, they can sometimes grow unwieldy as they accumulate recipes. To address this challenge, a new integration strategy has been devised. This approach involves breaking down the monolithic configuration file into smaller, more manageable units known as *configuration fragments*.

These fragments empower developers by granting them the flexibility to partition the configuration file into distinct pieces. It's important to note that each fragment must have the same type of structure as the configuration file. Each fragment can encapsulate the directives for executing a single task or a group of tasks. This modular approach enhances the flexibility of the system, allowing developers to fine-tune and adapt settings accordingly to their precise needs.

3.4.3 Configuration File Structure

The structure of the configuration file is an integral part of BSF and must be carefully designed. Without a well-defined structure, the application will not be able to execute properly.

The user-defined property values can be defined post-framework initialisation to provide better modularity of the framework. This allows for greater flexibility and more control over how the framework is run. It also enhances scalability and allows customisation to be made to better suit a specific application. It must meet certain requirements to ensure that its elements are well-defined and structured in a logical way.

The configuration file will contain specific types of variables that enable the adjustment of an execution without changing the configuration file itself. The two types of variables are - User-Defined Property Values, represented by *$var(VAR)*; and Internal Property Values, represented by *$var(@VAR)*.

User-Defined Property Values

The User-Defined Property Values, *$var(VAR)*, are intended to help modularity in task executions, requiring developers to define these variables before executing the core task. It is only possible to modify these variables through a BSF feature. BSF parses the configuration file and replacing the defined variables with their corresponding values. If any of these variables are missing, the execution process is halted, and an error message is promptly displayed to the developer.

Internal Property Values

Internal property values, *$var(@VAR)*, are integrated to facilitate developer access to specific directories within BSF. These are hard-coded and internally defined by BSF, with some dynamically adjusting values based on the currently executed task.

Property Values	Value	Description
$var(@ROOT)	*$(pwd)*	Root path of BSF environment.
$var(@BUILDNAME)	*task01*	Name of the current executed task.
$var(@SOURCE)	*sources/*	Sources path.
$var(@WORKSPACE)	*workspace/task1/*	Workspace path of the current task execution.
$var(@CONFIG)	*.bsf/config/*	Path to the internal configuration folder.
$var(@PERSISTENT_WS)	*.bsf/persistent_ws/task1/*	Internal persistent workspace path of the current task execution.

Table 5: Internal Property Values Description.

For example, when executing *task01* within the BSF environment, the definitions outlined in Table 5 are applicable.

Properties

The configuration file must comply with certain properties, pertaining to keys or labels used within the configuration file, to facilitate its use: these are divided into two main sections, *sources* and *tasks*. Sources must include all the necessary information such as repository name, URL, and any additional information that might be required. Each of the tasks within each property must also have a precise guideline which must be followed to complete the task.

sources Property

The *sources* section defines the repositories BSF can clone and utilise. While not obligatory, this process allows developers to consolidate the sources for automation purposes. Each source is represented as an object with a unique name and includes the following properties:

- **repo**: This property specifies the URL of the repository to be cloned;
- **branch (optional)**: If provided, this property specifies the branch of the repository to be cloned. In the absence of this property, the default branch will be used.

An illustration of the *sources* section in the *config.json* file is provided below:

```
{
    "sources": {
        "source1": {
            "repo": "https://github.com/user/repo1",
            "branch": "main"
```

```
        },
        "source2": {
            "repo": "https://github.com/user/repo2"
        }
    }
}
```

Listing 3.1: Sources Property Example

In the given instance, Listing 3.1 defines two sources: *source1* and *source2*. Each source is represented as an object with a unique name. The repository URL can be specified within each source object using the *repo* property. Optionally, the repository branch can be specified using the *branch* property. If the *branch* property is not provided, the default branch of the repository is used.

tasks Property

A task in BSF represents a specific recipe to be executed. Each task consists of parameters that need to be fulfilled for the task to be successfully completed. Except for the *description* property, which accepts a single string, each property accepts a *bash* command as input, either a single string or an array of strings. This design allows for flexibility in executing various types of scripts using bash command invocations, as the core execution, such as tests, is not dependent on *Ruby*. These scripts should be in the configuration folder upon the initialisation of BSF.

Here's an example of the *tasks* property in the *config.json* file:

```
{
    "tasks": {
        "task01": {
            "description": "Task 1 description",
            "pre_condition": "precondition command",
            "execute": "execution command",
            "publish_header": "commit message header",
            "comparator": "comparison command",
            "report": "report command"
        },
        "task02": {
            "description": "Task 2 description",
            "pre_condition": "precondition command",
            "execute": "execution command",
            "publish_header": "commit message header",
```

```
16              "comparator": "comparison command"
17          }
18      }
19  }
```

Listing 3.2: Tasks Property Example

Listing 3.2 showcases examples that demonstrate the structure of the *config.json* file, including the required properties for *sources* and *tasks*.

description Property

The *description* property provides a detailed description of the task's purpose. This helps to comprehend the task's objective and scope better, providing essential context on what the task entails.

pre-condition Property

The *pre_condition* property serves an important role in ensuring that all the necessary dependencies are valid before the execution phase begins. If the dependencies are confirmed, then execution will proceed. However, when certain dependencies fail during the pre-condition validation stage, failure will be printed alongside the corresponding dependency failure. This helps avoid errors during the execution phase, such as an invalid or missing binary while calling them. The *pre_condition* property is thus essential in confirming valid dependencies before the proper execution of each task.

execute Property

The *execute* property within the configuration file is responsible for handling the true objective of the tasks. This property controls all procedures within a unique workspace folder, such as building a Toolchain or executing tests. This ensures that all tasks are performed under a single scope, in a consistent and reliable manner. Furthermore, this helps maintain the order throughout the whole process and ensures that no steps are missed. The Execute property is thus a crucial part of ensuring efficient and successful completion of tasks.

publish_header Property

The *publish_header* property is responsible for setting the commit message that will be stored in the Git commit. It accepts input in JSON format and has the capability to handle multiple

JSON formats, which the framework seamlessly concatenates into a unified format.

comparator Property

The *comparator* property handles the invocation of an auxiliary script responsible for the comparison operation between previous execution iterations, which follows a specific syntax based on the input provided in the comparator feature execution.

report Property

The *report* property handles the invocation of an auxiliary script responsible for the report operation of an executed iteration, which follows a specific syntax based on the input provided in the report feature execution.

3.5 BSF Features

BSF involves features, each representing a sequence of internal actions aimed at achieving specific outcomes. Flowcharts are utilised to illustrate the logical pathways, decisions, and dependencies that guide the framework through its tasks. The subsections accompanying the flowcharts provide contextualization and clarify the intended use and purpose of each feature.

To simplify the following flowcharts, the initialisation and the finalisation instance of BSF are depicted separately and explained individually.

3.5.1 Initialisation Flowchart

The Initialisation phase is a common element across all feature flowcharts, consistently representing the very first step in the execution of the framework.

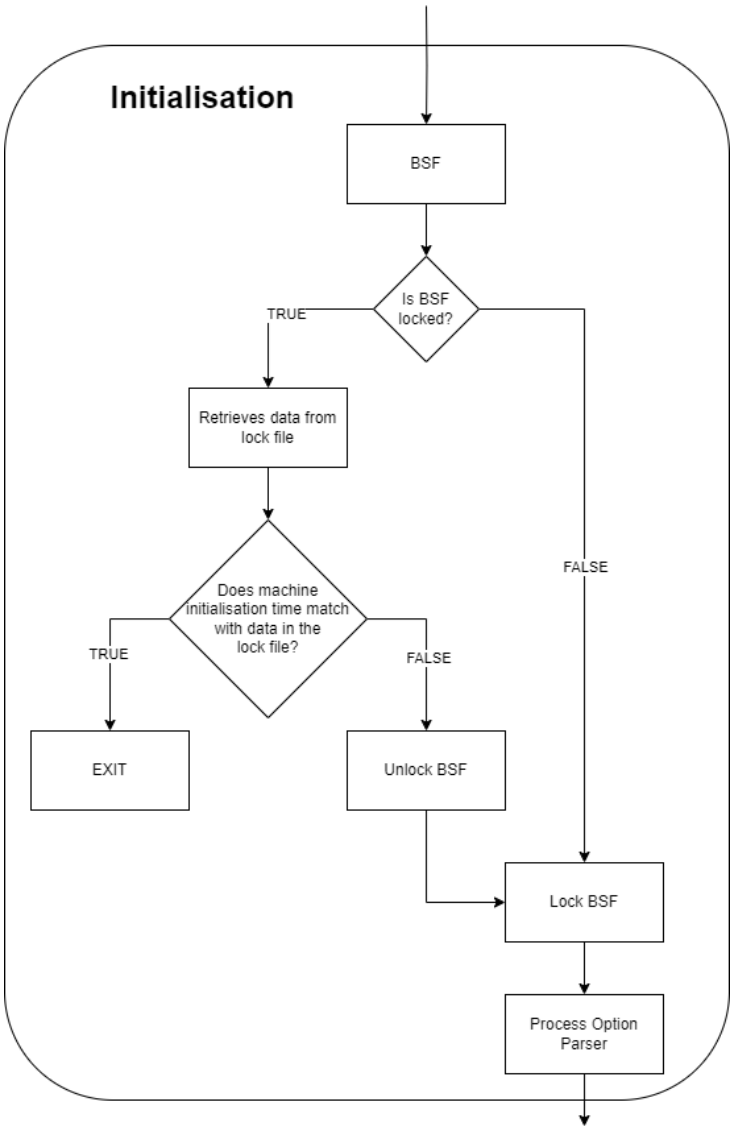


Figure 7: Initialisation flowchart

As shown in Figure 7, the procedure begins with an initial check to determine whether BSF is currently locked. If it is indeed locked, BSF retrieves data from a lock file, which contains relevant data pertaining to the machine's initialisation state. Subsequently, a second validation is performed to confirm if the machine's initialisation time corresponds to the data within the lock file. If a match is found, the framework exists successfully. In the event that the ma-

chine's initialisation time does not align with the data in the lock file, the process diverts to unlocking the framework. Following that, the framework secures itself by generating a lock file with the machine's initialisation data. Lastly, the flow transitions to the *OptionParser*, where the framework interprets developer input and executes the associated feature.

3.5.2 Finalisation Flowchart

The Finalisation phase is a universal component found in all feature flowcharts, symbolising the conclusion of the framework's execution, whether it's due to an exception or a successful completion.

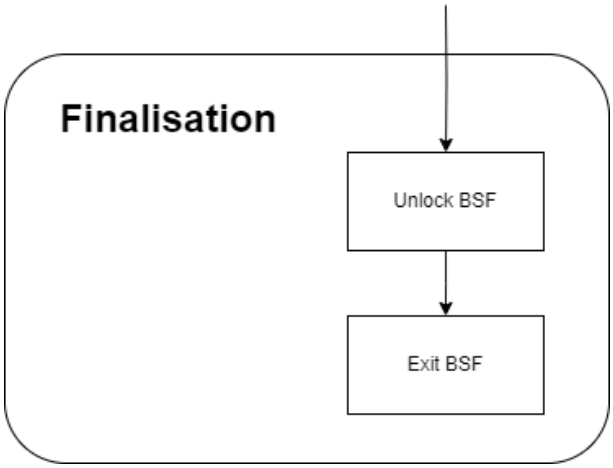


Figure 8: Finalisation flowchart

In the Finalisation phase, Figure 8, the framework is unlocked as it removes the lock file generated during the initialisation process and terminates the framework. This step marks the conclusion of BSF instance execution.

Init Feature

The *init* feature initialises a new BSF environment in a given directory. When executed, this feature sets up all necessary directory structures and configuration files that are essential for the application to begin execution.

Flowchart

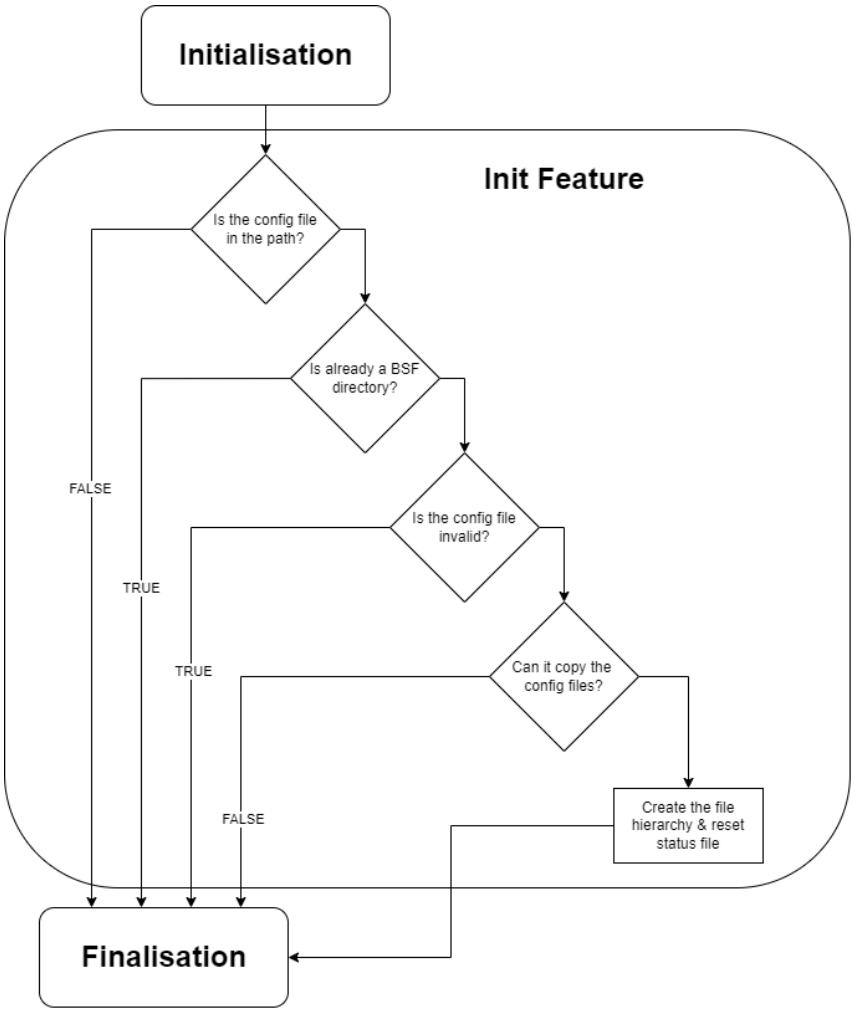


Figure 9: The flowchart outlines the workflow for the init feature.

During the initialisation phase, as illustrated in Figure 9, it commences with "Initialise" step, already described. Subsequently, it verifies whether the specified configuration folder contains the expected configuration file. If not, an exception is triggered, resulting in the process's termination. Several validations follow. Firstly, it checks whether the directory for creating the framework environment already exists as a BSF environment. In such a case, an exception is raised, ending the process. Additionally, it validates the provided configuration file's integrity, scrutinising all properties to ensure compliance with a defined structure. If any deviations are found, an exception is raised, leading to process termination. Once these validations are successfully completed, the process proceeds to copy all files and folders from the specified configuration folder to an internal configuration environment within the current directory. Any unforeseen issues during this copying process trigger an exception and halt the process. If everything proceeds smoothly, the framework extracts data from the newly acquired configuration file and generates a status file with all task statuses set to *Not*

executed.

In the finalisation phase, the *Finalise* step signifies BSF's unlocking, marking the conclusion of the workflow.

Syntax

```
$ <application> init <path/to/configuration_folder>
```

Use Examples

```
$ bsf init /home/user/config_folder
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

Execute Feature

The *execute* feature executes all the tasks' execute property in the configuration file.

Flowchart

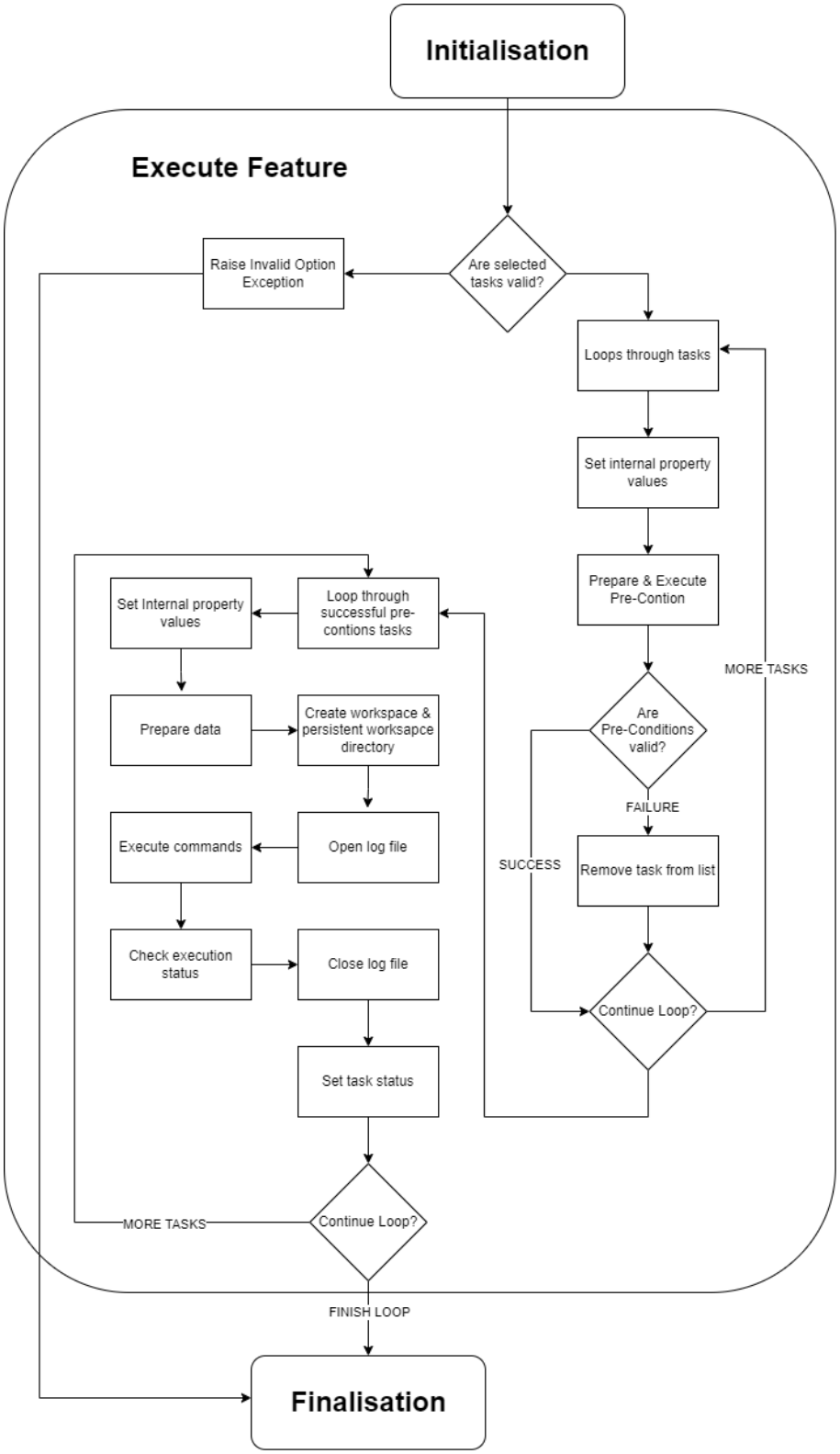


Figure 10: The flowchart represents a execute command flow.

As shown in Figure 10, it examines whether there are selected tasks to process. If tasks are identified, it advances to a series of operations designed to handle these tasks. Conversely, if there are no tasks, an exception is raised, leading to the termination of the process.

Within this series of operations, the process goes through a systematic assessment of each task. It involves a sequence of internal actions, including the integration of property values and the pre-conditions, the process either proceeds with the task or removes it from consideration. The choice to continue or conclude this task assessment loop hinges on specific criteria.

In this subsequent phase, the process undergoes another series of actions and evaluations. It prepares and analyses data, creates necessary directories, opens log files, communicates process messages, executes commands, and validates the execution status. Depending on the result of these series of actions, the process may either continue this phase of proceed to the final stage.

Syntax

```
1     $ <application> execute [tasks]
```

Use Examples

```
1     $ bsf execute
2     $ bsf execute task1
3     $ bsf execute task1 task2
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.
- **-y** or **–confirm**: If developer confirmation is required during the execution process.
- **-p** or **–parallel**: Execute each task concurrently.
- **-v** or **–verbose**: Display each command as it executes.
- **–ignore <number>**: Ignore a step accordingly to a given value.

Compare Feature

The *compare* feature provides developers with a report that examines the specified task's comparison among two or more previous iterations.

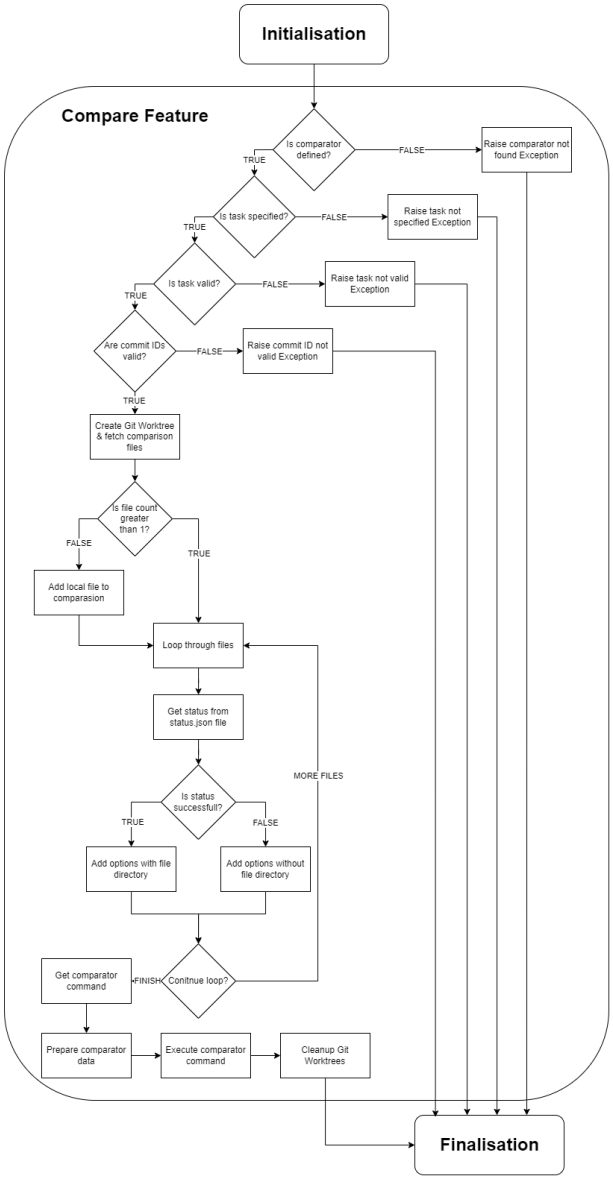


Figure 11: The flowchart represents the compare feature flow.

According to Figure 11, the process checks whether the *comparator* property is defined. If no *comparator* is defined, an exception is raised, and the process directs to the *Finalisation* step.

In the following step, it checks if a task is specified. If no task is specified, an exception is raised. If a task is specified, the process further checks if the task exists in the system. If that task exists, it proceeds to process options. If the task does not exist, an exception is raised,

leading to the *Finalisation* Step.

In the following, the process validates commit IDs using Git. If commit IDs are valid and exists in the Git repository, it proceeds to create a Worktree for processing based on the validated commit. In the presence of invalid commit IDs or those not found in the Git repository, an exception is raised, guiding the process to the *Finalisation* step.

After creating the Worktree, the process retrieves files. If there is only one file, it includes the local file and sets internal variables. In the presence of multiple files, it sets internal variables without adding a local file. The process iterates through the files, retrieves their status, and appends options based on their status.

Subsequently, the process acquires *comparator* commands, prepares data and executes these commands. The result is stored and Worktrees are cleaned up.

Syntax

```
1     $ <application> compare <task> <arguments>
```

Use Examples

```
1     $ bsf compare task01 53b7585a:3e2377d2
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

Publish Feature

The *publish* feature creates a new entry within the internal repository of BSF's environment, containing details of the currently executed tasks along with their corresponding publish_header messages used as commit message.

Flowchart

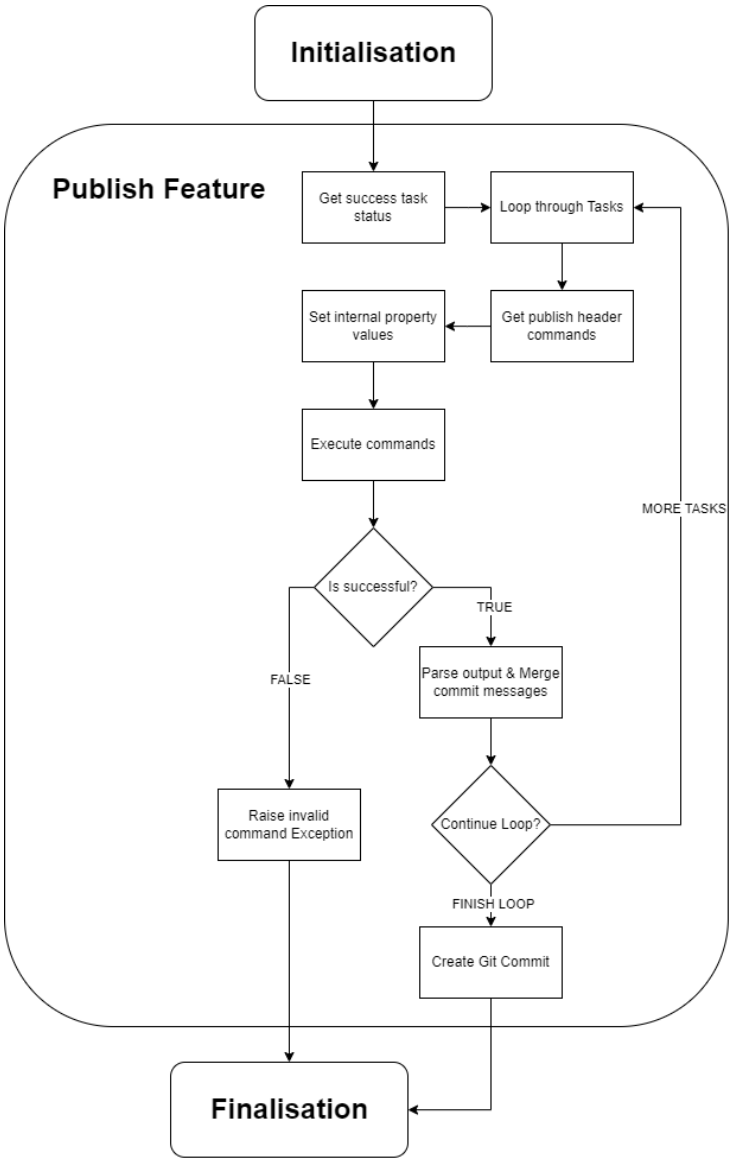


Figure 12: The flowchart outlines the workflow for an publish command.

As portrayed in the Figure 12, the framework retrieves the execution status of tasks from the status file. It selectively utilises tasks with a status value defined as *Passed*. With the task list established, it proceeds to traverse them. The process commences by retrieving the *publish_header* data from the configuration file for each task, configuring internal property values, and subsequently executing the associated commands. For each command execution, a validation check for success ensues. In cases of failure, an exception is raised, resulting in the termination of the BSF instance. Conversely, if the command is executed successfully, the output is pared and appended to a *hash map*. The loop then continues, repeating these steps until no tasks remain in the list. Upon exhausting the task list, BSF creates a Git commit

within its current internal environment.

Syntax

```
$ <application> publish
```

Use Examples

```
$ bsf publish
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

3.6 Auxiliary Scripts

The auxiliary scripts are designed to complement BSF, creating a comprehensive building and testing development environment.

3.6.1 Get Versions Script

The script is designed to analyse a user-provided `config.log` file and generate a versions report. It processes command-line arguments to identify the input file and tool name, extracting data from the log file.

```
1  $ head config.log
2  This file contains any messages produced by compilers while
3  running configure, to aid debugging if configure makes a
       mistake.
4
5  It was created by arc-toolchain configure 1.0, which was
6  generated by GNU Autoconf 2.71.  Invocation command line was
7
8    $ path/to/toolchain/configure --prefix=path/to/install --with
       -gcc-src=path/to/gcc/source --with-binutils-src=path/to/
       binutils/source
9
10 ## --------- ##
11 ## Platform. ##
```

Listing 3.3: Excerpt from a configured Toolchain "config.log" file

The Listing 3.3 displays a header section of a `config.log` file. This type of file are commonly generated during the configuration process of a software projects using tools like *Autoconf*.

The focus of this file are the included details about the invocation command-line used to run the configure of the Toolchain. In this case, the command is shown as:

```
1  $ path/to/toolchain/configure --prefix=path/to/install     \
2        --with-gcc-src=path/to/gcc/source                    \
3        --with-binutils-src=path/to/binutils/source
```

Listing 3.4: Excerpt with the configure command from the "config.log" file

The line, represented in 3.4, indicates how the `configure` script was invoked, specifying various configuration options such as the installation prefix and the paths to the source code GCC and Binutils as a example. These options determine how the Toolchain will be config-

ured and built.

The script scans the header of the `config.log` file for a line that starts with `$` (representing the line explained previously) and collects data related to all repository paths, including the Toolchain repository. For each repository path, it retrieves details like the remote repository *URL*, the current *branch*, the commit ID and checks for uncommitted changes. These details are then structured into a JSON object with the tool name as the key and printed in a human-readable JSON format.

The following example is the JSON format output from the script.

```
$ get_versions.rb -f ./config.log -n archs-gcc-baremetal
{
  "archs-gcc-baremetal": {
    "build_date": "2023-08-21 14:56:52",
    "configure": "path/to/toolchain/configure --prefix=path/to/
       install --with-gcc-src=path/to/gcc/source --with-
       binutils-src=path/to/binutils/source",
    "arc-gnu-toolchain": {
      "repository": "https://github.com/foss-for-synopsys-dwc-
         arc-processors/arc-gnu-toolchain",
      "branch": "master",
      "hash": "139f7b8f5143767330bce1a2c53fdc0239181fd9",
      "patch": false
    },
    "gcc": {
      "repository": "https://github.com/foss-for-synopsys-dwc-
         arc-processors/gcc",
      "branch": "arc64",
      "hash": "3fedceb05cfe5f319629c4560f5a3db7fc41e442",
      "patch": false
    },
    "binutils-gdb": {
      "repository": "https://github.com/foss-for-synopsys-dwc-
         arc-processors/binutils-gdb",
      "branch": "arc64",
      "hash": "22b1bf47268e4e63dc7af8c810dbb0aab1fabdb5",
      "patch": false
    },
    "newlib": {
      "repository": "https://github.com/foss-for-synopsys-dwc-
         arc-processors/newlib",
      "branch": "arc64",
      "hash": "2921a0dd103428ad939bfea7299f0329669cf958",
      "patch": false
```

```
29      }
30    }
```

Listing 3.5: Output from the Get Versions Script.

As shown in Listing 3.5, this JSON format file holds all the essential data required for recreating the built Toolchain if the need arises. This file is used within the publish header to disseminate information regarding the testing tools used.

3.6.2 Check Versions Script

When the preceding script hasn't been executed, meaning that for a particular testing scenario the Toolchain, simulator, or other essential tools haven't been built using the current system, an alternative approach is necessary to determine the versions of these tools. To address this, a script has been devised to extract information from the `-version` command of these tools. For instance, if a test utilises a Toolchain that wasn't built using BSF (signifying the absence of the JSON format file containing tool information), the version command for GCC is invoked, and its output is subsequently parsed.

```
1 $ arc-elf32-gcc --version
2 arc-elf32-gcc (GCC) 12.2.1 20220829
3 Copyright (C) 2022 Free Software Foundation, Inc.
4 This is free software; see the source for copying conditions.
     There is NO
5 warranty; not even for MERCHANTABILITY or FITNESS FOR A
     PARTICULAR PURPOSE.
```

Listing 3.6: Output from "gcc –version" command

Illustrated in Listing 3.6, the first line is parsed to extract the version number and the build date, which are then structured into a JSON format file. In the given example, `12.2.1` represents the version, while `20220809` signifies the build date. The result is presented as follows in Listing 3.7:

```
1 $ version_check.rb -t gcc "$(arc-elf32-gcc -v 2>&1)"
2 {
3   "arc-elf32-gcc": {
4     "build_version": "12.2.1",
5     "build_date": "20220829"
6   }
```

```
7 }
```

Listing 3.7: Output from Check Versions Script.

This approach ensures that even when BSF hasn't been used to build the tools, their versions can still be accurately identified and stored in a structured JSON format file for further reference.

3.6.3 testsuite-filter Script

The *testsuite-filter* script acts as a practical tool for analysing and filtering the results of a test suite. The filtering component of this script is at the core of its functionality. It plays a role in shifting through test results and scenarios, allowing for the extracting and classification of pertinent information based on predefined criteria outlined in a allowlist JSON file.

The filtering operation commences with the definition of criteria set within a allowlist JSON file. This allowlist contains specific rules and conditions that dictate which test results and scenarios should be retained and which should be discarded. The script then traverses the test scenarios, evaluating them against the criteria established in the allowlist. It is designed to discern and preserve those test scenarios that match the criteria, ensuring that only relevant information is retained for further analysis.

As the script processes test results, it classifies them into distinct categories. These categories typically include standard DejaGNU labels like *PASS*, *FAIL*, *XPASS*, and so on. One of the critical objectives of this filtering process is the identification of errors or unexpected test results.

```
1 $ testsuite-filter newlib ./allowlist/gcc.json \
2   `find build-gcc-newlib-stage2 -name g*.sum |paste -sd "," -`
3 === g++ gcc: Unexpected fails for arc-sim ===
4
5 === FAIL ===
6     c-c++-common/Wcast-align.c  -Wc++-compat  (test for excess
          errors)
7     gcc.dg/strlenopt-10.c scan-tree-dump-times strlen1 "memcpy
          \\(" 7
8     gcc.dg/strlenopt-11.c scan-tree-dump-times strlen1 "memcpy
          \\(" 6
```

```
 9      gcc.dg/strlenopt-13.c scan-tree-dump-times strlen1 "memcpy
           \\(" 6
10      c-c++-common/Wcast-align.c  -std=gnu++98 (test for excess
           errors)
11      c-c++-common/Wcast-align.c  -std=gnu++14 (test for excess
           errors)
12      c-c++-common/Wcast-align.c  -std=gnu++17 (test for excess
           errors)
13      c-c++-common/Wcast-align.c  -std=gnu++20 (test for excess
           errors)
14      g++.dg/contracts/contracts-tmpl-spec2.C   output pattern
           test
15  === XPASS ===
16
17
18  # Tests | # Detected Errors
19  --------+------------------
20   340081 |        9
```

Listing 3.8: Output from testsuite-filter Script.

As indicated in the Listing 3.8, the script diligently calculates the count of detected errors. To facilitate clear communication and analysis, the script then arranges the filtered results in a structured and comprehensible format.

The inspiration for this concept was drawn from the riscv testsuite-filter.

3.6.4 Compare Script

The *compare* script is a tool designed for the analysis and comparison of test results generated during a DejaGNU execution. It performs a detailed examination of two sets of test suite results, and provides insights into the differences and changes observed between two results test suites.

The script begins by handling the input files, which contain test results in a DejaGNU summary (.sum) structured format. It ensures that the files exist and reads their contents. It parses the test results from the input files, organising them into a structured data format. Test results are categorised based on their results, such as as *PASS*, *FAIL*, *XPASS*, *XFAIL*, *UNRESOLVED*, or *UNSUPPORTED*.

The core of the script lies in the comparison and analysis of test results. It systematically compares corresponding test cases between the test suites to identify changes in results. This includes scenarios where tests have transitioned from *PASS* to *FAIL* or vice versa, newly added tests, or tests that have been removed.

The script categorises the observed changes into different types, including:

- **New Failures**: Tests that were previously passing but now fail.
- **New Passes**: Tests that were previously failing but now pass.
- **Added Tests**: Newly introduced tests.
- **Removed Tests**: Tests that were present in the previous results but are now absent.

The script compiles summary statistics, such as the total count of tests results in the provided files as well as the count of each type of change (new failures, new passes, added tests, and removed tests).

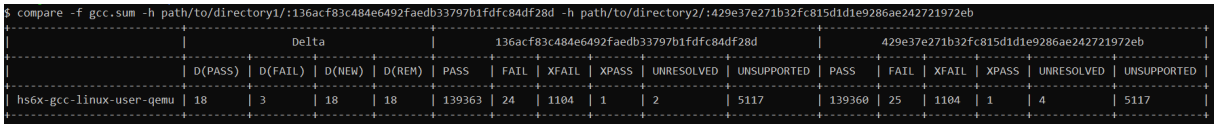


Figure 13: Output from Compare Script.

It offers flexibility in output formats through a user-defined parameter. It can generate a JSON output for structured data representation, textual output in tabular format as plain text, as presented in Figure 13, and as HTML support. In addition to providing output formats for the command-line interface, the compare script also offers the functionality to send an email report containing the generated information to one or multiple recipients, either directly or through CC.

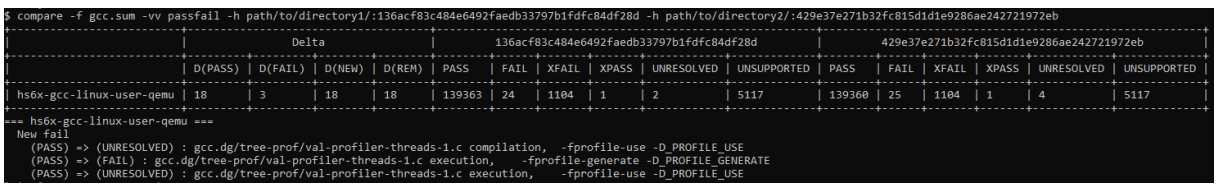


Figure 14: Output from Compare Script with Verbose.

The script can operate in verbose mode, as shown in Figure 14, providing additional information and filtering actions. It includes customisable filtering options to focus on specific types of changes, such as filtering out new passes, new failures, added tests, removed tests, passing to failures, or failures to passes.

3.7 Validating BSF

This section presents an in-depth validation of BSF. It starts with a step-by-step installation guide; a clear and practical understanding of the process is provided. Following that, a real-world case study will be presented, demonstrating BSF in action to reinforce comprehension of its practical applications. Lastly, a discussion of the software tests used to ensure that BSF operates properly.

BSF Installation

BSF can be found at:

```
1 $ git clone https://github.com/luismgsilva/testing-framework \
2     --branch stable
```

Prior to utilising BSF, the installation of *Ruby* is necessary. For those on Debian or Ubuntu, Ruby and its essential components can be installed with the execution of the following command:

```
1 $ apt-get install ruby-full
```

BSF relies on several *Ruby* gems to function properly. Execute the following command to install the required gem dependencies:

```
1 $ gem install erb json sinatra terminal-table
```

Using BSF with a Real Case Study

To validate that BSF functions as expected, a case study has been configured for Synopsys ARC CPU architecture. These configurations serve two primary purposes: assess the building of the necessary Toolchains and simulators for cross-compile testing; the other to set up the testing environment itself.

These configurations make extensive use of the previously mentioned auxiliary scripts.

Build

The configuration can be accessed through the following command:

```
1 $ git clone  \
2     https://github.com/luismgsilva/thesis_builder_tutorial \
3     $HOME/build_config
4 Cloning into '$HOME/build_config'...
```

The initialisation of the build environment is done using the `init` feature. It requires specifying a configuration folder containing a mandatory and valid `config.json` file.

```
# $HOME/builder
$ bsf init $HOME/build_config
$ ls -a
.  ..  .bsf
```

Available tasks and their descriptions in the configuration can be seen using the following feature:

```
# $HOME/builder
$ bsf tasks
archs-gcc-baremetal:
    Description: Task responsible to configure and build ARC-
   Toolchain for ARCHS CPU
arc-qemu:
    Description: Task responsible to configure and build ARC
   QEMU
```

After initialising BSF, the initial step is to set up all the necessary user-defined property values. To list all the available, execute the following:

```
# $HOME/builder
$ bsf vars
Input Variable PREFIX not defined
Input Variable MOD_VERSION not defined
Input Variable NJOBS not defined
Input Variable MOD_PREFIX not defined
```

There are four input variables that need to be defined, shown in Table 6:

Variable	**Description**
PREFIX	Represents the path to install the tools.
MOD_VERSION	Creates environment module files; this variable sets the version name.
NJOBS	Represents the maximum number of processes to be used.
MOD_PREFIX	Represents the path to the environment module file.

Table 6: Defining the significance of every available user-defined property value.

To define each of the input variables, execute the following:

```
# $HOME/builder
$ bsf set PREFIX=$HOME/builder/install
$ bsf set MOD_PREFIX=$HOME/builder/mod
$ bsf set MOD_VERSION=20230601
$ bsf set NJOBS=$(nproc)
```

Now, the values of the user-defined property values can be viewed:

```
1 # $HOME/builder
2 $ bsf vars
3 Input Variable PREFIX defined: $HOME/builder/install
4 Input Variable MOD_VERSION defined: 20230601
5 Input Variable NJOBS defined: 32
6 Input Variable MOD_PREFIX defined: $HOME/builder/mod
```

Upon definition, the execution of the tasks can now proceed.

```
1 # $HOME/builder
2 $ bsf execute
3 One or more pre-conditions have failed:
4 - archs-gcc-baremetal: test -d $HOME/builder/sources/arc-gnu-
     toolchain/.git
5 - arc-qemu: test -d $HOME/builder/sources/arc-qemu/.git
```

In the validation process, multiple pre-condition failures are encountered, prompting the confirmation of the presence of the pertinent sources. These sources have been configured to be located in the *source* folder using the *soruces get* feature.

```
1 # $HOME/builder
2 $ bsf sources get --shallow
3 Cloning into '$HOME/builder/sources/arc-gnu-toolchain'...
4 Cloning into '$HOME/builder/sources/arc-qemu'...
```

With the necessary *sources* in place, the execution of all tasks may now proceed.

```
1 # $HOME/builder
2 $ bsf execute
3 Executing archs-gcc-baremetal..
4 Failed
5 Executing arc-qemu..
```

As a task has been stated as *Failed*, the option exists to either allow the rest of the execution to continue or stop it to address the issue. The *log* feature can be used to examine the cause of the problem.

```
1 # $HOME/builder
2 $ bsf log archs-gcc-baremetal
3 BSF Executing: $HOME/builder/sources/arc-gnu-toolchain/
     configure --prefix=$HOME/builder/install/archs-gcc-baremetal
     /20230601 --target=arc --with-cpu=archs --disable-werror
4 sh: 3: /home/luis/builder/sources/arc-gnu-toolchain/configure:
     not found
```

The issue arises because, up to the current version, it is necessary to generate the configure file used to configure the Toolchain.

```
# $HOME/builder
$ cd ./sources/arc-gnu-toolchain
$ autoconf
```

In addition to the *execute* feature, the *log* feature with the *-f* (or *–follow*) flag can be utilised in a separate terminal to monitor the progress of a task in real-time as it unfolds (e.i *$ bsf -C $HOME/builder log –follow archs-gcc-baremetal*).

With everything set up currently, proceed to execute all tasks.

```
# $HOME/builder
$ bsf execute
Executing archs-gcc-baremetal..
Passed
Executing arc-qemu..
Passed
```

After the successful completion of all tasks with a *Passed* status, the build process is finalised. In case any of the tasks are marked as *Failed*, the same procedure utilised earlier should be followed to diagnose and rectify the issue.

```
# $HOME/builder
$ cat ./install/archs-gcc-baremetal/20240106/archs-gcc-
   baremetal.json
{
  "archs-gcc-baremetal": {
    "configure": "/data/luis/tmp/validate_thesis/builder/
       sources/arc-gnu-toolchain/configure --prefix=/data/luis/
       tmp/validate_thesis/builder/install/archs-gcc-baremetal
       /20240106 --target=arc --with-cpu=archs --disable-werror
       ",
    "build_date": "2024-01-06 18:45:47",
    "arc-gnu-toolchain": {
      "repository": "https://github.com/foss-for-synopsys-dwc-
         arc-processors/arc-gnu-toolchain.git",
      "branch": "master",
      "hash": "0da6c8f6c5e844d3bf297f17c3c36712a461993a",
      "patch": false
    },
    "gcc": {
      "repository": "https://github.com/foss-for-synopsys-dwc-
         arc-processors/gcc.git",
      "branch": "arc64",
      "hash": "38794ca06709f3e50b2be00bad8a25bb0e811852",
      "patch": false
    },
    "newlib": {
```

```
20        "repository": "https://github.com/foss-for-synopsys-dwc-
              arc-processors/newlib.git",
21        "branch": "arc64",
22        "hash": "2921a0dd103428ad939bfea7299f0329669cf958",
23        "patch": false
24      },
25      "binutils-gdb": {
26        "repository": "https://github.com/foss-for-synopsys-dwc-
              arc-processors/binutils-gdb.git",
27        "branch": "arc64",
28        "hash": "22b1bf47268e4e63dc7af8c810dbb0aab1fabdb5",
29        "patch": false
30      }
31    }
32  }
```

Listing 3.9: Output Versions Script.

As an shown in Listing 3.9, for each task, a JSON format file is created in the install folder, containing all the versions used during each execution.

Testing

Distinct from the build environment, where a folder containing the building configuration is used, in this case, a repository with a pre-existing testing environment that has undergone prior testing iterations will be applied. To initiate this process, execute the following command:

```
1  # $HOME/testing
2  $ bsf clone  \
3      https://github.com/luismgsilva/thesis_tester_tutorial \
4      $HOME/testing
5  $ ls -a
6  .  ..  .bsf
```

Similar to the build environment, the first step is to review all the input variables. In this scenario, since the configuration has been imported from a different machine, the input variables already possess values. However, these values may not align with our requirements. In this case, the objective is to adjust the current value of the *NJOBS* user-define variable to match the requirements of our machine.

```
1  # $HOME/testing
2  $ bsf vars
3  Input Variable NJOBS defined: 32
4  $ bsf set NNJOBS=$(nproc)
```

Upon proceeding to the execute phase, the pre-conditions validation will be prompted once again. In this case, there are dependencies for the sources of the respective Toolchains, including scripts that are essential for testing, as well as for the tools themselves because they house the respective test suites.

```
# $HOME/testing
$ bsf execute
One or more pre-conditions have failed:
- archs-gcc-baremetal-qemu:
    - test -d $HOME/testing/sources/arc-gnu-toolchain/.git
    - test -d $HOME/luis/delete/testing/sources/arc-gcc/.git
    - which arc-elf32-gcc
    - which qemu-system-arc
    - which runtest
```

It is advisable to utilise the same source versions that was employed in the building of the Toolchain. This practice helps ensure consistency and compatibility throughout the development and usage of the test suites.

```
# $HOME/testing
$ bsf sources get --shallow
Cloning into '$HOME/testing/sources/arc-gnu-toolchain'...
Cloning into '$HOME/testing/sources/gcc'...
```

As for the tools, either the recently created environment modules, as per Furlani (1991), can be loaded or the configuration can be done in the required environment variables.

```
$ module use --append $HOME/builder/mod
$ module load archs-gcc-baremetal/20230601 arc-qemu/20230601
```

After ensuring the validity of each pre-condition, it is now time to initiate the testing for each task.

```
# $HOME/testing
$ bsf execute
Executing archs-gcc-baremetal..
Passed
```

At this point, the system is prepared to compare past iterations that have been executed. Tasks can be performed individually or collectively to generate a report.

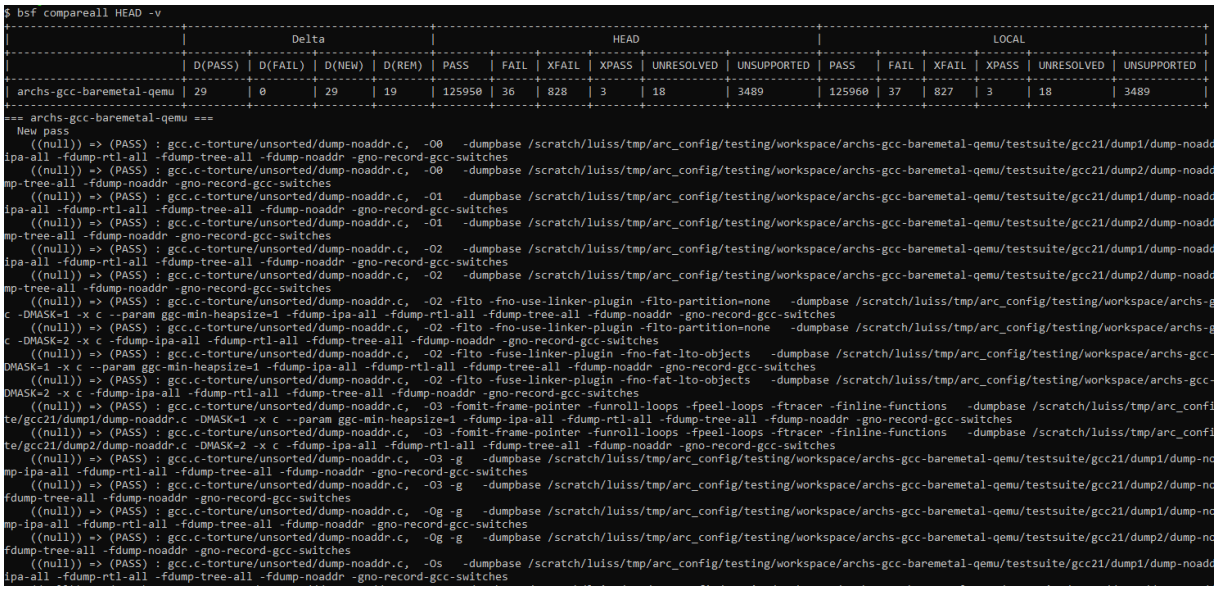


Figure 15: BSF Output for Compare Feature between HEAD and LOCAL with verbose.

In the figure 15, a comparison is drawn between the most recent past iteration executed, specifically referred to as "HEAD", and the locally executed iteration, denoted as "LOCAL."

Comparing two distinct executed iterations is also feasible. For instance, the current configuration environment includes test executions for two ARC releases: arc-2023.03 and arc-2023.09.

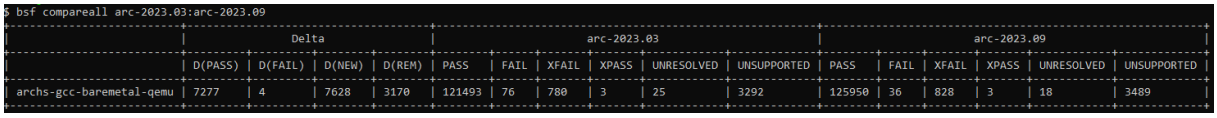


Figure 16: BSF Output for Compare Feature between two releases: arc-2023.03 and arc-2023.09.

In Figure 16, both releases are explicitly denoted as Git Tags within the same BSF feature, although the corresponding commit IDs can also be used for the purpose of comparing them.

3.7.1 Software Tests

Quality assurance in software engineering holds significant importance, serving as a process to validate the proper functionality and alignment with user expectations. The fundamental objective of engaging in software testing lies in the identification of defects, errors, and imperfections within the software. Various types of tests exist, including unit tests, functional

tests, system tests, among others. This thesis, however, exclusively addresses functional tests.

Functional Tests

Functional testing is a vital stage in software development that focuses on evaluating an application's essential functionalities by subjecting them to a predetermined set of guidelines. This testing procedure is meant to guarantee that the software operates as intended, according to the defined criteria and delivering the expected features.

Several functional tests have been developed to verify the behavior of BSF, ensuring that it functions according to expectations. These tests cover each potential component of BSF functionality, executing actions from initialisation to feature validation.

Below are presented two examples on how the BSF functional tests work.

Lock Test

```
#!/bin/bash

# This functional test verifies the locking system by running a
# sleep command within BSF and then executing BSF once more.

# Define the expected output
expected_output=$(cat <<EOF
bsf: Could not get lock
EOF
)

# Create the required directories
mkdir config/ ws/

# Create the configuration file
cat <<EOF > config/config.json
{
        "tasks": { "test01": { "execute": "sleep 5" } }
}
EOF

# Initialize and execute BSF
bsf -C ws/ init $(pwd)/config/ > /dev/null
(bsf -C ws/ execute &> /dev/null) &
sleep 1
```

```
26
27 # Get the actual output from BSF execute commmand
28 output=$(bsf -C ws/ execute)
29
30 # Check if the actual output matches the expected output
31 if [ "$output" = "$expected_output" ]; then
32     echo "Passed: $(basename "$0")"
33 else
34     echo "Failed: $(basename "$0")"
35 fi
```

Listing 3.10: Lock Functional Test

The depicted functional test in Listing 3.10 assesses the proper functioning of the locking mechanism. It initiates a fresh BSF environment, generating a task designed to keep a BSF instance idle for 5 seconds. In this interval, a separate BSF instance is executed, and it is expected to fail owing to the presence of an existing BSF instance.

User-Defined Property Value Test

```
1  #!/bin/bash
2
3  # This functional test validates the functionality of input
       variable definition.
4
5  # Define the expected output
6  expected_output=$(cat <<EOF
7  Input Variable VARIABLE defined: value
8  EOF
9  )
10
11 # Create the required directories
12 mkdir config/ ws/
13
14 # Create the configuration file
15 cat <<'EOF' > config/config.json
16 {
17         "tasks": {
18                 "test01": {
19                         "execute": "echo '$var(VARIABLE)'"
20                 }
21         }
22 }
23 EOF
24
25 # Initialize and execute BSF
26 bsf -C ws/ init $(pwd)/config/  > /dev/null
27 bsf -C ws/ set VARIABLE=value   > /dev/null
```

```
28
29 # Get the actual output from BSF vars commmand
30 output=$(bsf -C ws/ vars)
31
32 # Check if the actual output matches the expected output
33 if [ "$output" = "$expected_output" ]; then
34     echo "Passed: $(basename "$0")"
35 else
36     echo "Failed: $(basename "$0")"
37 fi
```

Listing 3.11: User-Defined Property Values Functional Test

The functional test illustrated in Listing 3.11 verifies the accurate definition of *User-Defined Property Values*. This test begins by establishing a new BSF environment, setting *VARIABLE* to *value*, and subsequently confirming the correctness of its assigned value.

3.8 CI with GitHub Webhooks and BSF External API

Prior to implementing the external API, the entire framework testing process was highly manual. Developers had to carry out the multiple steps themselves, starting with manually executing BSF to compile the tool. After that, they had to run BSF again separately to perform testing. Finally, the developers had to manually inspect the results and compare them with previous iterations.

Clearly, this manual approach was time-consuming. It also lacked efficiency and scalability, especially when dealing with frequent updates and iterations to existing tools. As a result, there was a pressing need to automate this entire process and streamline the development workflow.

By including an external API alongside with BSF and leveraging GitHub Webhooks, the testing can become more automated. When specific events occur in the development process, such as code commits or pull requests, the external API and Webhooks can trigger the necessary actions automatically. This includes building the tool, executing testing procedures, and comparing results with baselines, and providing a report.

3.8.1 External API Implementation and Integration with Webhooks

The proposed approach for establishing a CI environment with the BSF involves leveraging GitHub Webhooks. When a specific event occurs in a particular tool, the GitHub Webhook will notify the server hosting the external API. This API plays a central role in managing the subsequent actions required to initiate interactions with BSF instances.

One of these instances is responsible for the build process of the tool. It handles tasks such as creating necessary dependency files (e.g. module files) and collecting data about the versions used in that build, utilising the methods discussed earlier. Upon successful completion of this build task, a new request is sent to the second instance, which is in charge of testing the newly built tool.

The testing instance sets up the necessary environment by loading the built tool and the latest versions of other required tools into the environment. It then proceeds to compile and execute the testing process. Once the corresponding testing is finished, a comparison is triggered between the just-executed test and a predefined baseline for comparison. The results of this comparison are then sent to the developer who triggered the initial event.

All the essential information required for this CI process is contained within the message request received from GitHub. By combining GitHub Webhooks and the external API, developers can establish an automated CI pipeline that efficiently manages builds, testing, and result notifications, thereby streamlining the development workflow and ensuring prompt feedback to the development team.

When the developer receives the email with the testing report, it contains a detailed testing report highlighting any unexpected failures encountered during the execution. This report is generated by a test suite filter designed to identify known fails and errors in specific tests, aiding in the discovery of new, previously unknown failures during testing. Alongside the report, the email also includes a comparison with a baseline, which outlines the number of each type of test result and highlights the differences between them. This comparison clearly illustrates the variations between one iteration and another, allowing developers to pinpoint precisely what has changed and identify potential issues introduced during the development process. Overall, the email notification provides valuable insights into the testing results, making it easier for developers to understand the testing status and take appropriate actions to address any identified issues effectively.

3.8.2 Requirements Analysis

Functional and Non-Functional Requirements

In this section, the same approach is followed as in the previous BSF Requirements Analysis sections. Therefore, the functional and non-functional requirements are presented.

Functional Requirements

This section will present the functional requirements using use cases. These use cases will outline the various services available to the system's actors, providing a clear understanding of the application's functionalities.

Authentication Request

The *login* request initialises the process of logging in a known user in the system.

#	Request and Type	Request Data
1	POST /login	Description Request sent when a valid *user* wishes to authenticate on the system using a password. Request Data A JSON string with the format { "username": *user_name*, "password": *password* } where *user_name* is a string that contains the user's unique identification and password contains the authentication credentials for the user.

Table 7: Use Case API representing the *login* POST request type.

Register Request

The *register* request initialises the process of creating a new user in the database.

#	Request and Type	Request Data
2	POST /register	Description Request sent by the administrator to create a new user in the system. Request Data A JSON string with the format { “username”: *user_name*, “password”: *password*, "permissions": *permissions* } where *user_name* is a string that contains the user’s unique identification, *password* contains the authentication credentials for the user and *permissions* contain a list of permissions for the user.

Table 8: Use Case API representing the *register* POST request type.

Change Permissions Request

The *change_permissions* requests initialises the process of modifying the permissions of a specified user.

#	Request and Type	Request Data
3	PUT /change_permissions	Description Request sent by a authenticated and eligible superuser to change the permissions of a user in the system. Request Data A JSON string with the format { “username”: *username*, “permissions”: *permissions* } where *username* is a string that contains the user name and *permissions* the permissions to be changed.

Table 9: Use Case API representing the *change_permissions* PUT request type.

Kill Task Request

The *kill* request initialises the termination process of an ongoing task execution.

#	Request and Type	Request Data
4	POST /kill_task	Description Request sent by a authenticated and eligible user to kill a task in the system. Request Data A JSON string with the format { “task_id”: *task_id* } where *task_id* is a string that contains the task’s unique identifier.

Table 10: Use Case API representing the *kill* POST request type.

Start Task Request

The *start* request initialises a new task execution workflow associated with the tool that triggered the HTTP request.

#	Request and Type	Request Data
5	POST /start_task /webhook	Description Request sent when a user or a CI system wishes to start a new task execution. Request Data A JSON string with the format { “repository”: { “name”: *repository_name* }, “pusher”: { “name”: *user_name* }, “ref”: *ref*, “after”: *after*, “before”: *before* } where *repository_name* is a string signifying the repository name (e.g., GCC, Binutils), *user_name* is a string representing the user who initiated the event. The *ref* is a string indicating the updated branch, *after* is a string denoting the post-commit ID, and *before* is a string indicating the pre-commit ID.

Table 11: Use Case API representing the *start_task* POST request type.

List Tasks Request

The *tasks* request provides the list of all known tasks in the system.

#	Request and Type	Request Data
6	GET /tasks	Description Request sent when a valid user wishes to get a list of all the tasks requests in the system. Return Data A JSON string with the format

Table 12: Use Case API representing the *tasks* GET request type.

List Machines Request

The *machines* request provides the list of all known machines in the system.

#	Request and Type	Request Data
7	GET /machines	Description Request sent when a valid user wishes to get a list of all the machines in the system. Return Data A JSON string with the format

Table 13: Use Case API representing the *machines* GET request type.

CI: Update Task Status Request

The *ci_tasks* PUT request is used by the CI System to update the execution status of a specified task.

#	Request and Type	Request Data
8	PUT /ci_tasks	Description Request sent when a valid CI system wishes to update the execution status of a given task in the system. Request Data A JSON string with the format { “task_id”: *task_id*, “exit_code”: *exit_code* } where *task_id* is a string that contains the task’s unique identifier and *exit_code* is a string that holds the task’s execution exit code.

Table 14: Use Case API representing the *ci_tasks* PUT request type.

CI: Get Task Data Request

The *ci_tasks* GET request is used by the CI System to acquire the information of a specified task.

#	Request and Type	Request Data
9	GET /ci_tasks	Description Request sent when a valid CI system wishes to get the data of a given task in the system. Return Data A JSON string with the format { “task_id”: *task_id*, “task_name”: *task_name*, “ref”: *ref*, “before”: *before*, “after”: *after* } where *task_id* is a string that contains the task’s unique identifier, *task_name* is a string signifying the repository name (e.g., GCC, Binutils). The *ref* is a string indicating the updated branch, *after* is a string denoting the post-commit ID, and *before* is a string indicating the pre-commit ID.

Table 15: Use Case API representing the *ci_tasks* GET request type.

CI: Update task process ID Request

The *ci_update_process_id* request is used by the CI System to update the process id associated with the task currently in progress.

#	Request and Type	Request Data
10	PUT /ci_update_process_id	Description Request sent when a valid CI system wishes to update the process id of a given task in the system. Request Data A JSON string with the format { “task_id”: task_id, “process_id”: process_id } where *task_id* is a string that contains the task’s unique identifier and *exit_code* is a string that holds the task’s execution exit code.

Table 16: Use Case API representing the *ci_update_process_id* PUT request type.

Non-Functional Requirements

Throughout the development of the external API, consideration was given to its non-functional requirements, ensuring that the application meets performance, security, and usability standards, among others.

Programming Language

Unlike the programming language selected for BSF, *Python* was chosen as the programming language for the external API. In this context, there is a need for specific tools and libraries tailored to *Python*’s ecosystem. *Python*’s versatility and extensive library support make it an ideal choice for developing the API component of the system.

FastAPI

FastAPI emerged as the chosen solution for web API development due to its state-of-the-art features. The decision to adopt FastAPI was driven by its revolutionary capabilities in simplifying the creation of APIs using Python. Its ability to assure optimal performance and ease of management while upholding high standards of code quality and efficiency was critical in

the selection process.

Database

For the database it was chosen SQLAlchemy. SQLAlchemy is a widely-used Python library designed for working with relational databases. It provides a flexible and robust Object-Relational Mapping (ORM) framework. The ORM component utilises a declarative syntax, where the definition of the database schema uses Python classes. Each Python class corresponds to a table in the database, and SQLAlchemy handles the generation of the necessary SQL queries for creating and interacting with these tables.

3.9 External API Architecture

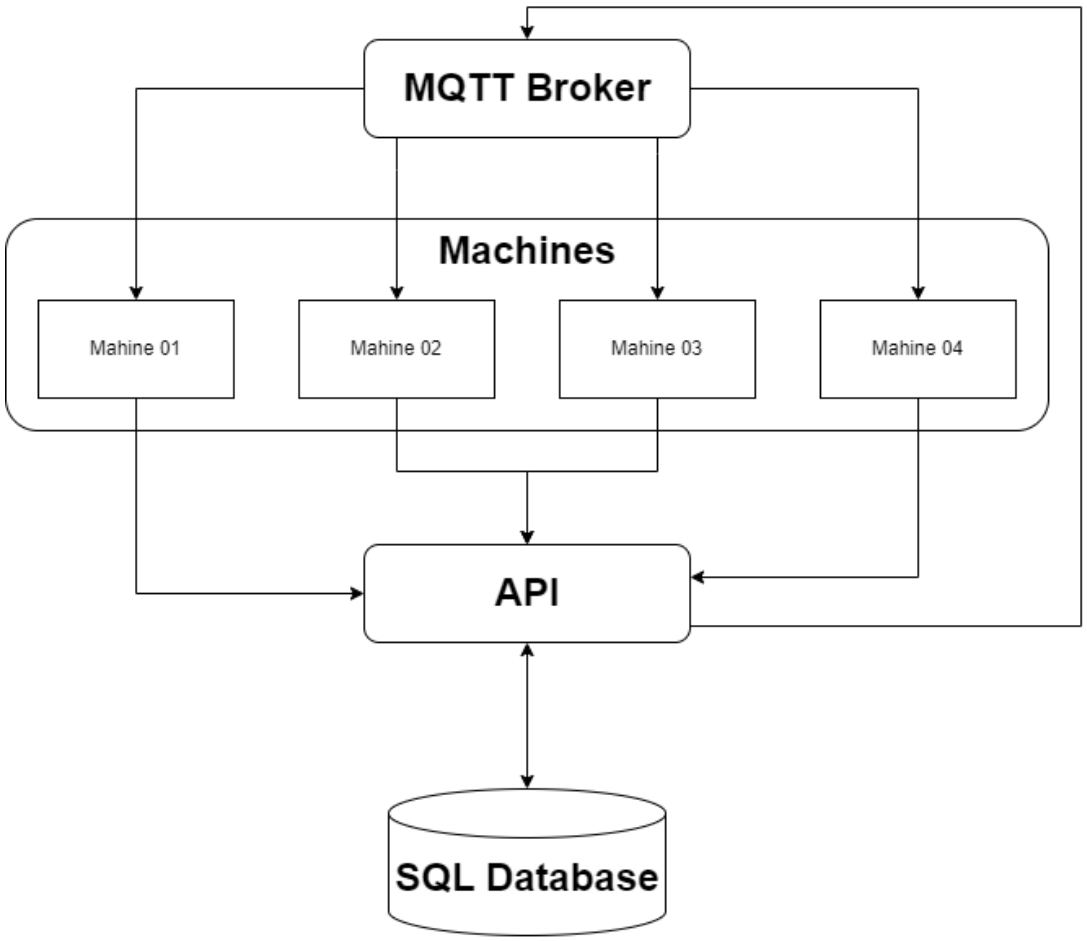


Figure 17: External API Architecture.

3.9.1 Overall System Architecture

As shown in Figure 17, at the heart of the system lies the API component. This component functions as the gateway for external users to interact with the system. The SQL database is responsible for storing and managing data structures, with the API directly communicating to perform CRUD (Create, Read, Update, Delete) operations on the stored data as needed. The *MQTT* Broker, as described in OASIS (2019), manages and facilitates communication among different machines in the system. These machines represent individual entities that establish communication with the *MQTT* Broker.

In the system, each machine possess the ability to subscribe for specific communication channels known as *MQTT* topics. Once the API publishes data to these designated *MQTT* topics, the *MQTT* Broker takes charge of efficiently directing these messages to their corresponding machines. These machines then accept and process the incoming messages. Furthermore, these machines are equipped to initiate communication by sending responses or updates to the API through HTTP requests.

3.9.2 Database

The initial database was developed thinking of only two entities (tables). One table designated to the machines that will perform the executions, and another table that would store all the tasks to be executed and all the historic of executed tasks.

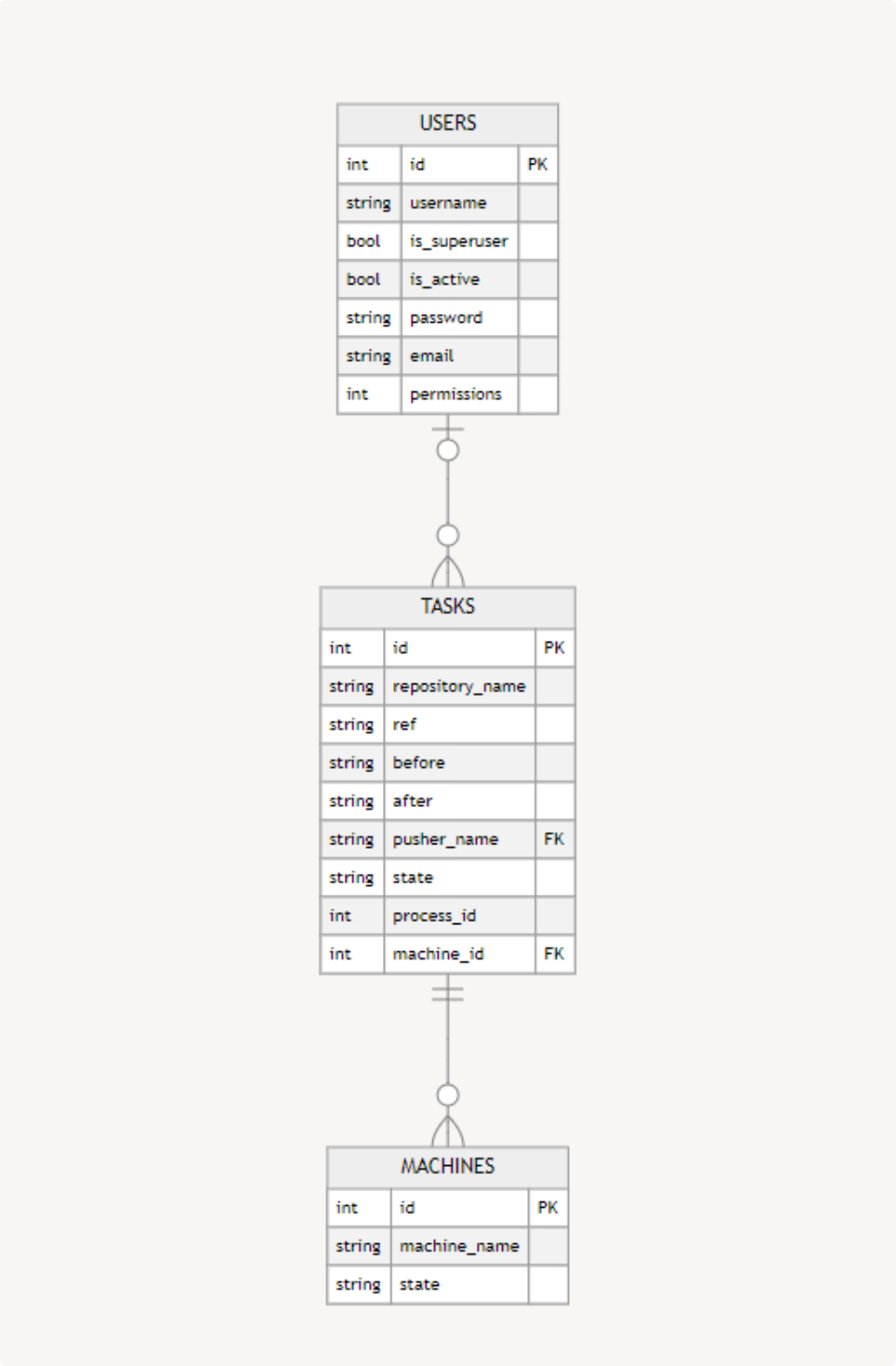


Figure 18: External API Database Structure.

Tasks Entity

Each task is intricately described by a spectrum of essential attributes. These attributes, in Figure 18, coalesce to form a comprehensive representation of the task within the system. Among these attributes, the primary key stands out as a pivotal component, functioning as

the task's unique identifier. This identifier streamlines tracking and management of tasks. Additionally, the task encapsulates a multitude of pertinent details extracted from the GitHub Webhook. These encompass the *ref*, representing the branch linked to the commit; the *before* and *after* commit IDs, denoting the commit's states prior to and after the changes; and the *pusher_name*, identifying the user behind the commit. The *state* attribute manifests as an enumeration encompassing states like *QUEUE*, *PENDING*, *EXECUTING*, *PASSED*, *FAILED*, and *KILLED*. Furthermore, a *process_id* attribute finds its place, responsible for storing the corresponding process ID on the executing machine. This facilitates the potential termination of the task's execution if necessary. Lastly, a *machine_id* attribute is integrated, serving as a pivotal link to the specific machine responsible for executing the task.

Machines Entity

Every machine within the system is defined by a set of distinctive attributes, shown in Figure 18. These include a private key, which serves as its unique identifier; a machine name, intricately linked to the machine's domain and enabling seamless association and communication; and a state designation. The state property is represented by an enumeration encompassing various statuses: *AVAILABLE*, denoting readiness for operation; *PENDING*, indicating the machine is awaiting activation; *RUNNING*, signifying active execution; and *MAINTENANCE*, representing a state of temporary suspension for maintenance purposes.

Users Entity

As the complexity of the external API expanded, a necessary emerged for the creation of a third database entity - named *Users* - geared towards managing access to the external API, as shown in Figure 18. Within this entity, a collection of essential attributes was encapsulated to regulate user interaction. These attributes include the primary *id*, serving as a unique identifier; a boolean property *is_superuser* , distinguishing whether the user holds administrative privileges within the application; another boolean property *is_active* , indicating the current activity status of the user within the system; an associated *name* linked to the user; a *password* serving as the account credential; an *email_address* facilitating potential communication; and lastly, *permissions*.

Bitmap Permissions

Bitmap permissions, often referred to as bitmask permissions, form a robust technique used in computer systems for managing access rights and permissions. This approach involves using binary digits, usually presented as binary numbers or sequence of bits, to represent and control different levels of access to resources.

- **Assignment of Permissions**: In bitmap permissions, different permissions or access rights, such as reading, writing, and executing, are assigned specific positions within the binary sequence. For example, if there are three permissions - read, write, and execute - they might be assigned binary values like read = 001, write = 010, and execute = 100.
- **Integration of Permissions**: When granting or denying permissions to users, the appropriate bits in the binary sequence are combined. To grant both read and write permissions, the bits would be set to 011.
- **Permission Verification**: When a user or process requests access to a resource, the system examines the permission bitmap associated with that resource. Access is granted only if the corresponding bits for the requested permissions are set; otherwise, access is denied.
- **Modification of Permissions**: Modifying permissions involves changing the relevant bits in the permission bitmap. Adding or removing a permission requires adjusting the bit values accordingly.
- **Storage of Permissions**: The permission information is often stored alongside the resource, such as in the metadata of a file or in a database entry. Each resource has its own permission bitmap.

The bitmap structure was organised into groups of four bits each, with each group being associated with a distinct *module_name*. For instance, in a bitmap labeled as *00000000*, the initial set of four entries *0000*, corresponds to GCC module, while the subsequent four entries *0000*, pertain to the QEMU module.

Within each of these four-bit sections, individual bits hold significance in representing specific permissions. The foremost bit, denoted as *0* , signifies the *GET* permission, followed by the second bit, also marked as *0* , which designates the *START* permission. The third bit, indicated by *0* corresponds to the *RESTART* permission, while the fourth and final bit, marked as *0* as well, stands for the *KILL* permission.

MODULES	**GCC**	**GCC**	**GCC**	**GCC**	**QEMU**	**QEMU**	**QEMU**	**QEMU**
PERMISSION	GET	START	RESTART	KILL	GET	START	RESTART	KILL
BINARY PERMISSION	1	1	1	1	1	0	0	0

Table 17: Example for the Bitmap Permissions for GCC and QEMU modules.

As shown in Table 17, to assign all permissions to the GCC module for a user and specifically enable only the GET permission for the QEMU module, the bitmap configuration is *1111 1000*. The initial four bits are allocated to GCC, signifying full permissions, while the subsequent four bits relate to QEMU, indicating the activation of solely the *GET* permission.

Password

Each user registered in the system must have a associated password. However, storing passwords directly in the database is inadequate from a security standpoint. Therefore, the implementation of a password encryption method becomes imperative. This technique involves transforming passwords into irreversible representations using cryptographic hash functions and salting, thus fortifying the security of password storage.

Cryptographic hash functions are one-way mathematical functions that transform input data (in this case, passwords) into fixed-length sequences of characters, known as hash value or digests. A key feature of cryptographic hash functions is that they are irreversible - is not possible to convert the hash back into the original input.

Furthermore, this solution employs the concept of token-based authentication. Tokens, generated upon successful login, encapsulate user-related information, and carry a limited validity period. These tokens serve as temporary access credentials, granting users access to specific resources for a predetermined duration.

The token encoding process involves creating a token containing essential user attributes, coupled with a timestamp indicating its creation time. Token decoding, on the other hand, is the process of extracting user-related data from the token to establish their identity.

To ensure secure access to protected resources, an authentication wrapper has been implemented. This wrapper validates the tokens presented by users, controlling entry to sensitive areas of the application. It acts as a gatekeeper, permitting access only to authenticated users with valid tokens.

3.10 Webhook Workflow

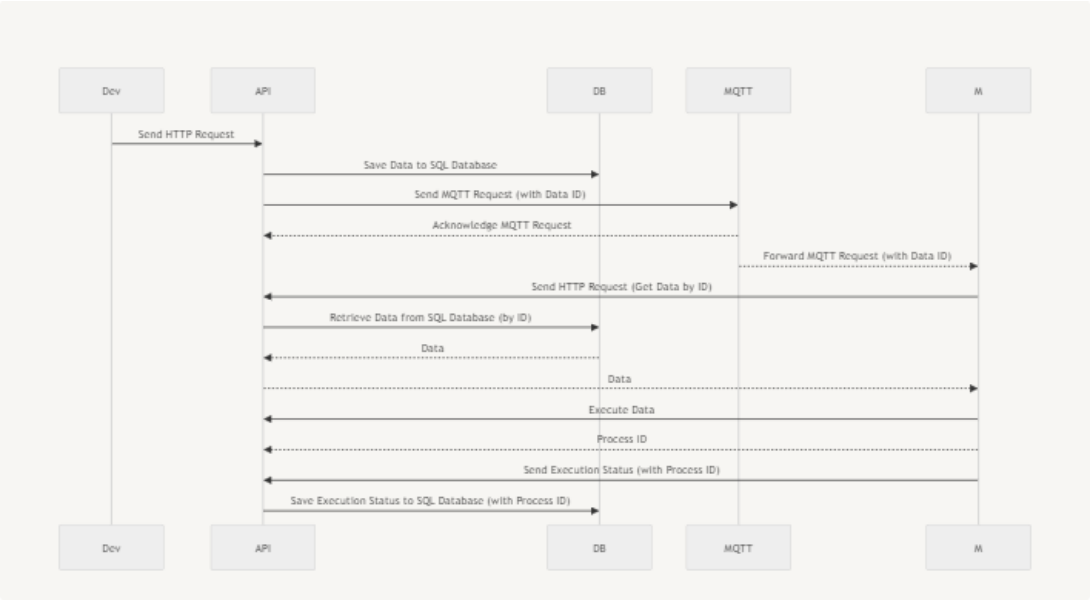


Figure 19: Webhook Sequence Diagram.

As evidenced by the sequence diagram in Figure 19, When a developer successfully generates a new commit for a designated tool GitHub initiates a Webhook transmission to the external API via an HTTP request, incorporating a payload. This payload encapsulates details pertaining to the *repository*, *developer*, *commit IDs* , among others.

For simplicity, the focus will be on a select subset of attributes:

```
{
    "ref": "refs/heads/arc64",
    "before": "43a39d87e26a37a44b9789f6fd2f10341dc99718",
    "after": "bfadc1e1ca2e452be1fd115a1f4100e931883632",
    "repository": {
        "name": "gcc",
    },
    "pusher": {
        "name": "luismgsilva"
    },
}
```

Listing 3.12: GitHub Webhook Payload Excerpt

The payload in the Listing 3.12 contains information about a recent push event within a GitHub repository. This information is essential for understanding the changes made and the con-

text of the push. The payload includes the following properties:

- **ref**: The *ref* property indicates the Git reference associated with the push event. It specifies the branch or reference that was updated or pushed. This reference helps to identify the location of the changes within the repository.
- **before**: The *before* property represents the commit ID (SHA-1) before the push event. This commit ID points to the latest commit on the branch before the push occurred. It serves as a market to denote the state of the branch prior to the changes.
- **after**: The after property signifies the commit ID (SHA-1) after the push event: This commit ID points to the latest commit on the branch after the push. it reflects the new state of the branch, including the changes introduced by the push.
- **repository**: The repository property encapsulates information about the repository involved in the push event. It includes details such as the repository's name. This information helps in identifying the specific repository where the changes were pushed.
- **pusher**: The pusher property provides insights into the user who performed the push event. It includes the *username* of the individual responsible for pushing the changes. This attribution helps in acknowledging the contributor behind the push.

Upon receiving the GitHub Webhook payload, the system promptly extracts the repository's name and the pusher's name, comparing against the users' permissions. The pusher's name denotes the *username* within the system (with a prerequisite alignment between the GitHub username and the system's username), while the repository name conveys the module name within the permissions structure.

As mentioned earlier, each user entry in the database encompasses a permissions bitmap corresponding to distinct module names. In this context, the system undertakes the critical task of validating the user's permissions *vis-à-vis* the specific module name, in this instance being GCC, and more precisely, against the relevant bit corresponding to the *START* permission.

If the user does not have the necessary permissions to start a new task related to GCC, the system will generate an error code in response. Consequently, the present task remains unregistered for execution. Conversely, if the user indeed possesses the authorised permission to initiate task execution, the system proceeds to establish a task entry within the database. The pertinent task-related information is stored, alongside designating the task's state as *QUEUE*. This ensures that if the system is currently in the process of executing tasks, the newly stored task will not overlap with ongoing executions. Instead, it will be added to a queue, effectively maintaining an ordered list of tasks awaiting execution.

Subsequently, a mechanism is triggered to assess the availability of a machine within the infrastructure. This mechanism employs the *First In, First Out* (*FIFO*) methodology, as outlined by Kruse (1994). This principle involves retrieving tasks based on the order of their initiation, giving priority to the earliest arriving task. Simultaneously, the system ensures an available machine is at hand. The process transpires exclusively under the circumstance that there exists a task designed as *QUEUE* and a machine denoted as *AVAILABLE*.

Upton this intersection of conditions, both the task's status and the machine's status undergo an alteration to *PENDING*. This transitional stage symbolises the ongoing validation process to confirm the machine's readiness for execution.

To confirm this, an MQTT request is dispensed to the corresponding machine, soliciting acknowledgment of its availability. Upton reception of the MQTT request, the machine, in turn, initiates an HTTP request to the external API. This request seeks comprehensive task information associated with the provided *task_id* present in the MQTT communication.

The external API, responsible for managing this HTTP request, directs a sequence of actions. It modifies the machine's status to *RUNNING* and concurrently elevates the task's status to *EXECUTING*. Subsequently, the external API furnishes the machine with essential task-related data, culminating in a seamless exchange of requisites for the task's execution process. Upon receiving the payload from the external API, the machines initiate a script tasked with overseeing the building and testing of the tool. Within this script, various parameters are passed, including the task's designation as GCC, its database-assigned *task_id*, the correlated branch

specified as *ref*, and the corresponding *before* and *after* commit IDs.

As soon as the script is initialised, its process id is retrieved and dispatched to the external API, where it is stored along with the associated task's information in the database. This step facilitates the termination of the task's execution, if required. This scenario presents a challenge as a conventional process termination using the bash *kill* command is insufficient due to potential creation of multiple sub processes by the script. To handle this complexity, concluding the execution of a task requires terminating all its subprocesses.

Terminate Child Processes

The underlying logic of eliminating subprocesses revolves around the effective management of a hierarchical process structure. It accomplishes this by pinpointing and subsequently halting a group of processes rooted at a specified parent process. The approach is as follows:

- It firstly executes a recursive exploration to identify all descendant processes linked to a given parent process. This systematic traversal collects process ids at every level, culminating in a comprehensive list of descendant process ids. Once this traversal concludes, the function output this list.
- A designated parent process is earmarked as the point of origin. Leveraging the previously mentioned function, the script searches into all descendants originating from this parent process. Following this assessment, a strategic operational command comes into play, orchestrating the termination of all processes specified in the descendant list.

The cumulative effect of this sequence guarantees the systematic shutdown of not only the parent process, but also all child processes and their descendants. This approach allows for efficient and controlled handling of complex process hierarchies, ensuring comprehensive termination when necessary.

3.10.1 Task Execution Middleware

The task execution middleware bridges the external API infrastructure and BSF. This middleware is responsible for setting up the environment to start the testing. Considering the presented case study, it compiles the Toolchain and start the GCC test suite with the newly built Toolchain.

A important consideration revolves around the version of the tool, specifically GCC in this instance. It's essential to note that not only the GCC repository is required for building the Toolchain but also a collection of repositories, for example Binutils, Newlib, GlibC, among others. All tools, excluding GCC, are expected to utilise the default branch. However, GCC must be configured accordingly to the branch updated.

The chosen approach handles the different tool versions, utilising Git's Worktree capability. The process unfolds in the following manner:

- The initial step is to assess the synchronisation between the default the specified branch. If a misalignment is detected, a separate Worktree is created for the particular branch. This involves actions such as fetching remote updates, generating a dedicated workspace focused on the specified branch.
- With this information in hand, a comparison transpires between the latest commit ID in the individual Worktree and the external target commit ID. A mismatch indicates the absence of the desired version within the Worktree. As a corrective measure, the Worktree is shifted to the target commit.

Upon configuration the tool's version, the subsequent crucial stride involves the construction of all accessible Toolchains. Subsequently, the recently compiled Toolchains are incorporated into the ongoing session by using the freshly generated Modulefiles. With these preparations complete, the testing phase commences. Upon its culmination, a meticulous comparison between the test results and a predefined baseline ensues, meticulously presented in HTML format.

As the culmination approaches, an email is composed, embedding the recently generated comparative insights in HTML. This email is dispatched to the user who initiated the continu-

ous integration workflow. Through this correspondence, the user is apprised of the nuanced alterations that their contributions have instigated within the realms of the tools.

Chapter 4

Conclusions and future work

In conclusion, the successful implementation of automation in a CI environment hinges on the establishment and widespread adoption of a standardized framework within the community. The paramount importance of maintaining consistent test configurations and environments cannot be overstated, as variations can compromise the validity of results. The existing challenge lies in the lack of universal implementation of these standards, leading to discrepancies between locally and remotely executed tests. To bridge this gap, it is imperative to establish and enforce clear guidelines and procedures for test execution across all development stages.

By creating an organised structure that simplifies the execution, comparison, profiling, exploration, and reporting of tests, the project's goal is to provide support and enhance the capabilities of toolchain developers at Synopsys. This approach takes into consideration the distinctive needs and preferences of Synopsys, ensuring practicality and effectiveness. The primary objective is to design a user-friendly tool that simplifies the complexities of deploying testing and reporting across various tools in the ARC Toolchain. This, in turn, allows developers to confidently and consistently apply the methodology.

The project proves beneficial for developers by enabling functionality in a CI environment. Toolchain developers actively contribute by introducing new recipes, while configurations ensure isolated execution environments. The framework effectively manages result persistency, with configurations playing a crucial role in defining how to compare results. The overarching intention is to provide a comprehensive solution that not only meets the unique requirements of Synopsys but also enhances the usability and efficiency of the ARC toolchain for developers.

The creation and implementation of functional tests have played a crucial role in ensuring the robustness and stability of the framework. These tests have been instrumental in uncovering and addressing issues that directly impact the overall functionality of the system. In conclusion, this project contributes significantly to the advancement of automation in the CI environment, providing a standardized framework that enhances the capabilities of the ARC Toolchain for the benefit of Synopsys developers.

4.1 Prospect for future work

The implementation of the Internal BSF API is currently in progress for future integration. This will facilitate a command-line interface and an API interface, making it possible to execute all available command-line features through an HTTP request.

The Internal API interface introduces a layer of interoperability that extends the reach of BSF across different platforms and technologies. With the ability to trigger actions through HTTP requests, developers can integrate BSF functionality into their applications, creating a more cohesive and interconnected system. The prospect of enabling different BSF instances to control one another introduces a new level of scalability and coordination.

In essence, the ongoing development of the Internal BSF API not only expands the ways in which developers interact with the system but also sets the stage for a more interconnected and adaptable infrastructure.

Bibliography

Bazaar, 2005. URL https://www.gnu.org/software/bazaar/. Accessed on 2023-02-19.

Jean-Marcel Belmont. *Hands-On Continuous Integration and Delivery*. Packt Publishing, 2018.

BuildBot, 2003. URL https://buildbot.net/. Accessed on 2023-02-19.

CircleCi, 2011. URL https://circleci.com/. Accessed on 2023-02-19.

Inc Free Software Foundation. Dejagnu: The gnu testing framework, -. URL http://gnu.ist.utl.pt/software/dejagnu/manual/x199.html. Accessed on 2023-02-13.

Inc Free Software Foundation. What is dejagnu?, 1993. URL https://www.math.utah.edu/docs/info/dejagnu_1.html. Accessed on 2023-01-13.

Inc Free Software Foundation. Dejagnu: The gnu testing framework - running tests, 2006. URL http://gnu.ist.utl.pt/software/dejagnu/manual/x397.html. Accessed on 2023-02-13.

John L. Furlani. *Modules: Providing a Flexible User Environment*. 1991.

Git, 2005. URL https://git-scm.com/. Accessed on 2023-02-17.

GitHub. About webhooks. URL https://docs.github.com/en/webhooks/about-webhooks. Accessed on 2023-02-18.

GitLab. What is continuous integration (ci)? URL https://about.gitlab.com/topics/ci-cd/benefits-continuous-integration/. Accessed on 2023-02-18.

Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. *Usage, costs, and benefits of continuous integration in open-source projects*. iEEE, 2016.

IBM-Rational-ClearCase, 1992. URL https://www.ibm.com/products/devops-code-clearcase. Accessed on 2023-02-19.

IBM-Rational-Synergy, 1990. URL https://www.ibm.com/products/rational-synergy. Accessed on 2023-02-19.

Ionos. Continuous integration: definition, 2020. URL https://www.ionos.com/digitalguide/websites/web-development/continuous-integration/. Accessed on 2023-01-18.

Jenkins. Jenkins, 2011. URL https://www.jenkins.io/. Accessed on 2023-02-19.

Robert L. Kruse. *Data Structures and Program Design*. Prentice Hall, 1994.

Clive Longbottom. The pros and cons of ci/cd pipelines, 2021. URL https://www.techtarget.com/searchsoftwarequality/tip/The-pros-and-cons-of-CI-CD-pipelines. Accessed on 2023-01-18.

Mercurial, 2005. URL https://www.mercurial-scm.org/. Accessed on 2023-02-24.

OASIS, 2019. URL https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. Accessed on 2023-04-11.

Perforce, 1995. URL https://www.perforce.com/solutions/version-control. Accessed on 2023-02-24.

Leandro Pinto, Saurah Sinha, and Alessandro Orso. *Understanding myths and realities of test-suite evolution*. SIGSOFT FSE, 2012.

Dr. Jens Saak and Dipl.-Math. Martin Köhler. *Scientific Computing I*. 2021.

Serengetitech. Introduction to git and types of version control systems, 2020. URL https://serengetitech.com/tech/introduction-to-git-and-types-of-version-control-systems/. Accessed on 2023-01-13.

Jeremiah Small. Continuous integration: Pros, cons, best practices, and why you need to start now, 2018. URL https://www.soliantconsulting.com/blog/continuous-integration-best-practices/. Accessed on 2023-01-18.

Subversion, 2000. URL https://subversion.apache.org/. Accessed on 2023-02-24.

Team-Foundation-Version-Control, 2005. URL https://learn.microsoft.com/en-us/azure/devops/repos/tfvc/?view=azure-devops. Accessed on 2023-02-24.

TravisCI, 2011. URL https://www.travis-ci.com/. Accessed on 2023-02-19.

Part III
Appendices

Appendix A
BSF Use Cases

This appendix is dedicated to presenting all Use Cases for the features developed within the framework. It encompasses all Use Cases that were not featured in the core of the dissertation.

A.1 Clone Use Case

The *clone* feature offers a valuable solution for both distributing the framework environment and sharing executed results with developers. It substitutes the *init* feature by initialising the framework via a Git repository instead of supplying a configuration folder.

Use Case	Use Case 05
Name	Initialisation.
Objective	The Actor initialises the framework using a Git repository.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The developer needs to have a Git repository with a framework environment set up.
Post-conditions in case of Success	The framework environment is initialised successfully.
Post-conditions in case of Failure	The framework environment is not initialised and the Actor is alerted with an error message.
Main Scenario	The Actor initialises the framework with a Git repository by cloning it to a specified directory.

Table 18: Use Case representing the initialisation of BSF by repository.

A.2 Status Use Case

The *status* feature offers a glimpse into the execution status of tasks.

Use Case	Use Case 06
Name	Status.
Objective	The Actor displays the status of all tasks.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The status of each task is displayed to the Actor.
Post-conditions in case of Success	The framework environment is initialised successfully.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor displays the status of each task's execution.

Table 19: Use Case representing the display of the status of each task.

A.3 Tasks Use Case

The *tasks* feature exhibits the names and descriptions of each task defined in the configuration file.

Use Case	Use Case 07
Name	Tasks
Objective	The Actor displays the name and description of each task.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The description is needed to be defined in the configuration file of each task.
Post-conditions in case of Success	The name and description of each task is displayed to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor displays the name and description of each task.

Table 20: Use Case representing the display of the name and description of each task.

A.4 Set Use Case

The *set* features allows developers to provide custom value of each input variable to be used in tasks' execution.

Use Case	Use Case 08
Name	Set.
Objective	The Actor defines the User-Defined Property Value.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The User-Defined Property Value needs to be referenced in the configuration file.
Post-conditions in case of Success	The User-Defined Property Value is defined with the value provided by the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor defines an User-Defined Property Value.

Table 21: Use Case representing the set of an User-Defined Property Value.

A.5 Vars Use Case

The *vars* feature exhibits the of all User-Defined Property and signifies whether each is defined or undefined.

Use Case	Use Case 09
Name	Vars.
Objective	The Actor displays all User-Defined Property Values definition.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The User-Defined Property Values need to be referenced in the configuration file.
Post-conditions in case of Success	The User-Defined Property Values is defined with the value provided my the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor displays all the User-Defined Property Values definition.

Table 22: Use Case representing the display of all User-Defined Property Values.

A.6 Sources Get Use Case

The *sources get* feature is responsible for cloning the repositories defined in the configuration file via Git.

Use Case	Use Case 09
Name	Sources Get.
Objective	Clone all associated repositories in the configuration file
Initialisation	Developer
Actors and Stakeholders	Developer
Frequency of Use	Frequently
Pre-conditions	The repositories must be defined in the configuration file.
Post-conditions in case of Success	The repositories are cloned into the framework's environment.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor clones the associated repositories in the configuration file.

Table 23: Use Case representing the clone feature of repositories.

A.7 Sources Delete Use Case

The *sources delete* feature is tasked with removing the cloned repository from the framework's environment.

Use Case	Use Case 10
Name	Sources Delete.
Objective	Delete a source from the framework's environment.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The repository must be presented in the framework's environment.
Post-conditions in case of Success	The repository is deleted from the framework's environment.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor deletes a repository from the framework's environment.

Table 24: Use Case representing the delete feature of repositories.

A.8 Sources List Use Case

The *sources list* is responsible for generating a list of all the repositories defined in the configuration file.

Use Case	Use Case 11
Name	Sources List.
Objective	Lists all sources defined in the configuration file.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The repositories must be defined in the configuration file.
Post-conditions in case of Success	The repositories defined in the configuration file are displayed to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor lists all defined repositories in the configuration file.

Table 25: Use Case representing the list feature of repositories.

A.9 Sources Show Use Case

The *sources show* feature is responsible for displaying all the current cloned repositories in the framework's environment.

Use Case	Use Case 13
Name	Sources Show.
Objective	Shows all sources in the framework's environment.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The repository must be presented in the framework's environment.
Post-conditions in case of Success	The repositories are shown to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor shows all repositories present in the framework's environment.

Table 26: Use Case representing the show feature of repositories.

A.10 Git Use Case

The *git* feature provides Git commands that can be used to interact with the Git version control system within the framework environment. These commands allows to manage the executed task iterations.

Use Case	Use Case 14
Name	Git.
Objective	Executes Git commands in the framework's internal repository.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The framework's internal repository must be initialised.
Post-conditions in case of Success	The Git commands are executed internally by the framework.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor executes Git command in the framework's internal repository. (e.i., bsf git log)

Table 27: Use Case representing the Git feature of the framework's internal repository.

A.11 Search Use Case

The *search* feature provides developers a way to search for a specific executed iteration.

Use Case	Use Case 15
Name	Search
Objective	Search a specific task execution iteration.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The framework's internal repository must be initialised.
Post-conditions in case of Success	The Actor locates what they were searching for.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor searches the commit messages of the framework's internal repository accordingly to the provided data.

Table 28: Use Case representing the search feature of the framework.

A.12 Diff Use Case

The *diff* feature provides a difference between the commit messages of two given entries in the internal repository of the framework's environment.

Use Case	Use Case 15
Name	Diff.
Objective	Displays the differences between two commit messages of the framework's internal repository.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	The commit IDs must be present in the framework's internal repository.
Post-conditions in case of Success	The Actor retrieves the differences between two commit messages.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The framework presents the discrepancies between two commit messages from its internal repository.

Table 29: Use Case representing the diff feature of the framework.

A.13 Ls Use Case

The *ls* feature offers developers a glimpse into the contents of the persistent workspace associated with a specific task.

Use Case	Use Case 17
Name	Ls
Objective	Display the stored files of an associated task in the framework's internal repository or locally.
Initialisation	Developer.
Actors and Stakeholders	Developer
Frequency of Use	Frequently
Pre-conditions	Existing of an entry in the framework's internal repository
Post-conditions in case of Success	The framework provides a list of the stored files to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor retrieves the list of stored files of an associated task in the framework's internal repository or locally.

Table 30: Use Case representing the ls feature of the framework.

A.14 Cat Use Case

The *cat* feature allows developers to peer into the contents of specific files located within the persistent workspace of a designated task. This feature operates as a textual window, providing a direct view into the data stored within the files.

Use Case	Use Case 18
Name	Cat.
Objective	Displays the stored file of an associated task in the framework's internal repository or locally.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	Existing of an entry in the framework's internal repository.
Post-conditions in case of Success	The framework provides the contents of a file.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor retrieves the stored file content of an associated task in the framework's internal repository or locally.

Table 31: Use Case representing the cat feature of the framework's internal repository.

A.15 Cp Use Case

Use Case	Use Case 19
Name	Cp.
Objective	Copies a file from the persistent workspace.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	Existing of an entry in the framework's internal repository.
Post-conditions in case of Success	The framework copies a file.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor copies a stored file of an associated task in the framework's internal repository or locally.

Table 32: Use Case representing the cp feature of the framework.

A.16 Compare Use Case

The *compare* feature allows developers to compare past executed iterations.

Use Case	Use Case 20
Name	Compare.
Objective	Compare two or multiple past iterations of executed results of a given task.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	Associated task's comparison property, comparator, must be defined in the configuration file.
Post-conditions in case of Success	The framework provides a comparison report of a given task to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor compares two or multiple past iterations of executed results of a given task using the provided method in the configuration file comparator property.

Table 33: Use Case representing the compare feature of the framework.

A.17 Compareall Use Case

The *compareall* feature allows developers to generate an unique report of all the past executed iterations of all available tasks.

Use Case	Use Case 21
Name	Compare All.
Objective	Compare two or multiple past iterations of executed results of all tasks.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	comparator_aggregator property must be defined in the configuration file.
Post-conditions in case of Success	The framework provides a comparison report of all tasks to the Actor.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor compares two or multiple past iterations of executed results of all tasks using the provided method in the configuration file comparator_aggregator property.

Table 34: Use Case representing the compareall feature of the framework.

A.18 Report Use Case

The *report* feature allows developers the opportunity to have a special property in the configuration file dedicated to a report.

Use Case	Use Case 22
Name	Report.
Objective	Generate a report of a given task.
Initialisation	Developer.
Actors and Stakeholders	Developer.
Frequency of Use	Frequently.
Pre-conditions	Associated task's report property must be defined in the configuration file.
Post-conditions in case of Success	The framework generates a report of a given task.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor generates a report of a given unique task using the provided method in the configuration file report property.

Table 35: Use Case representing the report feature of the framework.

A.19 Save Config Use Case

The *saveconfig* feature provides a valuable solution for securing framework configurations and guaranteeing that developers can restore their desired adjustments, even when con-

fronted with unexpected circumstances that could lead to the loss of the original configuration folder.

Use Case	Use Case 23
Name	Save Config
Objective	Saves the internal configuration to a path.
Initialisation	Developer
Actors and Stakeholders	Developer
Frequency of Use	Frequently
Pre-conditions	The framework must be initialised.
Post-conditions in case of Success	The internal configuration is duplicated to the provided path.
Post-conditions in case of Failure	The Actor is alerted with an error message.
Main Scenario	The Actor saves the internal configuration to a given path.

Table 36: Use Case representing the saveconfig feature of the framework.

Appendix B
BSF Features

This appendix is dedicated to offering details about every BSF Feature, encompassing their respective descriptions, flowcharts, and usage examples. It covers features that were not included in the core of the dissertation.

B.1 Save Config Feature

The *saveconfig* feature offers a valuable solution to safeguard framework configurations, ensuring developers can recover their preferred adjustments even in unforeseen situations that may result in the loss of the original configuration folder. The framework will extract the internal configuration residing within the framework's environment in the existing directory. Subsequently, it will replicate this content and move it to the specified destination.

Flowchart

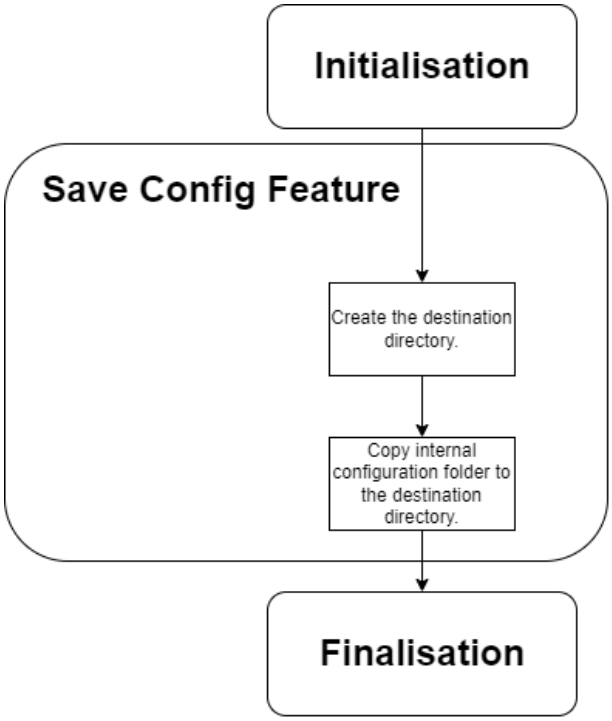


Figure 20: The flowchart outlines the workflow for the *saveconfig* feature.

Following the *Initialisation* step, as shown in Figure 20, if the destination folder doesn't already exist, it is created, and all content from the internal configuration environment is then copied to this newly created destination folder, leading to the *Finalisation* step.

Syntax

```
1     $ <application> saveconfig <path/to/destination_folder>
```

Use Examples

```
1     $ bsf saveconfig /home/user/destination_folder
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.2 Clone Feature

The *clone* feature presents a helpful approach for both distributing the framework environment and sharing executed results with developers. It replaces the *init* feature by initialising the framework through a Git repository rather than providing a configuration folder. The framework will mirror the environment from the specified repository and store it in the designated path.

Flowchart

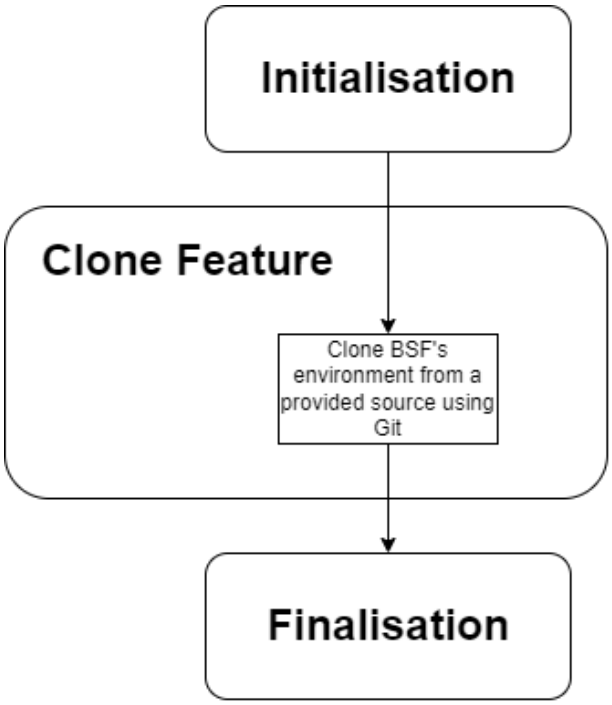


Figure 21: The flowchart outlines the workflow for the *clone* feature.

Subsequent to the *Initialisation* step, as illustrated in the figure 21, the provided repository URL will be used to clone a framework environment, and the cloned content will be stored in a specified directory. Following is the *Finalisation* step.

Syntax

```
1     $ <application> clone <repository_url> <path/to/
          destinaltion_folder>
```

Use Examples

```
1     $ bsf clone https://github.com/user/repository /home/user/
          destination_folder
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.3 Vars Feature

The *vars* feature exhibits a list of all User-Defined Property Values and signifies whether each is undefined or defined with their value.

Flowchart

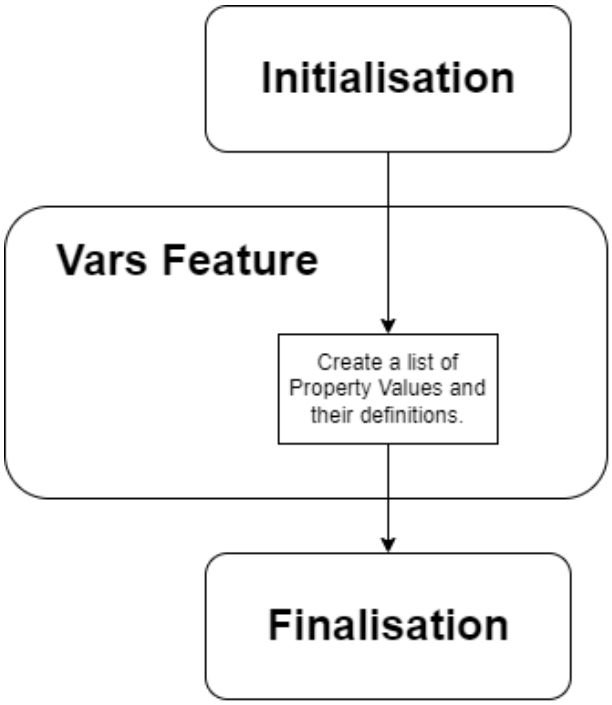


Figure 22: The flowchart outlines the workflow for the *vars* feature.

Following the *Initialisation* step, as presented in Figure 22, the framework proceeds to display a list of all User-Defined Property Values, accompanied by their respective names and values. In situations where they have not been defined yet, a message indicating that they are awaiting definition is presented instead of a value. Finally, the *Finalisation* step.

Syntax

```
1     $ <application> vars
```

Use Examples

```
1     $ bsf vars
2     Input Variable PREFIX defined: /home/user/install
3     Input Variable VERSION undefined.
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.4 Tasks Feature

The *tasks* feature exhibits the names and descriptions of each task defined in the configuration file.

Flowchart

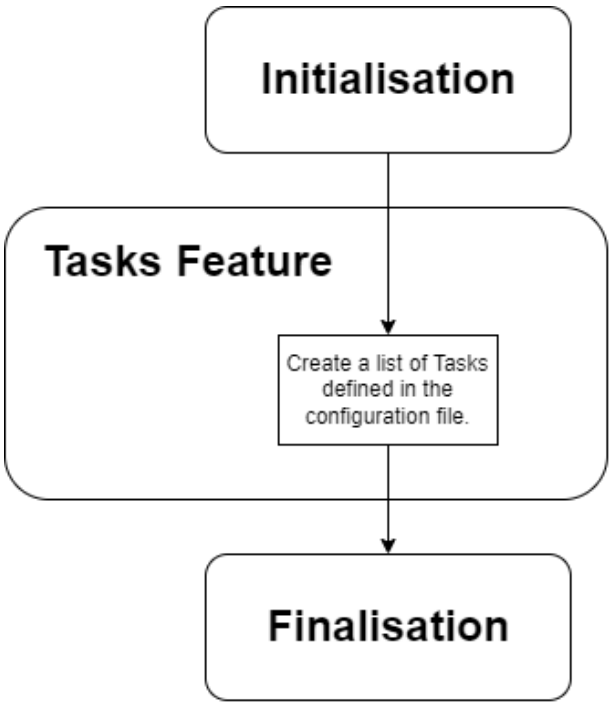


Figure 23: The flowchart outlines the workflow for the *tasks* feature.

Afterwards to *Initialisation* step, as depicted in Figure 23, the framework proceeds to present a list of all tasks, including their names and corresponding descriptions if available in the configuration file. Subsequent to this, the process concludes with the *Finalisation* step.

Syntax

```
1     $ <application> tasks
```

Use Examples

```
1     $ bsf tasks
2     archs-gcc-baremetal-qemu
3                 Description: This task runs the tests for the
                      archs gcc
4                                   baremetal (arc-elf32-gcc) using
                                        qemu (qemu-sysem-arc)
5                                   as emulator.
6     hs6x-gcc-linux-user-qemu:
7         Description: This task runs the tests for the hs6x gcc
8                   linux (arc64-linux-gnu-gcc) using qemu (qemu-
                      arc64)
```

```
9                    as emulator
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.5 Set Feature

The *set* feature allows the chosen User-Defined Property Value to become defined within the framework's environment. Subsequently, when is used in the configuration file, it is substituted with its defined value.

Flowchart

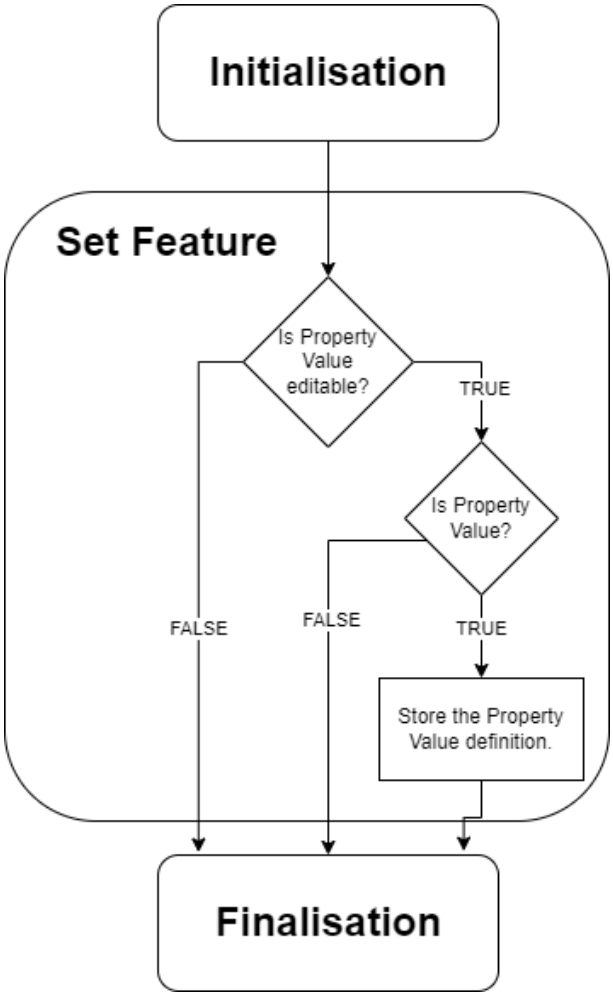


Figure 24: The flowchart outlines the workflow for the *set* feature.

Subsequently to the *Initialisation* step, as shown in Figure 24, the framework proceeds to perform a series of validations. First, it verifies the eligibility of the specified variable by determining whether it is a User-Defined or an Internal Property Value managed by the frame-

work. Following this assessment, it validates the existence of the variable within the configuration file. If the variable is not found, an exception is triggered, leading to the termination of the process. However, if the variable exists, the new value corresponding to that variable is stored. Wrapping up with the *Finalisation* step.

Syntax

```
1     $ <application> set <name>=<value>
```

Use Examples

```
1     $ bsf set PREFIX=/home/user/install
2     $ bsf set PREFIX1=/home/user/install
3     ERROR: PREFIX1 not an input variable
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.6 Diff Feature

The *diff* feature provides developers with a JSON formatted output that highlights the disparities between the two commit message entries in the internal repository of the framework's environment.

Flowchart

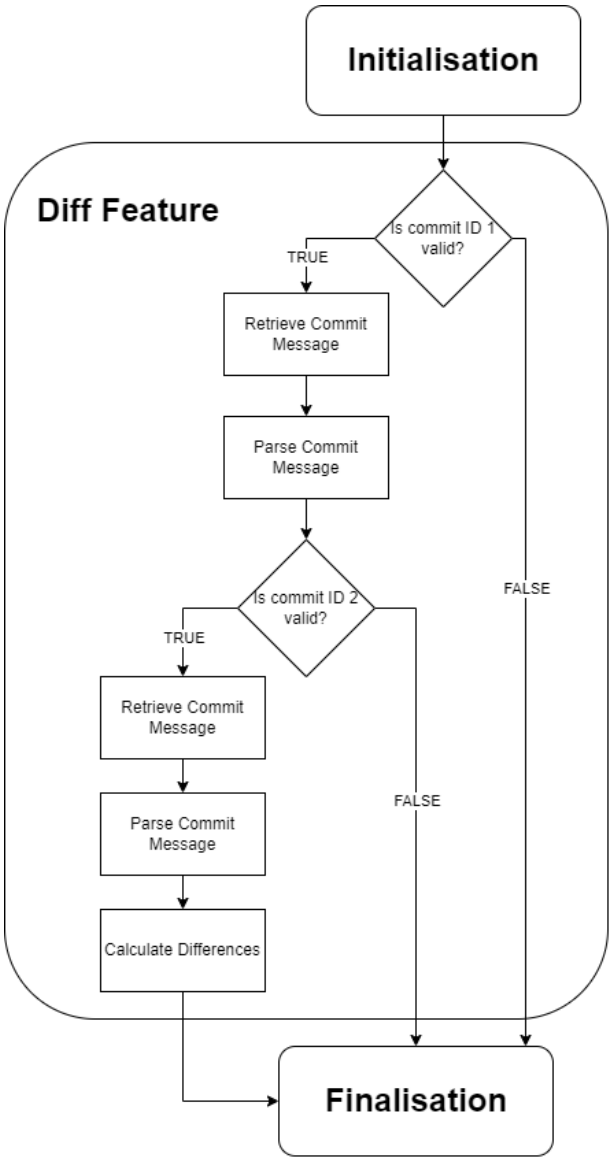


Figure 25: The flowchart outlines the workflow for the *diff* feature.

Post the *Initialisation* step, as presented in Figure 25, there are multiple steps involved in calculating the difference between commits. It checks the validity of two different commit IDs. If the first commit ID is valid, it proceeds to retrieve the commit message associated. Then, the retrieved message goes through a parsing step. After parsing, it checks another commit ID and the process repeats. If the second is valid, it proceeds to retrieve the commit

message and parse it. If any of the commit IDs are found to be invalid, it results in exceptions being raise.

The comparison is performed by examining the key-value pairs of the commit messages. If a difference is found, it is recorded. This process is recursive, allowing for the exploration of nested structures within the commit messages. The result is a representation of the disparities between the two sets of commit messages, presented in a well-organised JSON format. Completing the process with the *Finalisation* step.

Syntax

```
    $ <application> diff <hash1> <hash2>
```

Use Examples

```
    $ bsf diff 53b7585a 3e2377d2
    {
    "archs-gcc-baremetal-qemu": {
        "archs-gcc-baremetal": {
            "build_data": [
                "2024-01-06 15:29:49",
                "2024-01-06 16:58:11"
                ],
            "gcc": {
                "hash": [
                    "c805c9b4319162967b7bbd9040d8bb29aa068676",
                    "9729b848e5e4886f9adf10d2adf9d3358e5bf343"
                ]
            },
            "newlib": {
                "hash": [
                    "6395c43a09680402a27bcd774380aa4118e602f4",
                    "bb526c606cacdefb2d14f48ff31232ea00015ea4"
                ]
            }
        }
    }
    }
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.7 Git Feature

The *git* feature provides Git commands that can be used to interact with the Git version control system within the current framework environment.

Flowchart

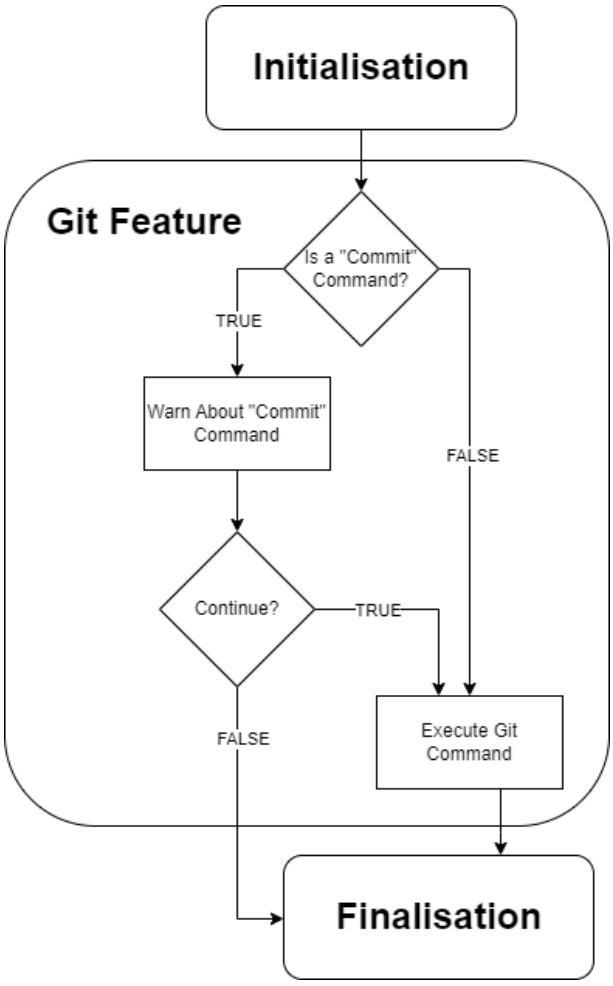


Figure 26: The flowchart outlines the workflow for the *git* feature.

When the *Initialisation* step is finished, according to the Figure 26, the process checks if a commit command is provided. If it finds a commit command, it proceeds to issue a warning about the commit action. Following that, the developer is prompted for confirmation. If the developer chooses to continue, the process executes the Git command within the internal framework's environment repository. If the developer opts to abort, the process moves directly to the *Finalisation* phase. In cases where a command other than a commit is detected, the process also executes the Git command without developer interaction..

Syntax

```
1     $ <application> git <commands>
```

Use Examples

```
1     $ bsf git log
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.8 Search Feature

The *search* feature provides developers with one or more commits if the provided parameters match with any entry in the internal repository of the framework environment.

Flowchart

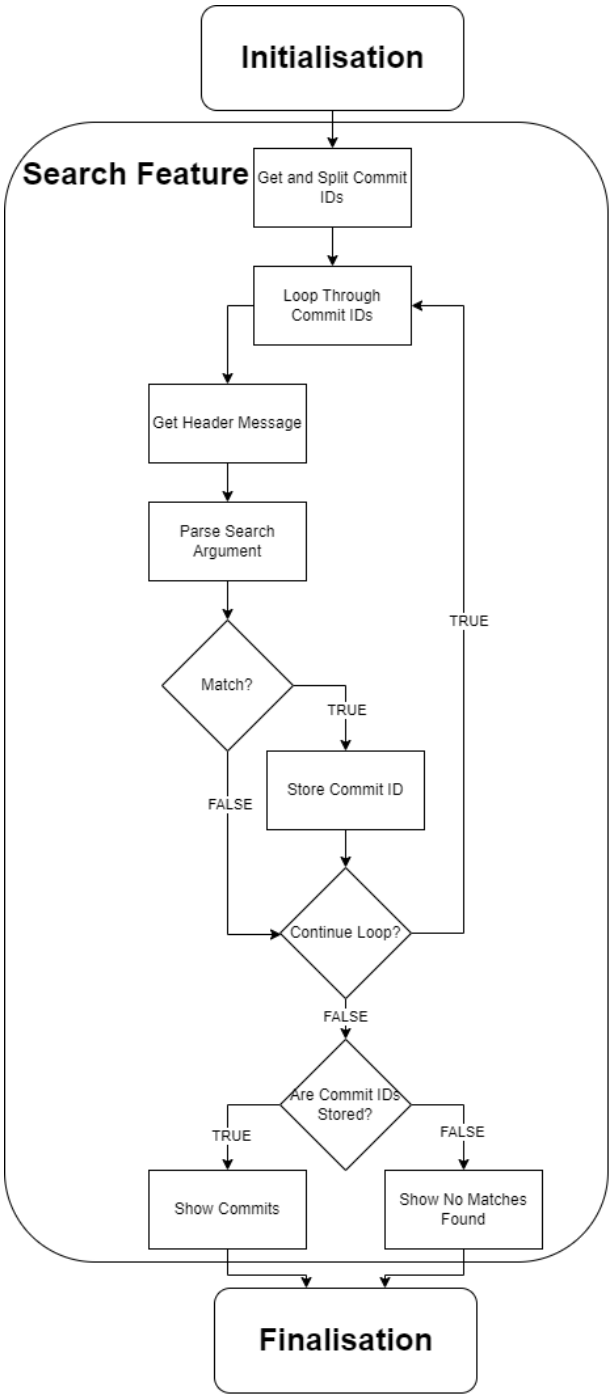


Figure 27: The flowchart outlines the workflow for the *search* feature.

After completing the *Initialisation* stage, as shown in Figure 27, it then proceeds to retrieve commit IDs, split them into individual items, and iterates through each, collecting header messages. These header messages are evaluated based on specific criteria. If a match is found, the commit ID is saved; otherwise, the process continues to the next one.
If additional commit IDs are present, the process returns to the loop. When no more commit IDs remain, it checks if any were saved. If saved commit IDs exist, the process proceeds to display them. However, in the absence of saved commit IDs, it indicates that no matches were found. Finishing off with the *Finalisation* stage.
Syntax

```
1     $ <application> search <var>=<value>
```

Use Examples

```
1     $ bsf search build_date=20221011
```

```
1     $ bsf search build_date=20221011 hash=484f32
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.9 Status Feature

The *status* feature provides developers with a concise overview of the execution status of each configured task. Each task is accompanied by its name and its corresponding state, which falls into one of the three categories: *Passed*, *Failed*, or *Not Executed*.
Flowchart

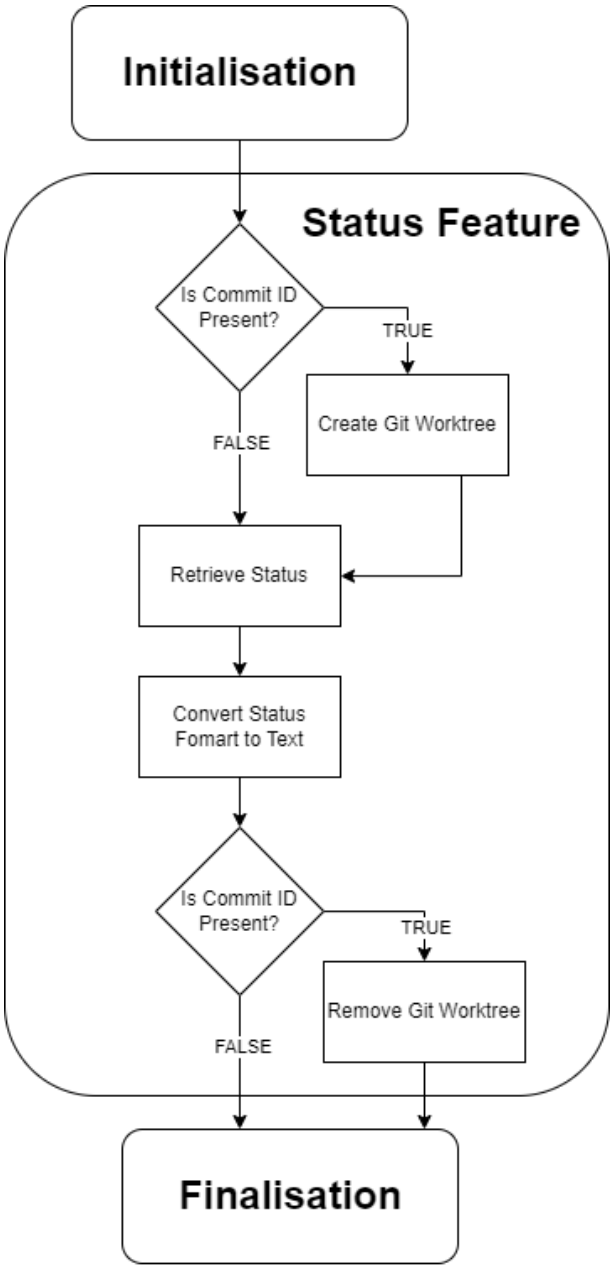


Figure 28: The flowchart outlines the workflow for the *status* feature.

In the aftermath of the *Initialisation* step, as presented in Figure 28, the process first checks for the presence of a commit ID. If a commit ID is detected, it proceeds to create a Git Worktree; otherwise, it directly advances to the subsequent step. Following this, the process retrieves and translates the status information into human-readable text. The formatted status information undergoes further processing, and the process reevaluates the presence of a commit ID. If a commit ID is still present, it proceeds to remove the Git Worktree; otherwise, it directly moves to the *Finalisation* Step.

Syntax

```
1     $ <application> status
```

Use Examples

```
$ bsf status
Passed: task01
Failed: task02
Not Executed: task03
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.10 Ls Feature

The *ls* feature offers developers with a detailed listing of the contents within the persistent workspace associated with the specified task. This listing includes the names of file and directories.

Flowchart

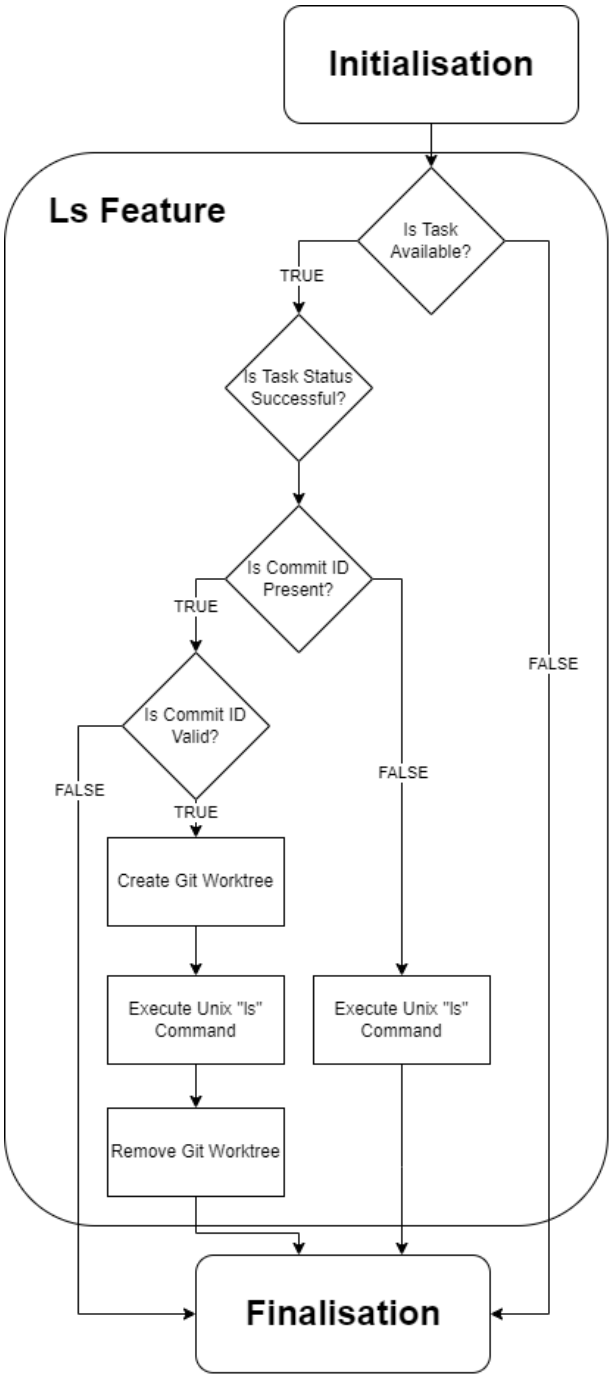


Figure 29: The flowchart outlines the workflow for the *ls* feature.

Following the *Initialisation* stage, as portrayed in Figure 29, it validates the specified task's

existence within the configuration file. If the task is not found, this triggers an exception, leading to the framework's termination. However, if the task is deemed valid, the process proceeds to confirm the presence of a commit ID. In cases where no commit ID is provided, the framework verifies the task's status within the local iteration. If the task's status is anything other than "Passed", an exception is raised, resulting in the framework's termination. Conversely, if the status is "Passed", it executes a *Unix ls* command on the specified folder, generating a list of all files stored in the persistent workspace of the designated task.
In scenarios where a commit ID is present, the framework performs an additional validation to ascertain its authenticity. If the commit ID is determined to be invalid, an exception is raised, leading to the framework's termination. Conversely, if the commit ID is valid, a Git Worktree associated with that specific commit ID is established. Subsequently, the task's status is examined in accordance with the commit ID. If the status fails to meet the "Passed" criteria, an exception is raised, ultimately resulting in the framework's termination. Conversely, if the status is "Passed", the process proceeds to execute the *UNIX ls* command on the specified folder, generating a list of all files stored in the persistent workspace of the assigned task, and subsequently removes the Git Worktree. Bringing to an end, reaching the *Finalisation* stage.

Syntax

```
1     $ <application> ls <task> [commit_id]
```

Use Examples

```
1     $ bsf ls
2     gcc.log
3     gcc.sum
```

```
1     $ bsf ls 3e2377d2
2     gcc.log
3     gcc.sum
4     glibc.sum
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.11 Cat Feature

The *cat* feature allows developers to peer into the contents of specific files located within the persistent workspace of a designated task. This feature operates as a textual window,

providing a direct view into the data stored within the files, maintaining the original format.

Flowchart

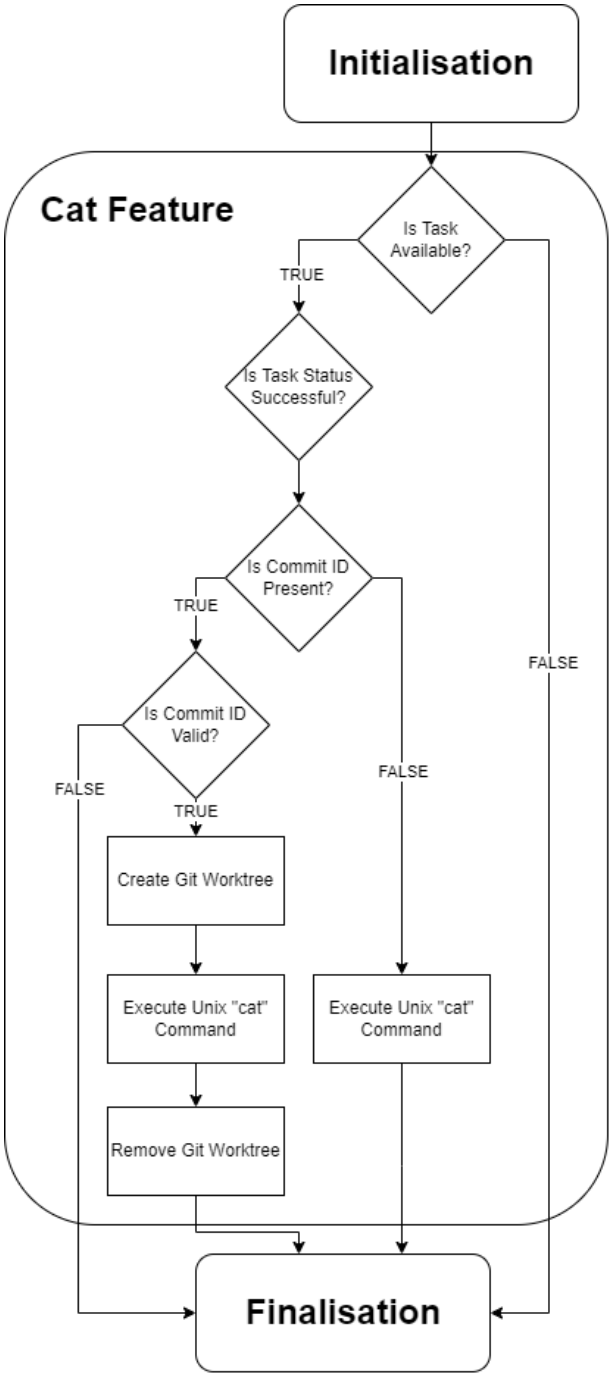


Figure 30: The flowchart outlines the workflow for the *cat* feature.

Subsequent to completing the *Initialisation* step, as evidenced in the Figure 30, it verifies the validity of the specified task within the configuration file. If the task is not found, an exception is raised, resulting in the framework's termination. However, if the task is valid, the process proceeds to validate the presence of a commit ID. If a commit ID is absent, the application

revalidates the status of the task in the local iteration. If the task's status is anything other than "Passed", an exception is raised, leading to the framework's termination. Conversely, if the status is "Passed", it executes the *Unix cat* command on the specified file.
If a commit ID is present, the framework further validates whether it is a valid commit ID. If it is found to be invalid, an exception is raised, resulting in the termination of the framework. However, if the commit ID is valid, a Git Worktree of that commit ID is created, and the task's status is checked accordingly for that specific commit ID. If the status is not "Passed", an exception is raised, leading to the framework's termination. On the other hand, if the status is "Passed", it proceeds to execute the *Unix cat* command on the specified file and removes the Git Worktree. Concluding with the *Finalisation* step.
Syntax

```
1     $ <application> cat <task> [commit_id]
```

Use Examples

```
1     $ bsf cat task01 file01
2     This is the content of file01 version 2.0.
```

```
1     $ bsf cat task01 file01 3e2377d2
2      This is the content of file01 version 1.0.
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.12 Sources Get Feature

The *sources get* feature will clone the repositories defined within the framework's environment configuration file.
Flowchart

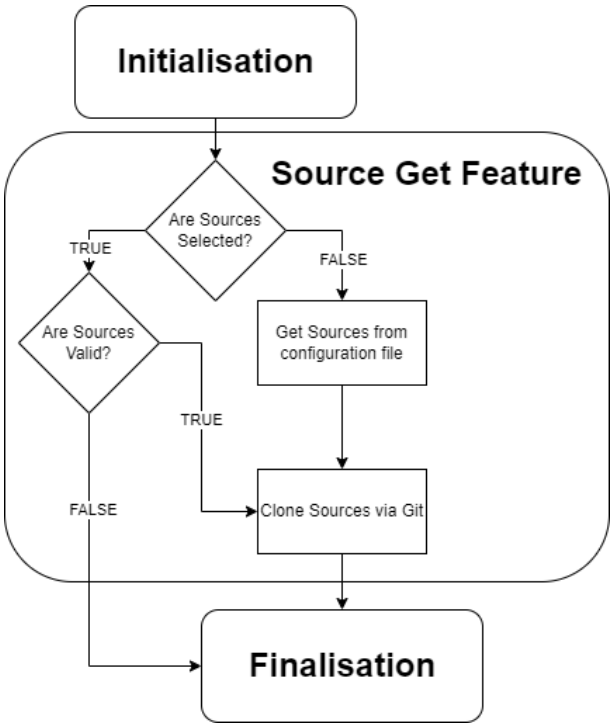


Figure 31: The flowchart outlines the workflow for the *sources get* feature.

After the *Initialisation* stage, as presented in Figure 31, the process checks for the presence of selected sources. If those are present, it proceeds to validate them. If they are invalid, an exception is raised, and the process moves to the *Finalisation* step. If the sources are valid, the process filters the configuration sources.

If no selected sources are initially detected, the process retrieves all the sources from the framework's environment internal configuration. Finally, the process clones the sources, and the entire process concludes with the *Finalisation* step.

Syntax

```
1    $ <application> sources get [sources]
```

Use Examples

```
1    $ bsf sources get
2    Cloning into '/home/user/ws/sources/arc-gnu-toolchain'...
3    remote: Enumerating objects: 1239, done.
4    remote: Counting objects: 100% (1239/1239), done.
5    remote: Compressing objects: 100% (1044/1044), done.
6    remote: Total 1239 (delta 148), reused 1211 (delta 142),
        pack-reused 0
7    Receiving objects: 100% (1239/1239), 45.19 MiB | 10.83 MiB/
        s, done.
8    Resolving deltas: 100% (148/148), done.
9    Updating files: 100% (1182/1182), done.
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.
- **-s** or **–shallow**: Creates a shallow clone of the repository, meaning that only the latest commit history is fetched.

B.13 Sources Delete Feature

The *sources delete* feature deletes a specified source from the respective sources folder in the framework's environment.

Flowchart

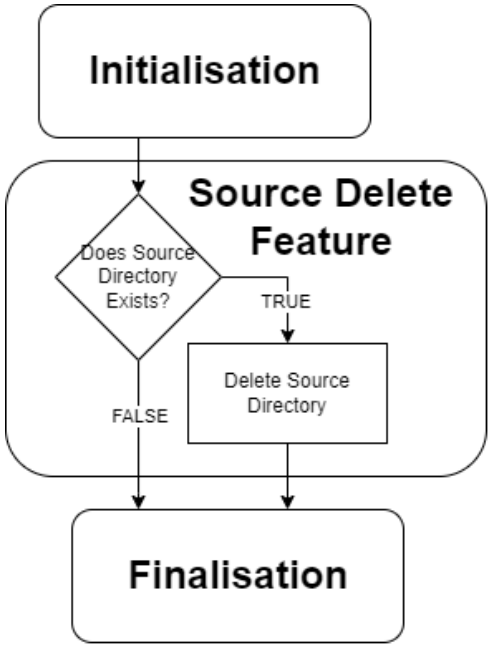


Figure 32: The flowchart outlines the workflow for the *sources delete* feature.

Once the *Initialisation* step is concluded, as shown in Figure 32, the process initially checks for the existence of a source directory. If the source directory is found to exist, it proceeds to delete the directory.

However, if the source directory does not exist, the process raises an exception, indicating that the source has not been registered or is missing. This exception then leads to the *Finalisation* step, concluding the entire process.

Syntax

```
1     $ <application> sources delete <source>
```

Use Examples

```
$ bsf sources delete arc-gnu-toolchain
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.

B.14 Sources List Feature

The *sources list* will display a list of all sources specified in the framework's environment configuration file. This list would include the name or identifier of each source, its associated URL (the repository's clone location), and the branch name (if specified) for that source.

Flowchart

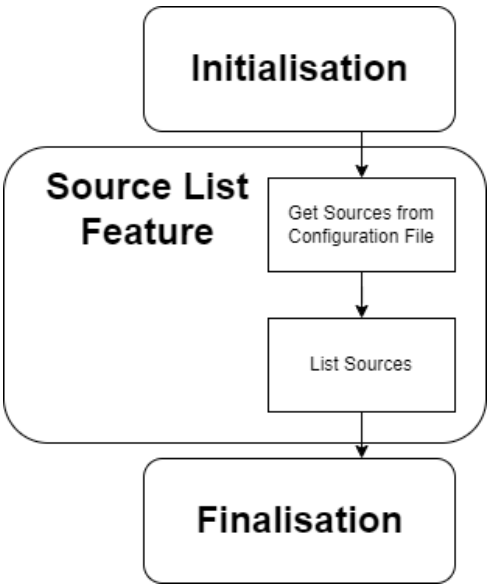


Figure 33: The flowchart outlines the workflow for the *sources list* feature.

After Initialisation, as illustrated in Figure 33, it proceeds to obtain the sources from the framework's environment internal configuration. Subsequently, the process lists the sources. Finally, it concludes by reaching the *Finalisation* step.

Syntax

```
$ <application> sources list
```

Use Examples

```
$ bsf sources list
repository01
    repo: https://github.com/user/repository01.git
repository02
    repo: https://github.com/user/repository02.git
    branch: branch_name
```

Options and Flags

- **–enable-debug-info**: Display all the internal commands executed as BSF is in progress.