Institut für Softwaretechnologie
Abteilung Software Engineering

Universität Stuttgart
Universitätstraße 38
D-70569 Stuttgart

Leveraging Continuous Integration in Space Avionics - A Design using Declarative Build Automation Paradigm

Ganesh Ayalur Ramakrishnan

Course of Study: Master of Science INFOTECH

Examiner: Prof. Dr. rer. nat. Stefan Wagner

Supervisor: Dipl.-Inf. Johannes Lieder (Airbus DS GmbH)

Commenced: October 1, 2016

Completed: March 31, 2017

CR-Classification: D.2.8, D.2.9

Erklärung

Ich versichere, diese Arbeit selbständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references other than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure.
I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

ACKNOWLEDGEMENT

I would first like to thank my thesis advisor **Prof.Dr.rer.nat. Stefan Wagner** of the Institut für Softwaretechnologie at Universität Stuttgart. His support was very valuable for the entire duration of my work. I was given the freedom to express my thoughts, opinions and ideas. His suggestions and teachings were a guiding light for me in the right direction.

I would also like to thank my supervisor **Dip.Inform. Johannes Lieder** of Airbus DS GmbH, Friedrichshafen for his support in this study. This work would not have been possible without his expertise and periodic interactions with him always added value to my work. I would also like to thank the complete On-Board Software Development team at Airbus led by **Dipl.Informatiker(FH) Harald Selegrad** for their inputs, passionate participation and review of my work.

Finally, I must express my very profound gratitude to my parents and brothers for providing me with unfailing support and continuous encouragement throughout my years of study. I would forever be indebted to my friends without whose help and support, this study would not have been possible. Thank You.

Ganesh Ramakrishnan
Friedrichshafen, Germany

Abstract

There are several benefits when Continuous Integration (CI) is adopted for a software development project. This provides for a mechanism to reduce the burden on developers during the build and test of the developed software, as well as help release the product on-time. Other benefits include capturing errors quite early in the development cycle, easier integration at defined intervals over the course of software development, and faster, comprehensive feedback to developers. However, in an embedded domain, adopting CI is a challenging activity. If the project size and complexity is high, there will be a large number of activities which need to be covered in the CI workflow. Not all tools used in software development provide seamless interfaces to the CI tool. There is a need to design the interface framework which can quickly grow to be complex and time consuming.

An effective CI workflow follows a set of best practices. Build automation is one of them. The existing literature does not provide comprehensive information to address the effect that the build automation tools have on the design and implementation of a CI framework in an embedded avionics domain. Tools like GNU Make and Apache Ant are primarily used for the build and test stages of development. However, these build tools are imperative in nature. As the build logic increases in complexity, the conciseness of build scripts reduces. The build run times should also not be large as the feedback cycle time would be longer.

This study aims to design a CI workflow for a space satellite On-Board Software (OBSW) development project. The objective is to bring out the limitations and challenges of using a conventional imperative build approach during the set-up of a CI framework for the project. The proposal is to adopt a build tool which is based on declarative build paradigms and provide for mechanisms to easily integrate with CI tools. This study is carried out as an action research (AR) with study results expressed as quantitative or qualitative metrics. A prototypical CI chain is implemented with a Jenkins CI server and Gradle as the primary build tool. Parameters such as performance, maintenance complexity of build logic, and features such as integration to a CI tool, reproducible builds are investigated.

Contents

1 Introduction **13**
1.1 Motivation . . . 13
1.2 Thesis Overview . . . 15

2 Background **17**
2.1 Continuous Integration . . . 17
2.2 Build Systems . . . 18

3 Research Approach **23**
3.1 Action Research . . . 23
3.2 Qualitative Metrics in Study . . . 24
3.3 Validity Threats . . . 25

4 On-Board Software Development (OBSW) - AS400 Central Software (CSW) **27**
4.1 Overview . . . 27
4.2 Software Development Environment (SDE) . . . 27
4.3 Software Development in OBSW - CSW . . . 29
4.3.1 Terminology . . . 29
4.3.2 AS400 CSW Architecture . . . 30
4.4 Software Development Workflow . . . 30
4.5 Build Automation in AS400 CSW . . . 32

5 Gradle for AS400 CSW **39**
5.1 Terminology . . . 39
5.2 Build System Design Aspects . . . 41
5.3 Analysis of Build Logic . . . 44

6 Continuous Integration in AS400 CSW **47**
6.1 Terminology . . . 47
6.2 CI Workflow for AS400 CSW . . . 48
6.3 Challenges faced during design . . . 52

7 Evaluation and Results **55**
7.1 Performance Comparison . . . 55
7.2 Complexity of Build Logic . . . 61
7.3 Feature Comparison . . . 63
7.4 Summary . . . 66

8 Conclusion **67**
8.1 Summary . . . 67
8.2 Future Work Items . . . 67

List of Figures

1 Block Diagram to show Overview of CI concept . . . 14
2 Expressiveness vs Conciseness - How the different build tools fit into this comparison graph? . . . 19
3 An example of a *build.gradle* . . . 22
4 The output of *gradle tasks* command . . . 22
5 Action Research (AR) cycle consisting of three distinct stages . . . 23
6 SDE set-up for AS400 Central Software Development . . . 29
7 The AS400 Project top level view. . . . 31
8 An excerpt of the AS400 Production Repository Structure . . . 31
9 Representation of the Gitflow Workflow containing master, feature, develop and release branches . . . 32
10 Source Tree of one of the Collections containing Constituents and Sub-Constituents . . . 34
11 Makefile for 'aocs' Collection . . . 35
12 Makefile for 'aocsApFw' Constituent . . . 35
13 Representation of the build methodology from Build Author's View Model 35
14 *build.xml* for AS400 Validation . . . 37
15 *build.gradle* at asw level . . . 40
16 An excerpt of the AS400 Production Repository Structure containing *build.gradle* files . . . 42
17 *build.gradle* at as400prod level . . . 43
18 *build.gradle* at as400prod level(contd.) . . . 44
19 Excerpt of *build.gradle* of a Collection containing Constituents . . . 45
20 Jenkins Master Slave set-up . . . 48
21 Overview of Stash - Jenkins connection . . . 49
22 Overview of the structure of a Job chain containing phases and jobs . . . 51
23 A detailed Jenkins workflow for the project under study . . . 52
24 Performance comparisons between Gradle and Make for clean task, specific target and incremental builds . . . 57
25 Specific builds without custom plugins applied to the project (N) and enabling configuration on demand (C) . . . 58
26 Gradle profiling results for a specific build . . . 58
27 Gradle profiling results for a specific build with the Gradle daemon running 59
28 Gradle profiling results for a specific build enabling --configure-on-demand 59
29 Gradle profiling results for a specific build enabling --configure-on-demand and --parallel execution . . . 60
30 Comparison of full image builds on Make, Gradle without daemon and with daemon(D), as well as parallel Gradle builds . . . 60
31 Summary of all profile reports . . . 61
32 Summary of results - Comparison of evaluation blocks . . . 65

1 Introduction

Software development teams look to maximize value of the product being developed throughout the process lifecycle. Hence, they adopt Agile methodologies [1] which focus on rapid delivery of valuable software through iterative planning and quick feedback loops. This provides several advantages such as easy adaptability to changing requirements, enhanced visibility of the project and on-time release of software. Extreme Programming (XP) [2] has evolved as one of the popular Agile methodologies. XP propagates Continuous Integration (CI).

1.1 Motivation

Developing software for critical space avionics is a challenging activity. The size, complexity and the cost involved in software development makes it mandatory to have good software engineering practices followed at each stage of the software life-cycle. It is necessary to reduce the risk associated with the project by allowing a mechanism to track each change made to the software and help detect errors early in the life-cycle. Software projects containing modules with a low coupling factor can be developed independently, possibly in distributed environments. At the time of release, the different features developed are often integrated to build the final software product. This integration in an embedded project will be hard to carry out if there does not exist a Continuous Integration (CI) framework. In projects adopting CI, build automation is a primary activity. The methodology used by the build tools significantly affect the effectiveness of the CI workflow for the project. Most teams require the developers to spend less time in trying execute the builds and tests on the software that is developed. There is a need for a successful automation of the chain of activities post the commit stage. This need triggers the study of build automation practices and the influence they bring in CI workflows associated with the project.

As part of this study, the primary objective is to investigate the set-up of an effective and efficient Continuous Integration (CI) workflow for an embedded space avionics software development project using declarative build automation paradigm. The study involves implementing the set-up in a live project and draw conclusions based on how the framework handles real-life scenarios. The study also attempts to quantify build automation systems in the space avionics domain by identifying parameters for performance, build design and complexity analysis.

Continuous Integration (CI) is a practice where developers in a project push to a central mainline several times. The software development environment containing the CI tools and other build tools perform the builds and tests in an automated manner. The developers then expect to receive status of the builds and tests quickly. The CI tool consists of an open source automation server such as Jenkins [3]. The Figure 1 shows a global view of the CI concept. Every project has multiple developers working on their respective features. It is important to track the validity of the individual contributions

continuously during the development life-cycle. A continuous integration server such as Jenkins runs on a remote machine. Commits made into the central repository triggers the activities on the CI server. The status of the activities are then communicated back to the developers.

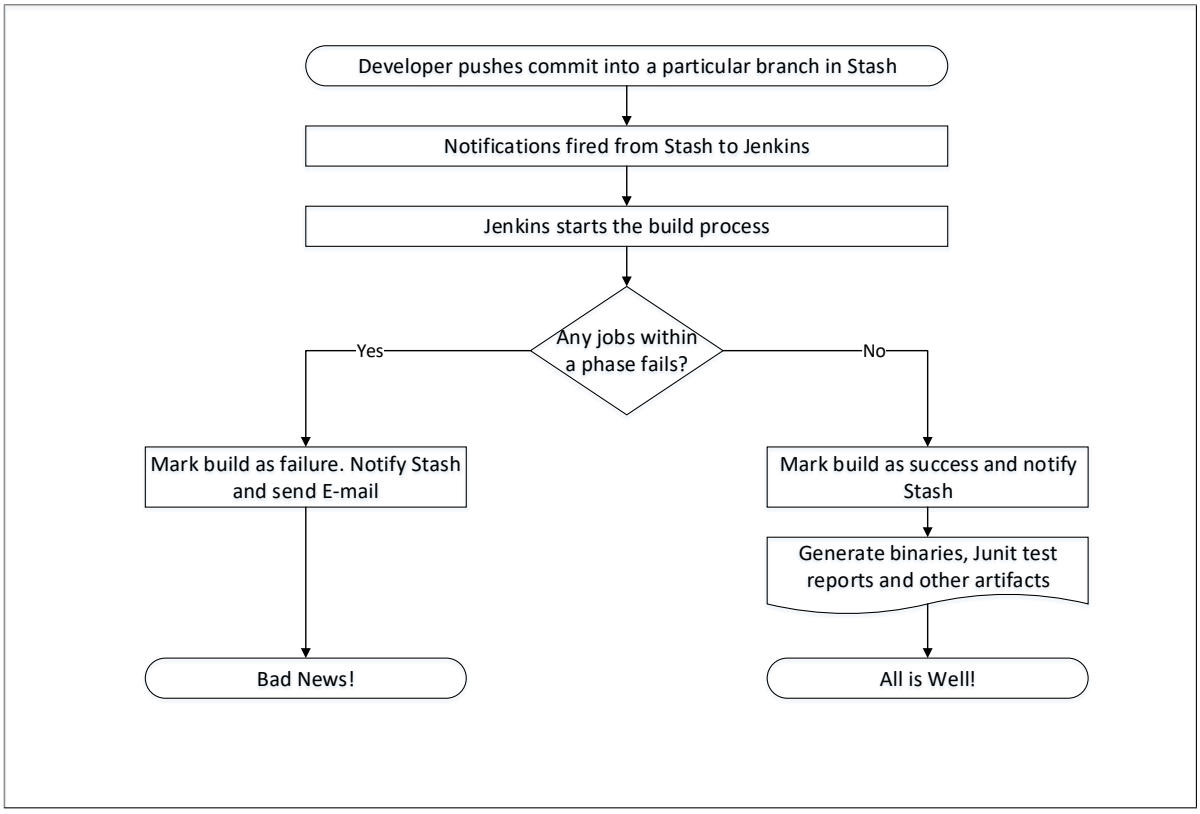

Figure 1: Block Diagram to show Overview of CI concept

The nature of the development project influences the type of CI workflow that the project adopts. For starters, the terminology used in an embedded software development project is slightly different from other development projects. For example, the term *build* in the embedded domain mostly refers to the compile & link processes only. The unit testing is handled separately as is verification & validation testing. If Git is the version control system in use, then the quality standards on the different types of branches are also quite different. The *master* could be chosen as the branch which is stable at all points of time. Hence, quality checks at *master* could be significantly more strict than in other branches.

Continuous Integration (CI) advocates having multiple commits into the central mainline every day. For every single commit, the build is executed from scratch. CI also advocates a higher integration frequency. However, depending on project specific constraints, the development team is allowed to fix the integration frequency. In any case, a team utilizing effective CI model should treat the periodic integration as a non-event [4].

This study attempts to provide a method to quantify build systems. This research is im-

plemented as an Action Research (AR). A detailed analysis of the existing build system is done. A discussion with the developers is performed to identify build activities which they commonly use and the challenges that they face with these activities. This initial phase gives rise to a set of observations, opinions and ideas. They are then taken into consideration to form a hypothesis of a new CI framework. A predominantly bottom-up approach towards Continuous Integration is done. A CI workflow is created based on the development Gitflow workflow. And the best practices of CI are then applied to this framework. A declarative build tool (Gradle) is used to express the build logic. At the time of design, the developer requirement as well as build design best practices are taken into consideration. A glue logic free system is one of the objectives of this study. This objective is achieved by extending Gradle and leveraging inherent CI tool (Jenkins) functionalities. Better performance is targeted. A reduction in the build-run time can translate to faster feedback and hence on-time release of software. Good software engineering practices are adopted, optimizations are studied, and continuous iterative development is carried out to achieve faster and comprehensive builds. The study also provides a definition for maintenance complexity of build logic when implemented in Gradle. The embedded nature of the project and inherent design methodology of existing build logic are two main factors influencing this definition. Opinions and experiences of the users of the system are taken into account. Additionally, literature covering similar studies are investigated and the knowledge has been incorporated in this study. Thus, the results for measuring complexity that are obtained are a mixture of quantitative and qualitative metrics. Taking the outcomes of these activities, the constructed hypothesis was verified.

Context of Study

Software development in embedded critical space avionics is large, complex and expensive. Experience suggests that software problems might result in failed missions. This study is done in collaboration with the department of On-Board Software (OBSW) development, Airbus Defence and Space GmbH in Friedrichshafen. The project under study is the development of central flight software (CSW) which runs on-board a satellite.

1.2 Thesis Overview

This thesis is divided into seven sections. Secton 1 is an introduction to the thesis. Section 2 deals with the background behind the study. It introduces two important components - Continuous Integration (CI) and Build Automation tools. Section 3 provides an overview of the Action Research (AR) methodology on which this study is based. In addition, it describes the Qualitative methods employed in the study. Since this study is performed in an industry, Section 4 provides a detailed report on the context of the software development project in the concerned industry. It describes an overview of the project, the tools used and the workflows adopted. Section 5 explores the Gradle build system which is designed for the project. It covers the terminology used and detailed design descriptions. Section 6 defines the Continuous Integration (CI)

workflow designed for the project as part of the study, its benefits and the challenges that were mitigated. Section 7 is an overview of the evaluation done to quantize build systems and the results obtained. Section 8 provides a Summary of the thesis as well as proposes future work that can be undertaken as an extension of this study.

2 Background

This section provides an overview on two important components of this study, Continuous Integration and Build Automation.

2.1 Continuous Integration

Kent Beck introduced the concept of Continuous Integration (CI) in its modern style as part of the Extreme Programming (XP) methodology about seventeen years ago [2]. Today, this practice has gained popularity in the field of software development. The practice encourages developers to share their working copies with a central mainline several times a day. The contributions of each of the developers needs to be merged to build up to the final product. The idea is to integrate code often, reduce the workload on the developers post the commit stage and receive frequent and fast feedback of the work.

There are a set of key practices which make CI effective for software development teams [4]. A few of these practices are handled in this study.

1. **Automation - Automate the Build**
   Build automation is when the source code is converted into a binary such as an executable or a JAR file depending on the type of sources. Several tools are available to perform this conversion. As part of this study, two tools - GNU Make and Gradle will be examined in detail.
2. **Comprehensiveness - Every Commit on the Mainline should Build**
   Another best practice is to build and test every single commit which is pushed to the mainline. This can be achieved by using a continuous integration server such as Jenkins.
3. **Visibility - Make it Easy for Anyone to Get the Latest Executable**
   Additionally, it should be easy for the users of the system to retrieve the executable and use it for their respective needs. Care should be taken to ensure that only the authorized users are capable of getting the executables.
4. **Transparency - Everyone can see what is happening**
   In addition, a good CI design would enable all authorized team members to view the status of the complete system at any point of time.
5. **Timeliness - Keep the Build Fast**
   One primary objective to invest in CI is to receive fast and comprehensive feedback on the status of the build. The framework should be designed to achieve this goal.

The project under study does not have a notion of continuous deployment. Also, it is not a requirement that every member of the team should commit into the mainline everyday. However, every commit that is eventually made should be processed and evaluated.

As part of this study, we would like to coin a term CI run. This is a set of activities

covered in the CI set-up such as checkout, build, unit test, validation test, static code analysis as well as ad-hoc build related activities. Earlier, we termed timeliness as a CI best practice. Hence, it is important to consider a notion of turnaround time whose bounds are dependent on the length of a CI run in the software development project.

An effective CI workflow brings about a large number of benefits to the team. There is a possibility to capture errors very early in the life cycle. This would reduce the cost of building the software. A main reason to adopt CI for projects is to not end up in what is termed as an "integration hell" [5]. This is effectively handled better when integrations are done more often rather than at the time of a release. CI helps to reduce repetitive manual processes. It also provides better project visibility. Developers start to notice trends in the state of their builds and periodically measure quality of the product. There is also an increase in performance leading to on-time releases. Some of these benefits have already been studied and documented in various literature surveys [6][7].

On the other hand, adopting Continuous Integration for a software development project can be a challenging activity. There are several articles, case studies and literature reviews which bring out the challenges that teams adopting CI face at the time of development. The nature of the software being developed could influence the CI workflow, especially if the project involves a close relation to hardware [8]. If the project size or complexity is large, CI processes need to be thought out with careful considerations. These factors might prolong the release cycles for the project thereby resulting in a slow feedback loop to the team members.

Another important challenge to be considered is the availability of hardware to run the builds. As discussed earlier, the builds need to be fast. And powerful hardware resources help to provide quick feedback of the build runs [9]. Software tools provided by the environment in which the build runs also play a significant role towards an effective CI workflow [10]. A good CI design would constantly strive towards seamless integration of software development tools with the CI framework.

2.2 Build Systems

The process of automatically creating a software build for a set of source files is called as build automation. It includes activities such as compiling, linking, running unit tests and running quality checks (eg. Static Code Analysis) on the source code. For automating software builds, a large number of tools are available. This thesis incorporates discussion of two such tools, GNU Make [11] and Gradle [12], in detail. In the field of software builds, an *execution unit* (EU) is defined as an activity which the build tool carries out. In Make or Ant they are called targets. In Gradle they are termed as tasks. Examples for EU's can be compilation of C sources, or cleaning of generated object files for a new build, creation of an executable and much more. Usually, EU's *depend on* other EU's. The build tools also offer a large number of ways in which these dependencies can be configured. A *control flow* is the path which the build tool traverses in a build script to execute the build logic. For example, generation of an executable from C sources

involves compilation of the sources, then a stage of link followed by packaging into a binary. Essentially, EU's along with their dependencies combine to define the control flow in software builds. The order of execution of the various EU's may be different and is based on the build author's requirement. Hence, a build script usually contains several targets which in turn means several control flows.

Two important terms that can be associated when discussing build systems are *expressiveness* and *conciseness*. Expressiveness is the capability of the build tool to describe complex build behaviour. Conciseness is when the build tool expresses the build logic in an easy to understand manner. In other words, the control flow of the build logic should be realized by the users of a team without difficulty. More often than not, these properties do not scale proportionately. When build logic tends to become complex, the conciseness of the build script decreases. This trend can be traced back to the build tool that is used to describe the logic. To express this in the form of a graph, consider Figure 2. The traditional build tools such as Make or Ant generally fit into quadrants two or three of the graph. Modern build tools such as Gradle try to be concise at the same time provides opportunity to represent complex build behaviour. Hence, they are slotted in at quadrant four.

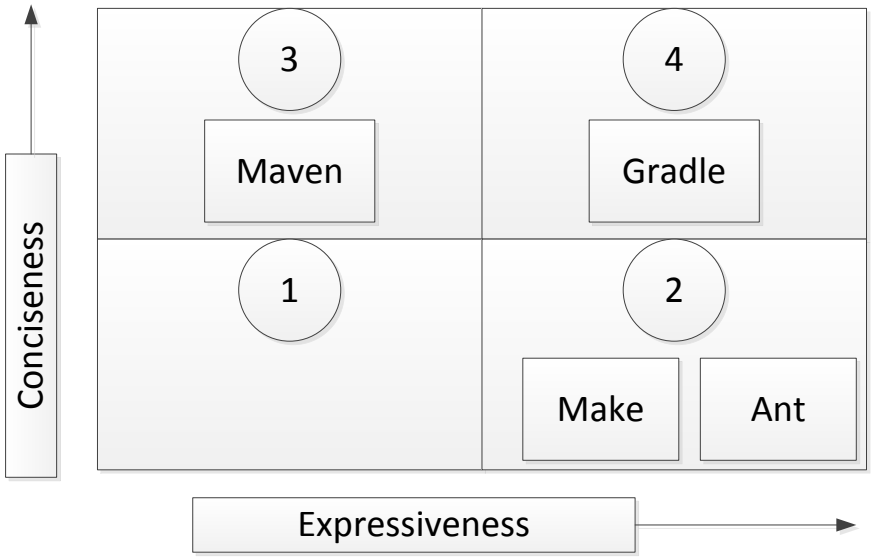

Figure 2: Expressiveness vs Conciseness - How the different build tools fit into this comparison graph?

A classification of build tools can be done on the basis of how they are used by the build author. One type is the *imperative* build system. Examples include GNU Make or Apache Ant. This means that it is the responsibility of the build author to define what the build should do and also how the automation tool should do it. In case of a

declarative build system, the build author is responsible for stating what the build logic should do. The build automation tool attempts to figure out how the requirements are to be met. This *build by convention* approach provided by declarative build systems enable to increase the conciseness of the build logic.

Build automation is generally considered as a CI best practice [4]. To interface the build tool with a CI tool, a framework needs to be designed and implemented. This is called *glue logic*. The implementation of this glue logic is done usually using shell scripting. However, if the build logic is inherently complex, this interface might be difficult to establish and maintain.

This section attempts to describe in brief about the build systems which are used in the project under study. A more detailed analysis of the build logic in the Make based and Gradle based build systems will be discussed later.

Make Make is a popular build automation tool which is responsible for creating an executable or a library from source code. It manages to do this by parsing a file called Makefile. Software development of the Make tool started more than 35 years ago [13] and there are several variants that are currently available. A very popular and widely used variant is the GNU Make. The format of the Makefiles are similar to one shown below [11].

```
target : prerequisites
    recipe
```

target can refer to two things. One is the name of the binary that Make generates. Generally, for C/C++ sources it is an executable or a library. *target* can also refer to an activity which Make carries out like compile, link or clean. ***prerequisites*** are inputs to the Make system for executing the target. They could be source files, or even other *targets*. A ***recipe*** is a set of commands which Make carries out. The *recipe* can consist of several commands. By default, a tab space must be included at the beginning of each *recipe* line.

The GNU Make build automation tool provides advanced features to enable build author's to describe complex build logic. It provides the concept of implicit rules [11]. These rules reduce the work load on the build author as it forces Make to use customary techniques for the desired behaviour. For example, there are built-in implicit rules which use several Make provided variables in their *recipes* like CFLAGS. There is a possibility to use the Recursive Make [14] functionality. This practice is commonly used when there are separate Makefiles within each subsystem of a larger system. In this case, Make recurses through each of the subsystems and executes the *targets* accordingly. However, the harmful effects of using Recursive Make to software projects is well researched in [15]. GNU Make also provides a mechanism to generate dependency information automatically [16].

Ant Ant [17] is an open source, software build automation tool similar to Make but targeted primarily for Java sources. It was developed as part of the Apache Tomcat project in 2000. It requires a Java platform, and uses an XML to represent the build logic. The default XML used by Ant is the *build.xml*.

Each Ant build file consists of a project and at least one target. Targets are further made up of Ant tasks which can be executed to obtain the desired build behaviour.

- **Project**
  Projects are represented in XML using the <*project*> tag. Three attributes are defined by the Project element. *name* is an optional attribute which denotes the name of the project. *default* denotes the default target for the build script. Any Project may contain multiple number of targets. This attribute defines which target should be executed by default. An optional attribute *basedir* which denotes the root directory for the project.
- **Target**
  Targets are represented using the <*target*> tag. They are a collection of Ant tasks. They have a *name* attribute and a *depends* attribute. The former defines the name of the target and the latter describes the targets on which the current target depends on. The *depends* attribute defines the order in which the targets are to be executed [18]. *description* is an optional attribute in which a short description for the target can be written. There are also some conditional attributes such as *if* and *unless*.
- **Task**
  Ant tasks are the commands which need to be executed by Ant. Tasks could be similar to an *echo* command which prints information on the terminal or a *javac* command which does compilation of the defined Java sources and the classpath.

A <*project*> can contain <*property*> element which allows to specify properties such as Ant version, Ant home directory location and much more.

Gradle Gradle [12] is also an open source build automation tool but replaces the Ant XML build files with a Domain Specific Language (DSL) [19] based on the Groovy [20]. It is a tool which provides for a declarative modelling of the problem domain and hence has a build-by-convention approach for software builds. Gradle is already a popular choice for build automation among many enterprises such as Google, Android, and Twitter to name a few [21].

Gradle defines ***tasks*** which is an activity carried out during the build such as compile, link or clean. The default name of the build file is *build.gradle*. Gradle is based on a graph of *task* dependencies, where the *tasks* do the actual work. These *tasks* could be custom tasks created by a build author. It also defines some default *tasks* depending on the configuration that has been mentioned in the DSL. This can be explained with the help of an example.

The first line in Figure 3 includes the *C plugin* into the build file. This line extends the

```
apply plugin: 'c'

model {
    components {
        main(NativeExecutableSpec) {
            sources {
               c.lib library: "hello"
            }
        }
    }
}
```

Figure 3: An example of a *build.gradle*

Gradle project's capabilities. It configures the project based on the conventions described further in the build script. For example, it adds some specific tasks or configures defaults. The Gradle model is a container for configuring the build logic. Based on the DSL specified within the model, Gradle recognizes key configurations and creates a control flow for the build by mapping a set of default tasks which the build author can leverage. For the above mentioned build file, Gradle creates the tasks shown in Figure 4.

```
Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project. [assemble, check]
clean - Deletes the build directory.
installMainExecutable - Installs a development image of executable '
   main:executable' [mainExecutable]
mainExecutable - Assembles executable 'main:executable'.
    linkMainExecutable - Links executable 'main:executable'
```

Figure 4: The output of *gradle tasks* command

The declarative nature of the build tool can be visualized using the above mentioned example. Based on the configuration defined by the build author, Gradle creates the build, clean, install, and link tasks for generating an executable. A detailed study of Gradle's methodology and its fit in an embedded space avionics software development project is analyzed in subsequent sections.

3 Research Approach

The Research Approach used in this study is described. Additionally, a small portion of the results obtained at the end of this study is qualitative in nature. Hence, the practices adopted to bring out qualitative metrics is also discussed in brief.

3.1 Action Research

Action Research (AR) is a methodology where researchers aim to solve real-world problems while simultaneously analyzing the approach employed to solve the problem [22]. Literature shows that the methodology is cyclic [23]. There is an intention or a plan to initiate an activity. This precedes action and a stage of review follows as shown in Figure 5. A prerequisite to Action Research is to have a problem owner who is responsible to both identify a problem as well as take steps towards solving it. There are a large number of key ideas which were developed through an implementation phase in real-world projects [24]. One major outcome of AR is an in-depth and first-hand understanding of the subject under study that the researcher obtains [25].

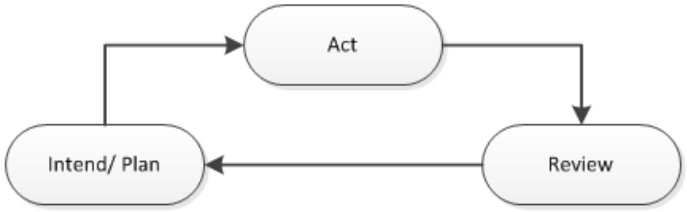

Figure 5: Action Research (AR) cycle consisting of three distinct stages

AR, when considered as a form of study, also presents challenges to be addressed. Since, AR brings about a notion of change to the project, an alignment with organizational level objectives need to be maintained. Also, the problem of researcher bias at the time of research might be a risk in the study. Finally, the cost of carrying out AR is to be estimated as it could prove to be expensive [22].

As mentioned earlier, the study of an alternative build automation tool in the field of space avionics to augment to an effective CI workflow is the objective of this study. The researcher in this study is the build author/CI developer. The researcher collaborates with the members of the software development team to analyze the existing structure of the project. The challenges which need to be mitigated are investigated. In this scenario, the researcher sees this as an opportunity to employ a declarative build paradigm to study first-hand how it would affect the build in an embedded environment. Observations and comments are made during and after development. These observations are then used to either support or refute a hypothesis.

3.2 Qualitative Metrics in Study

Quantifying build automation tools is inherently a difficult proposition. This study employs qualitative methods to derive metrics based on a small subset of results. For achieving this, a formal methodology to incorporate these methods into the study is followed. Literature provides starting points to employ qualitative methods systematically [26].

Qualitative metrics provide many advantages to the researcher. Results are richer and informative. Parameters which are hard to be expressed objectively can be explained in a subjective manner. The comments of the developers working on the project are taken into consideration and can be analyzed to offer a concise picture of the state of work.

Qualitative methods involve a phase of Data Collection followed by the phase of Data Analysis. These phases contain different methods for handling data. The methods adopted as part of this study is outlined.

Data Collection

This study incorporates two different data collection methods. They are *Participant Observation* [27] and *Interviews*. In *Participant Observation*, the researcher collaborates with the developer in gaining knowledge of the existing system. The developer is encouraged to verbally describe the activities which he carries out so that the researcher is able to understand the process flow. Additionally, the researcher attends the developer team meetings and records key ideas expressed during the meeting. As these meetings are generally organized periodically in a software development team, the researcher is in a position to record data such as terminology used, technical information exchanged as well as identify roles of the developers within the scope of the project. The second technique, *Interviews* [26], is used specifically to collect individual developer opinions and impressions about the subject under study. The researcher has employed a semistructured interview pattern in this study. The interview begins by having specific questions and gradually proceeds towards open-ended questions. The ideas expressed by the developer were recorded by taking notes. This technique of *Interviews* also provide a mechanism for the researcher to capture developer requirements in the study being carried out.

Data Analysis

Once the data has been collected through the methods described above, an analysis of the obtained data is conducted by the researcher. This involves two methods - *Generation of Theory* and *Confirmation of Hypothesis*. Based on the text which is generated by Data Collection, a preliminary processing is done. This processing is called as Coding [28]. Opinions which are similar or about a particular theme are grouped under a label. For example, consider the following statements made by different team members.

Developer 1 (D1): I would like to find out if this tool would give better performance (faster builds) ...

Developer 2(D2): The existing system takes about ten minutes to create the executable. If this time period can be reduced, it is a good thing ...

Based on the text above, a label called Performance can be introduced. Similarly, different themes can be drawn out from the text and a hypothesis can be generated.

The next method under Data Analysis is *Confirmation of Hypothesis*. Here, a practice called as Triangulation [26] is employed. A particular parameter under study is analyzed from various angles. For example, lets consider complexity of build logic as the parameter under study. Qualitative data obtained from interaction with the developers contained text such as:

Developer 1 (D1): There are many environment variables in the build logic and I am not sure where they are set ...
Developer 2 (D2): There are a large number of Makefiles that are included when I run make *on my sources. I am not really sure what these Makefiles do ...*

From a quantitative perspective, literature surveys can help point out measures for complexity of code based on source lines of code (SLOC), or occurrences of indirection. Hence, it provides an opportunity for the researcher to analyze the parameter from various perspectives. Based on this, it is possible to support or refute the hypothesis that is set.

Most of these methods if not all have been used in this study. Further details of how these methods have aided to help drive the research is discussed in subsequent sections.

3.3 Validity Threats

As this study involves the use of qualitative metrics during data analysis and result formulation, it is necessary to guard against potential threats to the validity of the research. Triangulation which was discussed earlier is one way to validate the results obtained.

The number of members involved in the software development project was seven. This limited sample space is a potential threat in the study. However, the number of interviews conducted with the members were exhaustive in nature. This means further interviews would lead to a low amount of significant information. This study is performed by only one primary researcher. However, all design level decisions, interview protocols, findings, and each iteration in development has been peer reviewed and discussed with two secondary researchers, one internal to the project (supervisor) and one external researcher(at the university). In this form, the reliability threat to the study is mitigated.

One important threat in software engineering study is internal validity threat. This refers to a risk arising as a result of interference in causal relations within the research.

Assume a parameter under study X is affected by Y. There exists a possibility that a possibly unrelated factor Z might also affect X. In this study, the researcher aims to study the challenges in Continuous Integration set-up in an embedded environment with emphasis on build automation practices. An initial round of discussions with some stakeholders in the project led to feedback regarding license issues faced in setting up infrastructure for the Software Development Environment including the CI server. Even though this is a valid comment, the researcher should be in a position to decouple the data obtained in the research.

This section described an overview of the research approach used in this study. The analysis also quoted various literature articles describing about the methods in detail. The next section explains the nature of the project under study.

4 On-Board Software Development (OBSW) - AS400 Central Software (CSW)

The case organization chosen for this study is the Space Systems satellite On-Board Software Development (OBSW) department at Airbus Defence and Space, Friedrichshafen. This chapter provides an overview of the software development activities for AS400 Central Flight Software (CSW) as well as a description of the existing Software Development Environment (SDE) for the same.

4.1 Overview

The actual software that runs in an On Board Computer (OBC) [29] in a satellite in operation is the On Board Software (OBSW). The OBSW is treated as isolated and independent software controlling the various applications such as power systems, propulsions, sensors and payload on a satellite. The AS400 Avionics development is an initiative towards a detailed definition and development of a generic, re-usable high power avionics system to be used on a variety of missions.

4.2 Software Development Environment (SDE)

The term Software Development Environment (SDE) refers to a set of software and associated hardware tools which are used by members involved in the software development project. The environment supports activities such as configuration management, source code development, and project management [30].

The team for building the AS400 Central Flight Software (CSW) is composed of two groups. A production team which provides the flight code and a validation team which performs the validation activities on the provided flight code. The production flight code is in C and the validation test framework is in Java. The SDE that is used by these teams is also composed of two parts. A client side SDE used by developers which consists of development packages in a Windows/Cygwin environment with Eclipse TOPCASED. And a server side SDE consisting of the Atlassian Tool Suite (JIRA, Stash, Confluence, FishEye, Crucible). A brief overview of these tools is discussed.

Git: The version control system used is Git [31]. It is a distributed system. Every developer in the team has a working copy in his/her local machine. They are allowed to 'push' their changes to a central mainline to make it accessible for other users of the system.
JIRA: An Atlassian tool which tracks and manages projects [32]. It uses an Issue management system. Issues [33] are assigned to developers to manage work on different software features as well as other development related activities.
Stash: This tool is also a part of the server side SDE. It is responsible for managing the central mainline located on the server. The authorization of the users who are allowed

to use the central mainline is well administrated using this tool. A link to JIRA issue management is achieved which maps the development activity to the respective branches in the repositories.
FishEye and **Crucible:** Atlassian tools in the server side SDE. FishEye is used to extract information from repositories, such as code version differences. Crucible is used for requesting, performing and managing code reviews [34].
Eclipse TOPCASED: It is the standard Integrated Development Environment (IDE) used by the production team. It is a platform which contains many plugins required for software development. It is a client side SDE tool.

As with many embedded applications, this software system is also developed in a cross development environment [35]. This means that the machine on which the code is compiled and linked is different from the actual deployment machine. The former is called as host system and the latter as target. The existing SDE set-up utilizes GNU Make [11] as the build automation tool and GNU cross compilation tools for the project specific target space processors on the host systems.

Figure 6 shows an overview of the SDE used in this project. The blue lines indicate the links between the various applications. The straight black arrows indicate the interface between the team members and the SDE. The developers generally use the cygwin based terminal to push Git commits into Stash. The Issue tracking and Code review activities are generally performed using a web browser. The authorization for the users of this server is managed using Atlassian Crowd which provides integration with the corporate LDAP server.

The dotted arrows and the blocks in red are introduced as part of this research. As part of this study, the tools which will be integrated to the existing system are Jenkins and Gradle. Jenkins is an open source automation server which is used for continuous integration. An instance of Jenkins runs on the same server which also caters to the Atlassian Tool Suite. This instance is called as the Jenkins Master instance. The actual build steps are done on so called Slave Machines which are separate Linux based virtual machines. Jenkins receives a hook from Stash when there is a commit into the branches which Jenkins is monitoring. At the end of the build, Jenkins updates Stash with the status. Hence, the developer gets an all-in-one view on the web browser regarding the status of the commit. Clickable links are also provided which enables the developer to open and view the results of Jenkins activities. Gradle is the build automation tool under study which is installed on the Linux based machines as it is here where the build processes are expected to run.

One objective behind introducing these features to the SDE is to drive development faster by leveraging the powerful remote virtual machines to automatically build and package the software. The description and set-up of the slave machines and the installation of Jenkins instances are beyond the scope of this thesis.

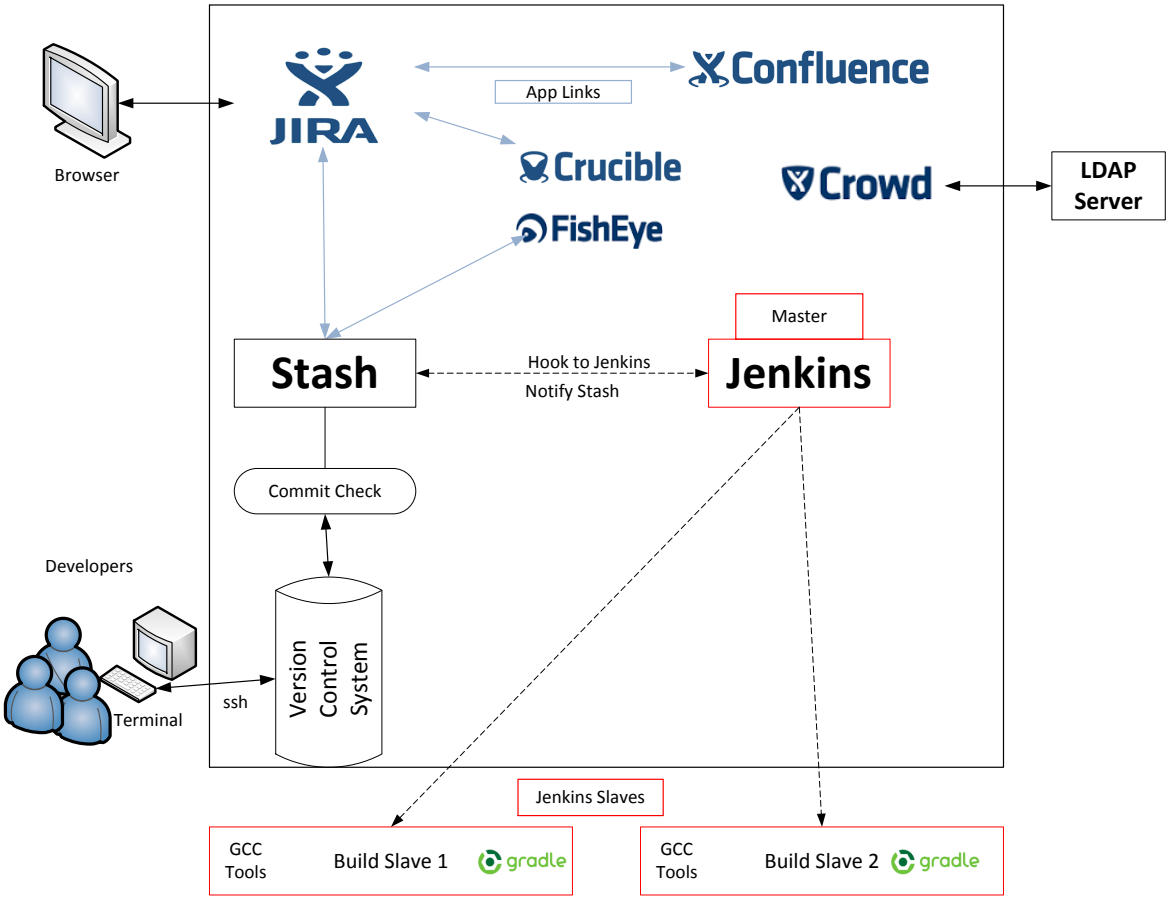

Figure 6: SDE set-up for AS400 Central Software Development

4.3 Software Development in OBSW - CSW

This section explains the project specific terminology, structure of the source code, git workflow and the build automation methodology employed.

4.3.1 Terminology

The source code for the AS400 CSW development is organized as a hierarchy. There are five levels of elements. The top most level is the production repository. The second level contains the different aspects of production such as project specific SDE scripts, source code ('fsw' folder), and libraries to be used in unit testing. The definition of the levels below this need to be distinguished using a view model. This study defines two distinct view models [36] for the hierarchy. They are the Developer's view model and the Build Author's view model.

Developer's View Model

- Applications - they are elements which implement functional processing that are associated with the satellite. Some examples include Data Management System (DMS) which implements data handling standards, and Attitude and Orbit Control Systems (AOCS) which maintains the orientation of the satellite.
- Components - Applications can be composed of smaller entities called Components. A typical Component is a function controlling an equipment on board the satellite.

Build Author's View Model

- Collections - The elements which are at the same level within the flight software (fsw) folder in the hierarchy are called Collections. They represent the various parts which build up to the Central Flight Software (CSW) executable.
- Constituents - Subordinates of collections which generally contain the source files are called constituents. They usually combine together to give the notion of an Application. Some Constituents contain further Sub-Constituents.

It should also be noted that Collections do not necessarily map directly towards Applications. Also, it is not mandatory for every Collection to contain Constituents or for every Constituent to contain Sub-Constituents.

4.3.2 AS400 CSW Architecture

As mentioned, the AS400 Project consists of several repositories. Figure 7 shows the various repositories that are used in this study. There are dedicated repositories for production (as400prod) and Validation (as400val). There is also a repository which contains the Make based build logic that is not project specific but shared amongst various related projects. In addition it contains an interface with the SDE for tools which perform Static Code Analysis and Unit Testing. The scripts repositories contain the Glue Logic which creates the interface between the build tools and the CI tools. In addition it contains some helper scripts which allow for integration with Jenkins server. The hierarchy of the production repository is similar to the code tree shown in Figure 8. Some of the Collections within these repositories are git submodules [37]. The folders in red are git submodules. The folders in blue are Constituents. The sub-directories within the folder 'fsw' are Collections. This is only an excerpt of the original structure. The actual code tree consists of a larger number of Collections and Constituents. Collections may or may not contain Constituents. The production repository as viewed from the top level of the AS400 Project is self contained in terms of production code.

4.4 Software Development Workflow

The AS400 CSW team employs a Gitflow Workflow [31] for software development. There are five different types of branches – *master*, *develop*, *release*, *feature* and *bugfix* branches.

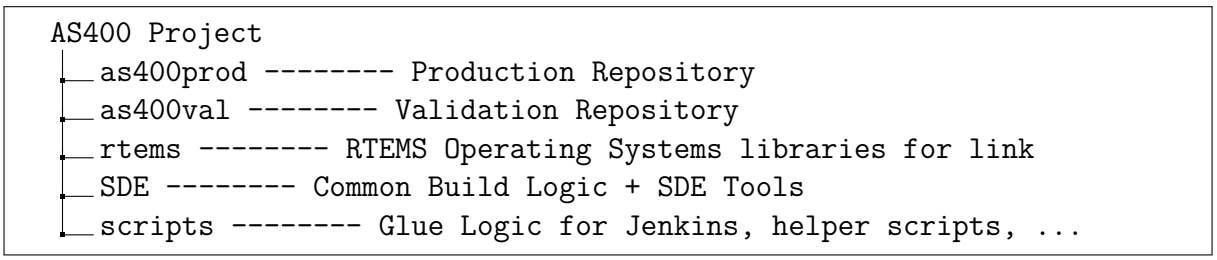

Figure 7: The AS400 Project top level view.

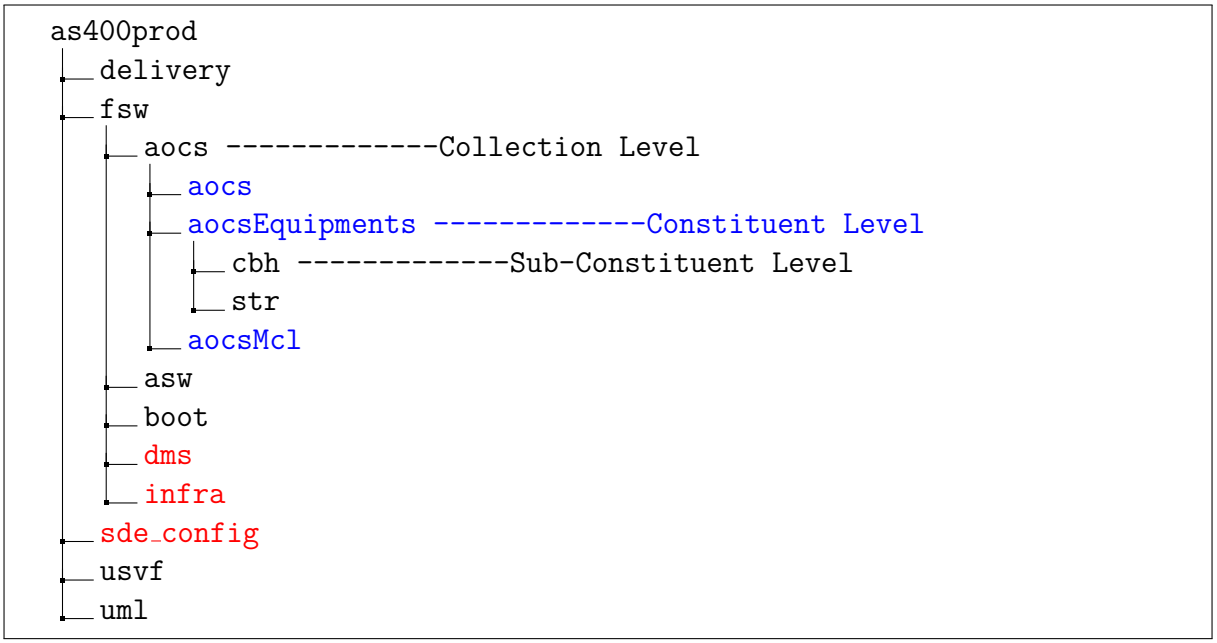

Figure 8: An excerpt of the AS400 Production Repository Structure

master contains the official release history. It represents a stable, fully tested release. There is always just a single master branch. Integration of the various software features are done on *develop*. Generally, there is just a single develop branch. The different features are merged into develop periodically and the team ensures that the required validation tests are done on the commits in develop.

The *feature* and *bugfix* branches are used primarily by the developers to work on individual applications. In the organization under study, JIRA issues are called as Software Modification Reports (SMR). The *feature* branches are created when an SMR is issued. They are usually derived from the *master* branch. It is then used by a developer (or a team of developers) to perform the modifications required by the SMR. Commits are made regularly to the branch during development. Ideally, the developers would like to receive fast feedback of the status of their commit when working with *feature* branches. If the feedback is positive, and if there is a need to merge the contents of the *feature* branch with the *develop* branch, the developers can issue a Pull Request [38]. Just ahead of a software release, a new branch called *release* is forked off from *develop*.Ideally,

this branch is used for bug fixes, documentation generation or other release-oriented activities. When it is ideal for a release, it is merged with the *master*.

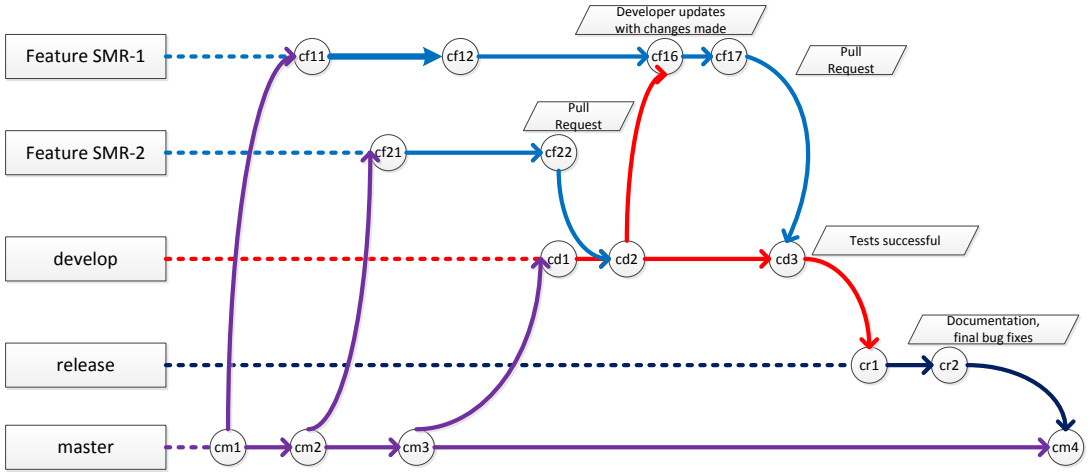

Figure 9: Representation of the Gitflow Workflow containing master, feature, develop and release branches

Figure 9 shows an example of how this workflow is used in general by the developers. The developers derive *feature* branches from the *master* branch. So *cf11* and *cf21* are the first commits on the *feature* branches SMR-1 and SMR-2 derived from *cm1* and *cm2* (first and second commits on the *master*) respectively. At *cf22*, the developer is finished with the work and wishes to integrate with the develop branch. A Pull Request is made and the merge is carried out. The developer working on *feature* branch SMR-1 now gets the content of *develop* and continues the work until *cf17*. Another Pull Request is made and the *develop* is integrated. At a certain point when the *develop* is ready for a release, like at *cd3*, it is forked off on to a *release* branch and after final fixes (if required), it is merged with *master*.

However, for the organization under study, the *release* branch is used in a different way. At times, *feature* branches are integrated directly into *release* before a release. This is done to maintain a record of the different features that were integrated for a particular release.

4.5 Build Automation in AS400 CSW

This section describes in detail the tools and the methodology behind the builds in AS400. In the production environment, the builds for this project are done using GNU Make. In validation, the builds are done within the Eclipse IDE. However, as a headless

framework is required for running the builds on remote machines, Apache Ant is used as the build tool. For compilation and linking, GCC tools (part of GNU software) for usage with the RTEMS operating system are used. For Java sources, an eclipse java compiler is used. The build system used in the AS400 Project is an inherited system. It has been used previously on successful software projects developed on similar platforms.

AS400 Production Build Process

The next part of this section describes the build logic employed to compile and link the production sources into respective binaries at different points during the build stage. The sources are compiled using Makefiles which are defined in each of the appropriate Collection and Constituent directories within the source code tree. There are also several 'common' Makefiles which define variables and rules. The build system uses a recursive Make [39] model. Essentially, the CSW is to be delivered as an executable file. To achieve this goal, the build system uses the technique of Partial Linking [40]. Partial Linking generates a relocatable output file which can in turn be used as an input to a linker. It can be achieved easily by appending the *-r* option to linker arguments. When Partial Linking is invoked, the unresolved references remain unresolved, i.e the errors which a linker raises in normal operation are suppressed. Additionally, this method of linking eliminates duplicate copies of debug sections and merges the symbol table into one. The output of this link stage produces a file which is called as Partially Linked Object (PLO) file.

Assume we are considering the AOCS collection in the source tree. It is represented in Figure 10.There are Makefiles within each Constituent and Sub-Constituent directories (in red) as well as within the Collection (in blue). The content of the Makefiles at Collection level and Constituent levels are shown in the Figures 11 and 12.

As this example deals with invoking Make from the aocs Collection level, the process recurses through the various Constituents within this level. Some Constituents like 'aocsEquipments' contain further Sub-Constituents. The invocation also recurses to those levels. The default target defined in these Makefiles will be executed. In this case, it is *linkpart*. Initially, the sources within the Constituent or Sub-Constituent directories are compiled. For Constituents which contain Sub-Constituents, each Sub-Constituent will then perform a partial link on the generated objects. Constituents which do not contain Sub-Constituents also perform a partial link on its generated objects. This produces a single Partially Linked Object (PLO) file at the respective Constituent or Sub-Constituent level. For Constituents which contain Sub-Constituents, the Makefile at the Constituent Level then performs a partial link on all PLOs generated by the Sub-Constituents. This gives rise to a PLO for that Constituent. Once all Constituents have generated their PLOs, the process now shifts to the Collection Level. The Makefile at the Collection level does the next stage of link. Here all the PLOs produced at its Constituent levels are again partially linked. The build process is so designed that every PLO that is generated is stored at a level higher than the directory in which the *linkpart* target is executed. This is a common behaviour exhibited by all Collections in the

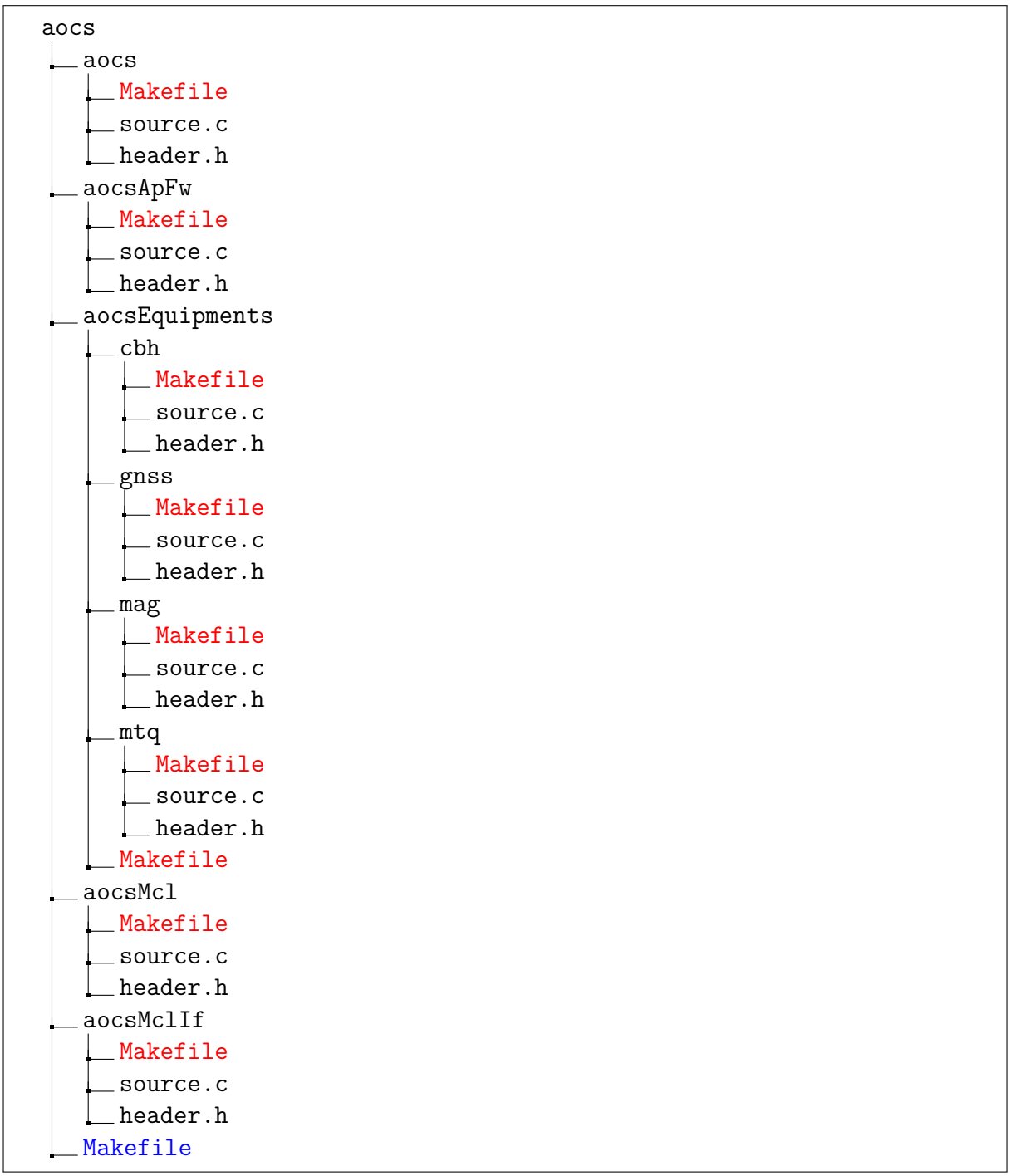

Figure 10: Source Tree of one of the Collections containing Constituents and Sub-Constituents

AS400 source tree. As mentioned earlier, as all Collections are located within the 'fsw' directory, all PLOs of the various Collections are written to a 'lib' folder also present within this directory. At the time of generating the complete CSW executable, there

```
DEFAULT_MAKE = linkpart

# EXTERNAL_LIB : Definition of libraries
    to use for link, either local to the
#                 directory or distant
EXTERNAL_LIB = lib/libaocsApFw.o \
           lib/libaocs.o \
           lib/libaocsEquipments.o \
           lib/libaocsMclIf.o \
           lib/libaocsMcl.o \

LIBDIR = $(OBSW_PATH)/lib

# EXTERNAL_LIB_INST : Definition of
    libraries to use for link in
    instrumented mode, either local to the
#                 directory or distant
EXTERNAL_LIB_INST =libaocsApFw.o\
           libaocs.o \
           libaocsEquipments.o \
           libaocsMclIf.o \
           libaocsMcl.o \

# INCLUDE OF GLOBAL RULES
#-------------------------
include $(CPL_REP_MAKE)/Make.rules
```

Figure 11: Makefile for 'aocs' Collection

```
DEFAULT_MAKE = linkpart

EXTRA_INCLUDE = -I$(AOCS_PATH/
    aocsEquipments \
                -I$(AOCS_PATH) \
                -I$(DHS_PATH) \
                -I$(INFRA_PATH) \
                -I$(IO_PATH)/busMgr \
                -I$(IO_PATH)/busCplr \
                -I$(IO_PATH)/pmCplr \
                -I$(IO_PATH)/rmapCplr

# SRC : Definition of sources to compile
SRC = AocsApFw.c \
      AocsSync.c \
      AocsAsync.c

EXTRA_CLEAN = AocsApFwParam.c

# INCLUDE OF GLOBAL RULES
#-------------------------
include $(CPL_REP_MAKE)/Make.rules
```

Figure 12: Makefile for 'aocsApFw' Constituent

is a final link stage where all the PLOs of the different Collections are linked together to produce the software executable. Alternatively, depending on the target defined in a Makefile, the build system also provides for building a static library from the sources. For certain Collections like 'infra', these libraries are used at the final link stage to create the Executable. Hence, it can be observed that three different types of binaries can be obtained - PLOs, executables and Static Libraries.

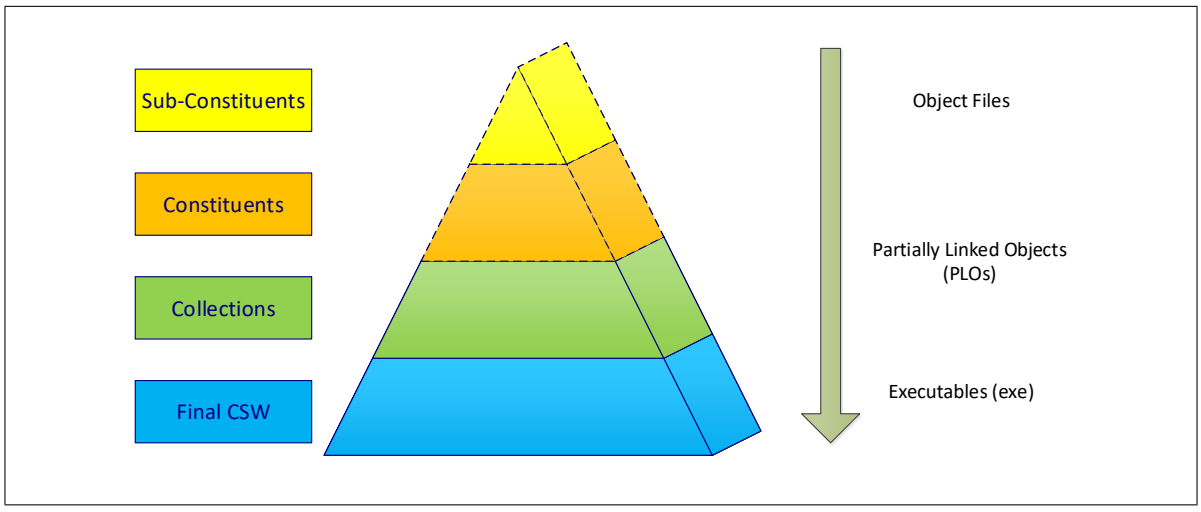

Figure 13: Representation of the build methodology from Build Author's View Model

The triangle graph in figure 13 shows the nature of the binary generated at the various levels in the source code hierarchy. It can be seen that the final Central Software (CSW) is actually built upwards from the sources contained at various levels. The dotted lines in the diagram signify that the Constituents and Sub-Constituents for a Collection are not mandatory. In these cases, a PLO is generated directly from the sources.

The build system also provides for building two different variants of the image. The variants differ primarily in the way the sources are compiled. The project defines two distinct database exports which contain several *#define* directives [41] for many identifiers used within the source codes. As these variants represent different configuration settings, the requirements to have two different image variants arises. These image variants are called the SYSDB (system database) and SWDB (software database). The build system allows for generation of these variants based on a command line property *DATABASE_DIR*.

The project also defines two other variants based on the platform for which the sources are compiled. They are the SCOC3 [42] and native PC. It should be noted that not all sources in the tree can be compiled for PC. A good example is the 'startup' Collection which contains assembler sources. Hence, the notion of a complete CSW executable for a PC does not exist. However, in the realm of Unit Testing, the compilation for PC platform is important for capturing bugs. For producing variants depending on the platform, different tool chains are used. For SCOC3, the GNU sparc-rtems compiler and linker are used while for PC the native GNU compiler and linker packages are used. The existing build systems allows for generation of these variants also based on a command line property *CPU*.

In addition to compilation, partial link and link, the build system provides targets for several other activities such as Static Code Analysis, Clean, Linecounter, and Code Instrumentation. The scope of this study does not cover all these targets comprehensively. However, some of these targets are handled and documented wherever applicable.

Now, let us analyze the Makefiles in Figures 11 and 12 . Each of these Makefiles includes the 'common' Makefile which is called Make.rules. It is like a root Makefile which defines and describes the targets used in these Makefiles. As shown in the figure *linkpart* is the default target in both these Makefiles. The activity carried out in these Makefiles is partial linking. In the Constituent's Makefile, the variable SRC determines the sources that need to be built. The EXTRA_INCLUDE variable adds a set of paths which the compiler traverses through to find the include headers for the sources. In the Makefile at Collections level, the EXTERNAL_LIB variable defines various partially linked objects which need to be sources to the linker. The LIBDIR variable defines the directory to which the output of the link process would be written to.

AS400 Validation

The validation activities on AS400 CSW is discussed next. OBSW development requires careful and comprehensive validation activities to ensure that the correct software is de-

livered. For Validation activities, the AS400 team uses a Software Validation Facility (SVF) [43]. It is a software test bench which provides a simulated model of the environment where the software is expected to run. It is considered as an hardware-in-loop emulation system as it contains a copy of the target space processor. For the project under study, the SVF is offered by an external team as a package which can used at the time of compilation of test sequences. Several versions of the SVF are released at various points in the software development and the validation team incorporates these changes appropriately.

All the sources required for validation are contained within the repository 'as400val'. The validation framework for this project is in Java. An Eclipse based IDE is used by the validation team members to build and run their tests on the generated CSW Executable. The validation tests contain several Collections based on the Applications to be tested. These Collections contain test sequences in Java. These validation test sequences require test classes developed by the team as well as the SVF packages in the classpath during compile time. Hence, a JAR file containing all these required classes is prepared. For the purpose of a headless build of this JAR, an Ant build XML is created. This is similar to the one shown in Figure 14.

```
<?xml version="1.0" encoding="utf-8"?>
<project>

    <property environment="env"/>

    <property name="build.compiler" value="org.eclipse.jdt.core.JDTCompilerAdapter"/>

    <target name="clean">
        <delete dir="build"/>
    </target>

    <target name="make">
        <mkdir dir="container"/>
        <javac srcdir="${env.CPL_REP_BASE}/simopsenv/src/obswTest;${env.CPL_REP_BASE}/
            test/benchConfig" destdir="container"
        source="1.8" target="1.8" encoding="Cp1252" nowarn="true">
                <classpath>
                <fileset dir="/data/products/libs/">
                        <include name="**/*.jar"/>
                            </fileset>
                </classpath>

                <compilerarg compiler="org.eclipse.jdt.core.JDTCompilerAdapter"/>
            </javac>

            <jar destfile="nsvf.jar"
                        basedir="container"
                        includes="**/*" />
        <copy file="nsvf.jar" todir="${env.NSVF_DIR}/svfJarfiles/" />
    </target>
</project>
```

Figure 14: *build.xml* for AS400 Validation

After the JAR file is generated, the required test sequence is compiled and executed using shell scripts provided by the SDE. These scripts are capable of running tests individually

or in batch mode. Alternatively, the IDE provides a tool called TMA to run the tests and generate test reports. Hence, the JAR file lies in the classpath of the test sequence execution command. It should be noted that the developer test classes which are a part of the JAR may be modified differently by the different members of the validation team. One developer should not break another developer's tests. Hence, the framework is so designed to compile all test sequences after the JAR has been created. This detects if any of the test sequences would be broken as a result of changes induced on the sources in the JAR.

Automating the validation build framework on remote machines is a challenging activity. The simulator for the test is an executable which has the constraint that at any point of time, only a single test instance can run reliably in a machine. Hence, it should be ensured that there are no parallel running processes at the time of a test invocation. Since, the compilation of JAR file is natively done by an IDE, it effectively encapsulates the process from a developer's point of view. Hence, a headless design of this JAR generation is hard to set-up.

5 Gradle for AS400 CSW

This section aims to define the Gradle model which is developed for the software development project under study.

The build logic implementation in this study uses the Gradle Native build methodology. As mentioned earlier, Gradle follows a build by convention approach. This means that Gradle comes with a pre-conceived notion of commonly used build functions. For example, if there is a requirement to create an executable from a set of C sources, Gradle fixes default directories to search for these sources and associated headers, and defines a control flow such as a compilation task followed by a link task, check, and install tasks. It also decides the location of placing the generated objects and executables. It is also possible to alter these defaults if required.

Gradle achieves this using a Rule based model configuration. This enables build authors to describe *what* they would like to do rather than *how* they would like to get it done.

5.1 Terminology

This section describes some common Gradle terminologies that are used and how they map to the project specific terminologies described in Section 4.

- **Component**: Gradle component represents a logical piece of software in build logic. It is an abstraction for a unit of code which can depend on other code units. For example, it can represent a library or an executable being created. Components are characterized by a specific name. In this study, the component names may map to the name of the Collection/Constituent being built.
- **Source Set**: This represents the set of source files which are grouped according to the language in which they are written. Gradle components generally use these source sets in the build. In this study, three types of source sets are used - C Source Set, Java Source Set and Assembler Source Set.
- **Tool Chain**: This represents the tool chain that is to be used for building the software. In embedded software development, there is cross development which is achieved by using different tool chains. In this study, there are two different tool chains which are used to produce different variants of the image - *sparc-rtems-gcc* tools for cross-compilation on a target processor and the native PC *gcc/g++* tool chain.
- **Platform**: Platform has two attributes - architecture and operating system on which the build is to be done. Each platform is also associated with a corresponding tool chain.
- **Binaries**: Binaries are the output of build logic. Gradle components representing appropriate source sets are compiled and linked accordingly to create binaries. In this study, three different types of binaries are generated by Gradle - Executables,

Libraries and Partially Linked Objects (PLOs). Linker and compiler arguments can be provided as attributes when defining binaries in the Gradle software model.

The terminologies which are explained can be illustrated with the help of an example in figure 15. Consider the *build.gradle* at the 'asw' Collection level. There are two components being described. Each of these is characterized by a C source set. The build file also defines the target platform on which the builds are to be executed. In addition, as C sources are involved, there is a need to mention the possible locations of header files which are referenced by the sources. This is done by using what are called as Gradle's Prebuilt Libraries. Since the header files are spread across multiple directories, it is possible to create groups with each group containing a set of relative path to headers. These groups are then added appropriately to the build files and a linkage is achieved for compile time operations.

```
1  // build.gradle for asw
2
3  apply plugin: 'c'
4  project.ext.SRC=["Asw.c"]
5  model {
6          components {
7                  libasw (NativeExecutableSpec){  //define the type of component to be built
8                          targetPlatform "SCOC3" //choose the tool chains defining the SCOC3
                               platform
9                          sources {    //define the source set used
10                                 c {
11                                         source {
12                                                 srcDirs "."
13                                                 include SRC
14                                                 }
15                 //define the location of header files required for compilation of sources
16                                         lib library: 'as400prodHeaders', linkage: 'api'
17                                 }
18                         }
19                 }
20                 asw (NativeLibrarySpec){
21                         targetPlatform "SCOC3"
22                         sources {
23                                 c {
24                                         source {
25                                                 srcDirs "."
26                                                 include SRC
27                                                 }
28                                         lib library: 'as400prodHeaders', linkage: 'api'
29                                 }
30                         }
31                 }
32         }
33 }
```

Figure 15: *build.gradle* at asw level

It should be noted here that in build files at collection level, not every information required for the build is defined explicitly. As most of the collections have a common behaviour, this is factored out and included in a separate build file at a higher level in the hierarchy. The next subsection describes about this adopted methodology in detail.

Gradle tool runs on a Java Virtual Machine (JVM) and the start-up time can be significantly long as it requires several supporting libraries. Gradle uses a daemon to perform these activities. It is a long-living background process which helps to increase the speed of the builds being executed. As a result of this, every build need not undergo a long bootstrap process. The daemon can be enabled or disabled through the build logic.

5.2 Build System Design Aspects

The figure 15 was at collections level. As mentioned, the build logic within a collection level is not comprehensive. These build files usually inherit from a common build file at the root level of the project. In Gradle, this is called as a Multi-Project Build [12].

Multi-Project Builds in Gradle

Section 4 provided a detailed overview of the organization of the source code hierarchy in the project. The software executable that is generated in the build process is build up from Constituents and Collections. There is a build file in each of the Collection, Constituent and Sub-Constituent levels. In Gradle's terminologies, these build files are called *sub-projects*. The sub-projects which share common behaviour are grouped in a separate build file. This file is called a *root project*. There exists also a *settings.gradle* file at the same level as the root project's build file. This file defines the sub-projects which should inherit the root project's functionality.

A good build system should have a global, fine-grained view of dependencies across all Collections and Constituents [15]. The Multi-Project build methodology that is adopted provides this feature. It is expressed in tree form in the figure 16. There is a root project build file (in red) and sub-project build files (in blue) within each Collection and Constituent.

The *build.gradle* depicted in figure 15 is a gradle sub-project. Some of the features used are expressed in the root project's build file. Excerpts of the build logic in the root project are shown in figures 17 and 18.

This configuration is for sub-projects which need to produce a partially linked object (PLO). A custom plugin called *PartiallyLinkedObjectRules* plugin is created. By default, Gradle provides support to build executables and static libraries by default. Since, the PLO generation need to be distinguished, a separate custom plugin is created. Two distinct platforms are defined and the tool chains are distinguished based on these platforms. In the sub-projects, the platform on which the build is carried out is mentioned.

Every new sub-project becomes a part of the Gradle build when the name of the project is included in the *settings.gradle* file under the root level directory. In this study, the root project can be divided into three distinct sections.

1. Configurations for a set of sub-projects at Collection/Constituent/Sub-Constituent level which creates PLOs or Static Libraries.

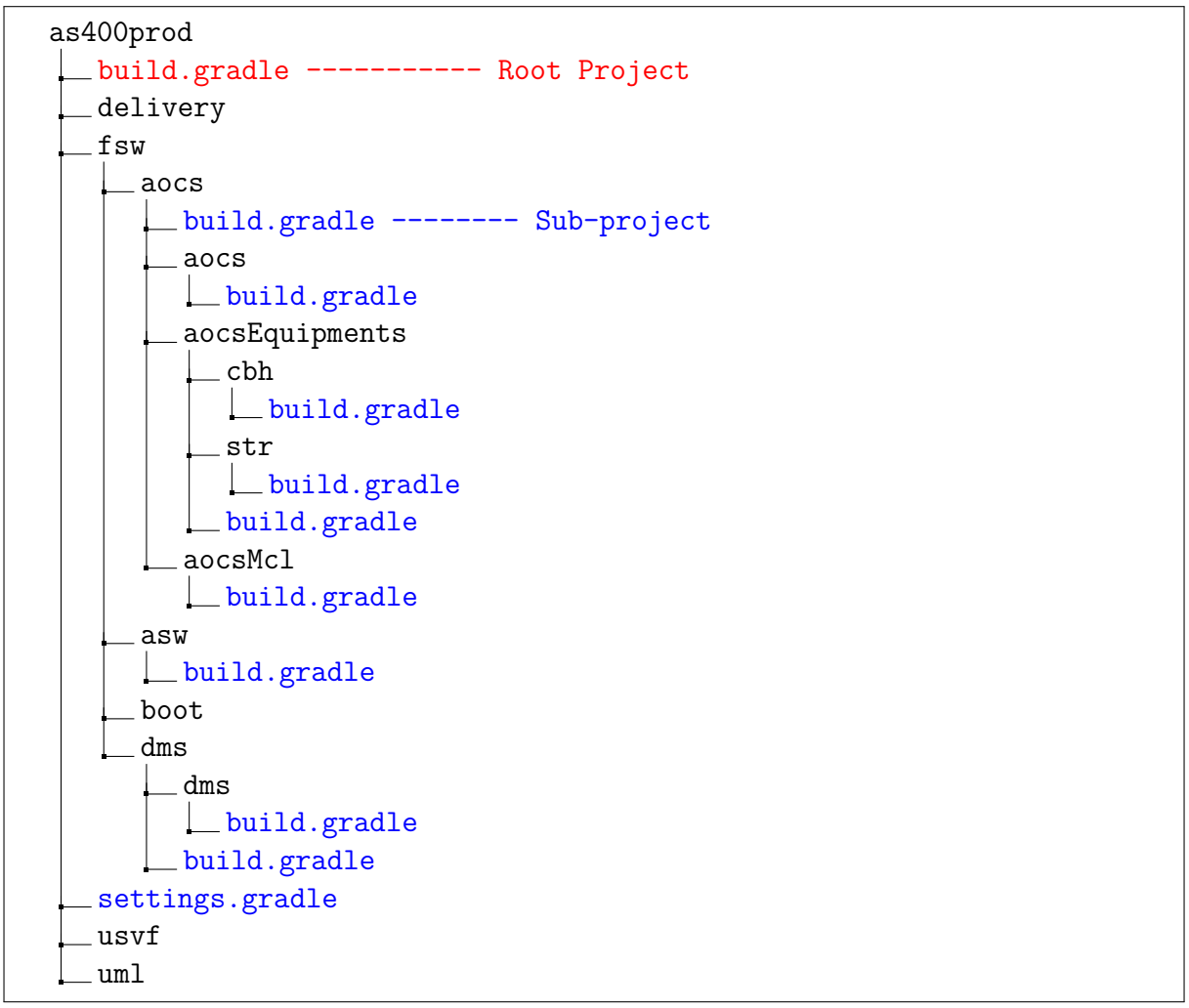

Figure 16: An excerpt of the AS400 Production Repository Structure containing *build.gradle* files

2. Configurations for handling unit tests.
3. And the third part which contains configuration for building the complete CSW and related custom tasks.

The two sets of sub-projects are filtered based on the names of the projects. They contain behaviour common for the sub-projects. Generally, the custom plugins which are created are applied specifically for these sub-projects. The design also incorporates custom Gradle tasks for performing various functions. There is a custom task for creating a Global Artifact version marker. The marker is of the form $PROJECT-$VERSION-GIT_HASH. The $PROJECT and $VERSION variables are set in a file called *gradle.properties* also present at the same level as the root project. The developer version controls this before the build, usually before a release image. The GIT_HASH variable is set by running Git commands to get the HEAD commit id as well as dirty information if present. There are

```
1  // build.gradle for as400prod — root project
2  //the following configuration is common for all sub—projects where there is a need to create
       either a partially linked object file or a Static library.
3  //filter based on name of project.
4  configure (subprojects.findAll {!it.name.contains('SVDD')}) {
5  //apply custom plugin because PLO generation not inherently defined by Gradle
6  apply plugin: PartiallyLinkedObjectRules
7
8  apply plugin: 'c'
9
10         model {
11
12 //define the tool chain used for cross compilation. Take care when providing the arguments if
       order of arguments matter
13                 toolChains {
14                         gcc(Gcc) {
15                                 path "/opt/rtems/4.6_20130612/bin"
16                                 target("SCOC3"){
17                                         assembler.executable = "sparc—rtems—gcc"
18                                         cCompiler.executable = "sparc—rtems—gcc"
19                                         linker.executable = "sparc—rtems—ld"
20                                         staticLibArchiver.executable = "sparc—rtems—ar"
21                                         assembler.withArguments { args ->
22                                                 Collections.replaceAll(args,"assembler","
                                                       assembler—with—cpp")
23                                         }
24                                         cCompiler.withArguments { args ->
25                                         }
26                                         linker.withArguments { args ->
27                                         }
28                                         staticLibArchiver.withArguments {
29                                         remove '—rcs'
30                                         add 0, 'crs'
31                                         }
32                                 }
```

Figure 17: *build.gradle* at as400prod level

custom check tasks which perform checks on the generated executable. Alias tasks have been set for creating the different variants of the image. A separate CI task is created for interface with the CI server. It contains all the tasks which need to be executed on the remote machine when initiated. The ordering of the tasks is also carefully defined using Gradle constructs such as *mustRunAfter* and *shouldRunAfter*.

Prior to the start of the design with Gradle, interviews were done with the developers of the system. The interview was primarily semi-structured. Features which were used often during the builds were recorded. The challenges faced by developers during build time was also recorded. The notes prepared contained text such as:

Developer 1(D1): When I want to run Make on my developed source code, I need to use a large number of low level command line arguments. . .
Developer 2(D2): I feel there is a lot of boiler plate code in the build logic. Often, I am forced to touch the same environmental variable in multiple build files.

```
33                                  path "/home/a27495454/test_build/scripts/bin"
34                                  target("PC"){
35                                          cCompiler.executable = "gcc"
36                                          cppCompiler.executable = "gcc"
37                                          linker.executable = "g++"
38                                          cCompiler.withArguments { args ->
39                                          }
40                                          cppCompiler.withArguments { args ->
41                                          }
42                                          linker.withArguments { args ->
43                                                  List origArgs = new ArrayList(args)
44                                                  def map_name = new File(System.getProperty("
                                                        user.dir")).name
45                                                  List preArgs = [ "-Wl,-Map,${map_name}.exe.
                                                        map,--cref","-Wall" ]
46                                                  args.clear()
47                                                  args.addAll(preArgs)
48                                                  args.addAll(origArgs)
49                                          }
50                                  }
51                          }
52                  }
53  //define the applicable platforms
54                  platforms {
55                          SCOC3{
56                                  architecture "sparc"
57                          }
58                          PC{
59                          //default OS and architecture detected by Gradle
60                          }
61                  }
```

Figure 18: *build.gradle* at as400prod level(contd.)

5.3 Analysis of Build Logic

This section describes how Gradle handles the build and unit test process. As mentioned earlier, Gradle performs the build by executing tasks. They form a Directed Acyclic Graph (DAG)[44]. Every Gradle build has three distinct build phases - Initialization, Configuration and Execution.

In the Initialization phase, Gradle determines the different sub-projects and the root project which would be participating in the build. During the Configuration phase, Gradle configures the build logic in all the projects. In the Execution phase, Gradle executes the actions described in selected tasks which are required for the invoked build. The *doFirst* and *doLast* closures in a task are always executed at this stage in the build life-cycle.

In this study, it is possible to build the complete software as well as small portions of the source code. Both these scenarios are explained with the help of examples. Consider the case where the Collection 'asw' needs to be built. From figure 15, it can be deduced that two different types of binaries will be handled in the build file - PLO and Static Library. The platform for which the compilation needs to be done is SCOC3. The tool chains which would be doing the compilation and linking are obtained by processing the root project. The different sources that need to be compiled are defined using the variable

reference $SRC. The required header files at compile time are gathered by passing the entries under the set 'as400prodHeaders' to the compiler. For the PLO generation, Gradle by default describes tasks such as compile, link, check and clean. For Static Library generation, Gradle defines the *CreateStaticLibrary* task. The default locations into which the PLO and *.a* files would be written to is inside the Gradle build directory which can be referenced using $buildDir. For PLOs, the custom plugin then ensures that the PLO is copied to an appropriate location within a folder at a level above the Collection. This folder contains two sub-folders which stores PLOs generated for the two image variants SYSDB and SWDB. For Collections which contain Constituents and Sub-Constituents, the process is more elaborate. The core build logic inside a Collection which contain Constituents is shown in figure 19.

```
tasks.withType(LinkExecutable) {
if (name == "linkLibaocsExecutable") {
        dependsOn << ":fsw/aocs/aocs:libaocsExecutable"
        dependsOn << ":fsw/aocs/aocsEquipments:libaocsEquipmentsExecutable"
        dependsOn << ":fsw/aocs/aocsApFw:libaocsApFwExecutable"
        dependsOn << ":fsw/aocs/aocsMcl:libaocsMclExecutable"
        dependsOn << ":fsw/aocs/aocsMclIf:libaocsMclIfExecutable"
        source = fileTree(dir: 'objs_exe/sysdb').files
}
else if (name == "linkLibaocs_swdbExecutable") {
        dependsOn << ":fsw/aocs/aocs:libaocs_swdbExecutable"
        dependsOn << ":fsw/aocs/aocsEquipments:libaocsEquipments_swdbExecutable"
        dependsOn << ":fsw/aocs/aocsApFw:libaocsApFw_swdbExecutable"
        dependsOn << ":fsw/aocs/aocsMcl:libaocsMcl_swdbExecutable"
        dependsOn << ":fsw/aocs/aocsMclIf:libaocsMclIf_swdbExecutable"
        source = fileTree(dir: 'objs_exe/swdb').files
}
}
```

Figure 19: Excerpt of *build.gradle* of a Collection containing Constituents

The default task which does the link in Gradle is called the *LinkExecutable* task. This default is for linking objects for creating an executable or a Shared Library. For a PLO, there is a need to hook into this task. To express dependencies before the partial linking stage, *dependsOn* construct is used. Conditional statements are used to distinguish the SYSDB image and the SWDB image. An attribute called *source* is provided which defines the object files which need to be provided as an input to the link stage. As mentioned, each Constituent writes its PLO to a folder at a higher level. This folder will be at the same level as the Collection. Hence the *source* attribute references all files in this folder through the relative path. Similar logic is replicated across all Collections and Constituents in the project tree.

The build process produces two executables of the Central Software (CSW) which are called SYSDB and SWDB images. These executables are created through linking the PLOs of the various Collections. The root project contains the logic to create these binary variants. Two dedicated Gradle components, CSW and CSWswdb, are created. For this build, no sources are defined. The tool chain, linker and compiler arguments are appropriately set. The *LinkExecutable* task of these components contain dependen-

cies instructing the construction of PLOs of the participating Collections in the build. Sources which were modified would be re-built, and the new PLOs would be copied to the appropriate folders. The files within these folders are then provided as an input to the linker for the final link stage. Custom check tasks as defined in the project execute after the binaries have been generated.

The execution of unit tests are slightly different from building the executable. The existing system defined a set of unit tests for selected Constituents and Sub-Constituents. Each of these unit tests is transformed into an executable and the test is run in a simulator. For each Constituent/Sub-Constituent, the cumulative results of multiple tests convey the status. Shell scripts are used to execute a list of tests in batch mode. A fundamentally different approach towards unit testing was then proposed. A unit testing framework using CppUTest [45] is created. Gradle build system is designed to support this unit testing framework. In this methodology, it is intended to have one executable per Constituent/Sub-Constituent and run a suite of tests against this executable in the simulator.

The CppUTest framework uses a unit test harness written in C++. A harness is a piece of software which describes how the production code is expected to behave during unit testing. The source code for CppUTest is version controlled in the production repository. The binaries for the respective operating systems are also version controlled. A *build.gradle* file of a Constituent consists of two types of sources - production C sources as well as test harness and test related C++ sources. An executable is generated based on the target platform and the defined tool chain. By default, executables are built for both SCOC3 and PC platforms. A custom task is provided to run the Unit Test. This involves invoking Gradle to perform the build followed by a shell script to launch the simulator. The custom tasks are present in the root project and is common for all unit tests across the project.

6 Continuous Integration in AS400 CSW

Best practices that help create an effective CI workflow was discussed in Section 4. During the design of a CI workflow for this project, these best practices were taken into consideration. The benefits of having a CI set-up for a software development project were also discussed. This section will provide a detailed description of the Jenkins CI workflow designed for the AS400 CSW development project.

6.1 Terminology

This section describes terminologies used in the CI workflow design. Jenkins is the open source automation server in use. The terms described in this section are purely within scope of the project under study.

- **Job**: A Jenkins job is user-configured definition of work which Jenkins should execute. Examples are checking out a Git repository or building a piece of software. The term job is synonymous with the term Jenkins project.
- **Phase**: A set of Jenkins jobs which perform similar functions are grouped to obtain a Phase. For example, jobs checking out selected repositories onto a workspace in the server can be grouped and called as Checkout phase.
- **Job Chain**: A job which groups job phases to carry out a Jenkins build. It defines the control flow through the different phases. A mapping of activities from the production domain to the CI domain can be achieved by defining these activities as part of a job chain.
- **Node**: A machine on which Jenkins jobs execute. In this study, there are three nodes. One master node where the Jenkins instance runs. The master provides the user interface to communicate with Jenkins. It also provides configuration for the slaves as well as security settings. Two slave nodes on which the build, test and static code analysis runs. The generated artifacts are copied back to the master node where they are archived. Slaves nodes are also referred to as build nodes.
- **Executor**: Slots in which Jenkins jobs are executed. A node can contain zero or more executors. In this study, each of the slave nodes are equipped with four executors. At any point of time, a maximum of one job can run per executor.
- **Check Window**: A condition which is checked at the end of a phase. It helps determine the control flow of a job chain.

Figure 20 shows an overview of the Jenkins set-up that is used in this project. The Jenkins master instance allocates jobs to the executors on the slave nodes. Jenkins contains default algorithms to map the jobs to the executors. In this study, both the slave nodes run linux operating system. Project specific restrictions on this allocation are discussed in the next section.

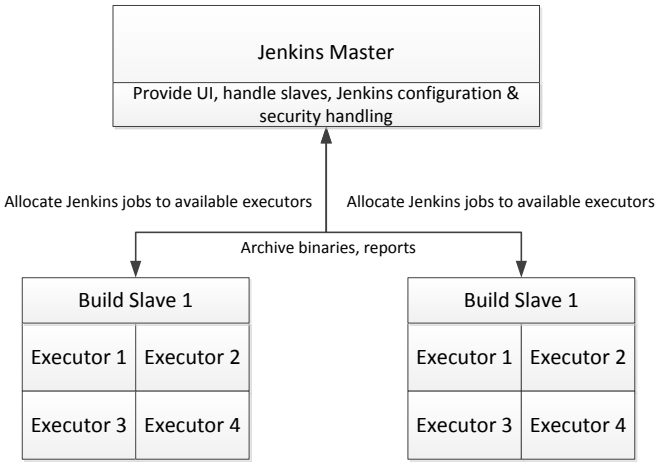

Figure 20: Jenkins Master Slave set-up

6.2 CI Workflow for AS400 CSW

This section explains the design of the CI workflow in Jenkins. The design is explained considering the fact that the Jenkins master and slave nodes were already set-up with the required plugins, configurations and security restrictions of the project. Further discussion of these parameters are beyond the scope of this thesis.

The flow chart in figure 21 provides an overview of how Jenkins integrates with the Atlassian tools used in software development. There is a possibility to set up a web hook to Jenkins from Stash. When a commit is pushed to a branch being tracked by Jenkins by any of the developers, a hook is sent to Jenkins CI server to initiate the build. After Jenkins finishes the build, the status of the jobs are notified back to Stash. Hence, the framework uses Stash as the central user interface for repository management as well as to view status of Jenkins jobs. A web-link to Jenkins GUI is also provided for each job for each commit in Stash. Using this link, it is possible for developers to navigate to Jenkins environment and view finer details of the build. The artifacts are stored automatically on the master node and is visible to all members of the project.

Jenkins jobs are configured in two ways. Either through the Jenkins GUI or using XML. The XMLs describe the configuration of the jobs. There exists Jenkins Command Line Interface (CLI) scripts which are invoked to push these XMLs into the server. They can then be viewed on the Jenkins consoles and modified if necessary. There are three different types of Jenkins jobs which can be set-up.

1. **Free-style job**: A flexible configuration which is used to perform a defined task. They have the limitation that they can run on one of the slave machines.
2. **Matrix Job**: A job which is capable of executing in few or all of the defined build slaves concurrently. For example, it is necessary to clone the repositories on Jenkins workspaces in all the build slaves. This can be achieved by defining the job as a matrix job.

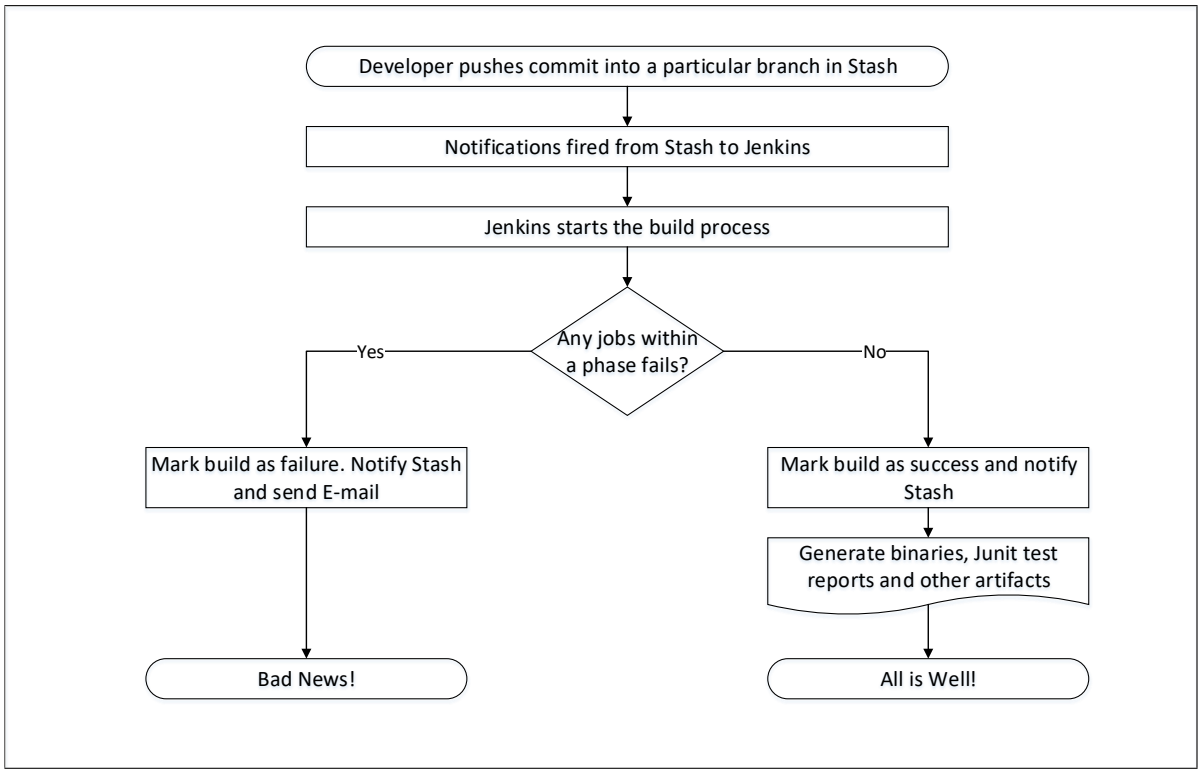

Figure 21: Overview of Stash - Jenkins connection

3. **Multi-configuration Job**: A job which can be used to define a CI workflow by defining the phases and the jobs within these phases as well as the check windows. A classical example is the job chains which is discussed next.

The CI workflow is derived from the Gitflow workflow. The Gitflow workflow was discussed in detail in Section 4. There are seven Jenkins job chains which were defined for the project. They are:

1. Master Job Chain

   This job chain defines the control flow for the CI run when a commit is made into the *master* branch in the production repository.

2. Develop Job Chain

   This job chain defines the control flow for the CI run when a commit is made into the *develop* or *release* branch of the production repository.

3. Feature Job Chain

   This job chain defines the control flow for the CI run when a commit is made into the *feature* or *bugfix* branch of the production repository.

4. On-Demand Job Chain

   This is a special job chain which is created to handle Ad-Hoc requests. It provides

the developers with a flexibility to configure the branch which needs to be tracked by Jenkins. A manual trigger of this job chain is also possible from the Jenkins GUI.

5. Check Master Job Chain

   This job chain handles the static code analysis when a commit is made into the *master* branch in the production repository.

6. Check Develop Job Chain

   This job chain handles the static code analysis when a commit is made into the *develop* or *release* branch in the production repository.

7. Check Feature Job Chain

   This job chain handles the static code analysis when a commit is made into the *feature* or *bugfix* branch in the production repository.

This classification was necessary because the branches are handled differently in the production domain. *master* branch is expected to be stable at all points of time and meet high quality levels. The *develop* branch is also expected to clear high levels of quality standards and a build error of a commit on this branch is to handled quickly. The *feature* branches have relatively low quality standards set and hence their control flow is less strict as compared to *develop* or *master*. Each of these job chains have a dedicated workspace on all the slave nodes. The path of this workspace is propagated to all the jobs defined within the chain through a Jenkins global environmental variable called $WORKSPACE.

The figure 22 shows an overview of the structure of a job chain containing phases and each phase containing a set of jobs. To summarize the overall structure, a job chain can consist of one or more phases. Each phase can contain one or more jobs. By default, the jobs within a phase execute concurrently on the same machine. Matrix jobs can execute concurrently on the defined slave nodes concurrently. Concurrent job execution within a phase can limited by using build throttling. Jenkins global environmental variables are passed from the job chain to child jobs.

A detailed flow diagram of the Jenkins CI workflow is portrayed in figure 23. It takes into consideration three job chains (Master, Develop and Feature). Generally, every job chain has a common set of phases.

- **Checkout Phase**: This phase comprises of jobs for checking out the correct version of the different repositories that are needed for the build. The jobs are of matrix type as they need to be executed in all the build slaves. Jobs within this phase can execute concurrently to keep the builds fast. The check window at the end of this phase causes the CI run to quit because Jenkins was unable to correctly check out the required repositories. Often, this might happen if Jenkins does not have correct permissions, or if the slave nodes are running out of memory, or if the commit into git submodules are not checked in correctly.
- **Make Extra Libraries Phase**: During the build, there is a need to reference

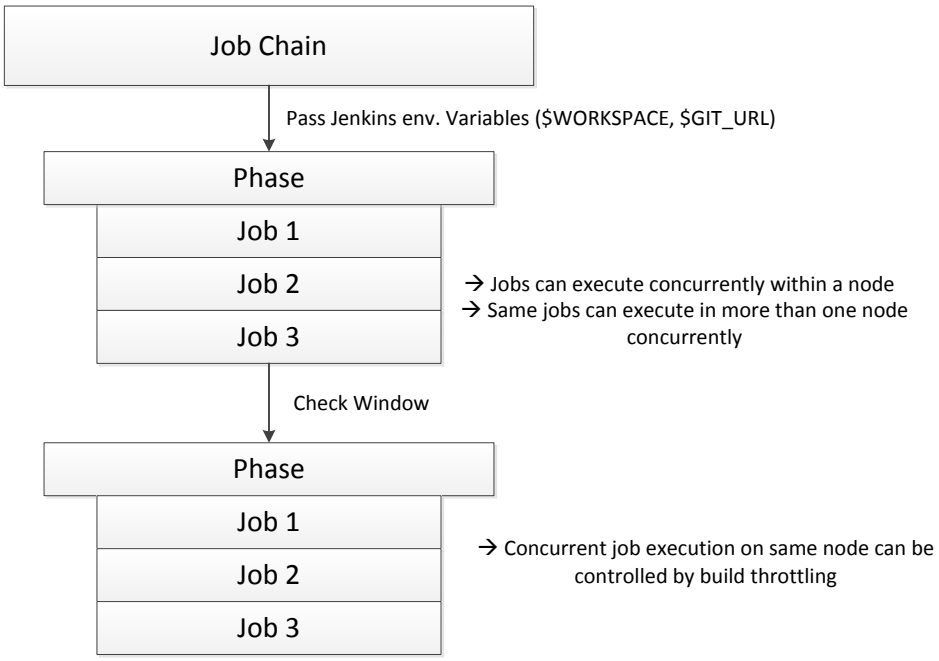

Figure 22: Overview of the structure of a Job chain containing phases and jobs

some libraries which can be compiled before hand. The job in this phase carries out this activity. It is needed only if Make is the build system. In case of Gradle, this phase is redundant because the build logic treats this within the root project.

- **Unit Testing Phase**: Unit Testing is carried on several Applications in the production repository. One job per Application is assigned. The obtained results are displayed as JUnit test reports. The tests are done only by compilation against the SCOC3 platform. A separate phase need to be created if the test results need to be obtained for cross compilation against native PC platform. If Gradle is the build tool, then the same phase can be used to obtain both sets of results.
- **Build Phase**: This phase contains two jobs when Make is the build system in use. One job for creating the SYSDB variant of the image and another for SWDB variant. When interfaced with Gradle, one job will cover creation of both these variants.
- **Workspace Clean-up**: The build directories are cleaned before any build is invoked. However, to capture and correct sporadic errors due to repeated cloning in the same workspace, a dedicated job is provided for this handling.

In addition to these phases, jobs for other functionalities are provided. Additional Checks is a job which runs a set of shell scripts on the generated executables. The check merge conflicts job detects merge conflicts by attempting to merge the feature branch with develop. It does not perform the actual merge but a dry run is carried out. The check windows play an important role in determining how the job chains are handled. The

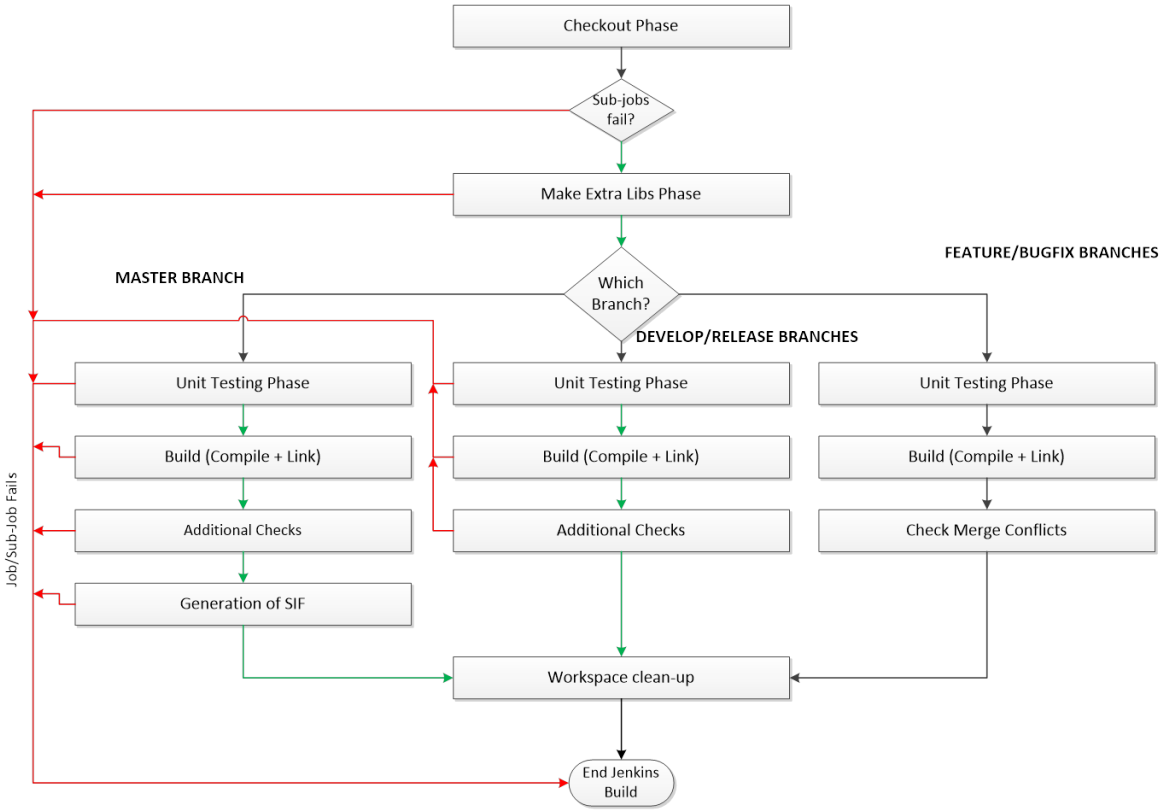

Figure 23: A detailed Jenkins workflow for the project under study

red lines in the workflow lead to stopping the CI run because the quality standards have not been met and further continuation of the run might result in further similar errors. The green lines indicate that the workflow has no errors and should be in a position to continue till the end. The black lines signify that the CI run proceeds irrespective of errors encountered. This is primarily important for feature job chains where developers often expect quick and comprehensive feedback.

The existing set-up uses Logiscope Rulechecker tool for carrying out Static Code Analysis. By default, the code analysis runs on the entire source code in the repository and takes significantly longer time to execute than the build and tests. Hence, this activity is set-up as a separate job chain so that it does not act as a blocking parameter in terms of performance for other activities.

6.3 Challenges faced during design

Developing a CI workflow for an embedded project is a challenging activity. This section describes some challenges that were faced at the time of design and some actions which were taken to mitigate them.

Setting up of a CI server and build slaves requires additional powerful hardware. The

Jenkins master instance runs on the same server which handles the Atlassian tools. Hence, through Atlassian Crowd, the access to Jenkins can be managed. In this project, the slave nodes are linux flavoured virtual machines. The software development activities on the client SDE is predominantly in windows/cygwin. A shift to a linux operating system brings about a need to make the build and glue logic compatible. Significant effort was put into this activity. However, over a period of time, the automation mechanism in powerful hardware would prove to be more advantageous[7].

Jobs within a phase are able to run concurrently on the same machine. In the scope of this project, two builds when executing on the same machine might interfere with each other's process flow leading to non-deterministic results. This is a common occurrence when builds are carried out using GNU Make. Hence, the concurrent job execution within a phase needs to be controlled. This is done by using a build throttling mechanism in the CI server. Jobs which are not allowed to run concurrently on the same machine are grouped under a single label. Jenkins master instance is instructed to read this dependency before allocation of jobs to the executors on the build slaves. In this manner, concurrent runs of non-compatible jobs on the same machine is avoided. In the realm of Validation testing, there is a project specific limitation that not more than one test is allowed to run in a machine at the same time. A label for validation tests is also generated and the issue is handled in a similar way.

In the space avionics domain, the software development is often large and complex. Hence, CI workflows designed for a project is expected to be both effective and efficient. An effective CI workflow is one which is adequate to perform the required operation. An efficient workflow is one where the task is carried out not just in an effective manner but with significant speed and less effort. In other words, the workflow should handle *scalability*. As mentioned in the earlier section, there are large number of jobs in each job chain. Managing such large numbers would involve a considerable amount of effort and time. To mitigate this challenge, a python based tool called Jenkins Job Builder was investigated. Through the use of this tool, there is a possibility to define the configurations for the jobs once in the form of YAML. The tool is capable of parsing the YAML files and converting them to XML format. This can later be pushed to the Jenkins server. Jobs which share common behaviour, such as jobs belonging to an arbitrary phase, can be handled in an effective manner through the use of a well defined template system. The configuration information of the jobs are spread across multiple YAML files which are then processed by the tool to generate the XML.

Also, in the space avionics embedded software development, there is a large scale re-use of developed software products. So, a CI workflow designed at the time of development of one project should be easily adaptable for child projects which are forked from this one. The job chains for the child can be set-up using the Jenkins Job Builder tool as mentioned above. However, if the same Jenkins master instance have to be re-used, there is a need to manage the security configurations in case the parent and child projects have different access levels. Jenkins provides options for mitigating this problem by providing a project matrix authorization access to the Jenkins jobs. There are two levels of access

control. A global control which manages the access of the entire Jenkins master instance as a whole. The second is a job level control which manages access of the selected job to only a subset of users.

The artifacts which are created on the slave nodes are copied back to the master and archived. Since each job chain consists of a large number of jobs, the number of artifacts created by each CI run is significantly large. The memory requirements for storage of these artifacts increases over time. In this project, an upper bound of the number of artifacts to preserve for the check feature and build feature branches was fixed. This means that the artifacts for only a fixed number of commits can be accessed through the Jenkins GUI. CI runs older than the oldest entry in this list are automatically deleted. The upper bound was determined by considering historical data such as the average number of active *feature* branches at any point of time and the frequency of integration of these branches to *develop*.

Increasing the number of build slaves will translate to faster build and feedback times. However, a large number of tools are installed in the build slaves. Examples include the cross compiler, linker, the GNU Make package, python package for Job Builder tool. Any changes made to any of these tools need to be applied homogeneously to all the slave nodes. Considering the number of slave nodes and the magnitude of the change being made, the effort involved would be significant in terms of time and complexity.

7 Evaluation and Results

One objective of this study was to determine the factors influencing the selection of a build system for the software development project. In this regard, using both qualitative and quantitative data, some factors were formulated and analyzed. This section presents the evaluation of the build system and the results that were concluded from the evaluations.

This study outlines three *evaluation blocks* for analysis. The first is performance. This means the time required for the build or unit test process to execute. Second is the maintenance complexity of build logic. As the term complexity can have different contextual significances, later sections handle this in detail. Finally, the features offered by the build systems in the current state are discussed. These evaluation blocks are not completely mutually exclusive and quite often there are dependencies between these blocks.

A comprehensive migration of the build system from Make to Gradle has not been achieved as part of this study. The evaluation is only based on features which are present in both the build systems. Hence, all the collected data and formulated results are normalized with respect to this aspect.

The hardware that was used to run these tests were essentially the same. The build slaves on which Jenkins jobs run were chosen as the evaluation machines. For the tests, the SDE was maintained in a homogeneous state. The versions of all tools and software packages in the machines were identical. Each of the evaluation blocks are discussed in detail.

7.1 Performance Comparison

Performance is an important quality which needs to be analyzed when considering build systems for a software development project. By performance, we mean the time required for the build tool to execute a specific Make target or Gradle task. A large number of use cases in the existing project are studied. The Make targets and Gradle tasks under analysis are comparable in terms of the functionality it achieves. The commands are invoked multiple times, the outliers are removed and the average of the obtained values are taken into consideration.

To compare performance between the build systems, actual real life use cases are used. In the project under study, there are a large number of build tasks which are done on a day-to-day basis. A most common task is a *full image build*. In our context, a full image build means building the two image variants (SYSDB and SWDB) from scratch. As mentioned, Gradle runs in a JVM and requires a significantly long time to bootstrap. This is captured during performance measurement tests and is called as *first time build*. More often than not, developers generally work on a particular feature in the project and hence would require to build a small portion of the project rather than the whole. We

call this a *specific build*. Another type of task taken into consideration is the *incremental build*. In incremental builds, a build is generally not started from scratch. After a first build, the subsequent builds of the same task is called an incremental build. If there are no changes to factors affecting a build, then it is an *up-to-date* check.

The figure 24 represents three types of tasks and their results. The Make clean target runs faster than the Gradle clean task. When comparisons are made for a specific build, Gradle is faster than Make. The first time build on a machine using Gradle is on an average 45% slower as compared to Make. This can be explained by taking into the fact that the bootstrap period for a Gradle daemon brings about this delay. The incremental build times also showed that Make builds were slightly faster than those of Gradle. The interpretation of these results can be traced back to the build logic. Gradle has three phases of operation in the build life-cycle. When profile reports are generated for Gradle builds, it can be observed that a significant amount of time is spent during the configuration phase where Gradle attempts to configure all the projects defined in the build logic. In addition, the custom developed *PartiallyLinkedObjectRules* plugin has been applied to the Gradle task in consideration in this analysis. The plugin has a *doFirst* and *doLast* closure which effectively eliminates the implementation of the UP-TO-DATE feature offered natively by Gradle. These reasons can be attributed to the spike in the execution times of Gradle builds.

The next set of measurements also compare a specific build for Make and Gradle. However, in this case, the Gradle task that is chosen does not have any custom plugins containing activities in the execution phase applied to it. We infer from this set-up that specific builds in Make are still faster than Gradle. However, the up-to-date checks in Gradle are better than the use case where custom plugins were applied. This study is further extended to include Gradle's *--configure-on-demand* option of running builds. This prevents Gradle from configuring all projects which is the default behaviour and it configures only those relevant for the task to execute. As expected, savings in build times was recorded as shown in figure 25. The option N represents build of a task containing only native Gradle configurations and a C represents the build with configuration on demand enabled.

The results in figure 25 were analyzed further. A profile report for the Gradle build is obtained. This provides a split-up of the amount of time spent in each of the phases in a Gradle build. The build times in Gradle are spread across primarily five different categories.

1. **Startup**: There is a startup phase where Gradle determines which projects are going to take part in the build.
2. **Settings**: The startup is followed by a phase where Gradle reads the *settings.gradle* file.
3. **Loading Projects**: It loads the required projects for the build.
4. **Configuring Projects**: Gradle then performs activities in the configuration phase.

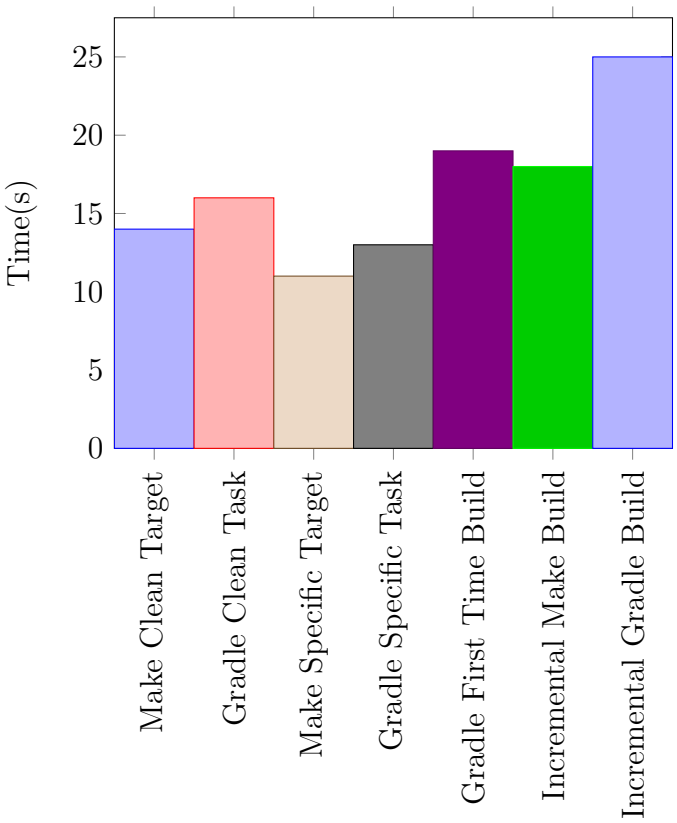

Figure 24: Performance comparisons between Gradle and Make for clean task, specific target and incremental builds

5. **Task Execution**: Finally the tasks are executed.

The first profile report in our analysis is that of a Gradle build without the daemon. This run takes the longest time as there is no optimization applied on the build. The results of the profile report is shown in figure 26.

The first of the optimizations under study is that of using the Gradle daemon. The result is shown in figure. The results of the same specific build with the daemon is shown in figure 27. For a specific build, a reduction of even a few seconds is a good measure. This is because a large number of specific builds go into making the final software and reduction in each of these build times contribute to overall reduction in build time.

Builds executed with *--configure-on-demand* option run faster. An analysis of the profile report shows that the time spent in configuring projects is lesser. This is because only those projects which are required for the build to run are executed. The analysis results are shown in figure 28

The specific builds with parallel execution provided further savings on build times. The pre-requisite to have reliable parallel builds in Gradle is to ensure that the projects are decoupled. This means that one project does not inject configuration on another project. In the scope of this study, efforts were taken to ensure that the projects were decoupled.

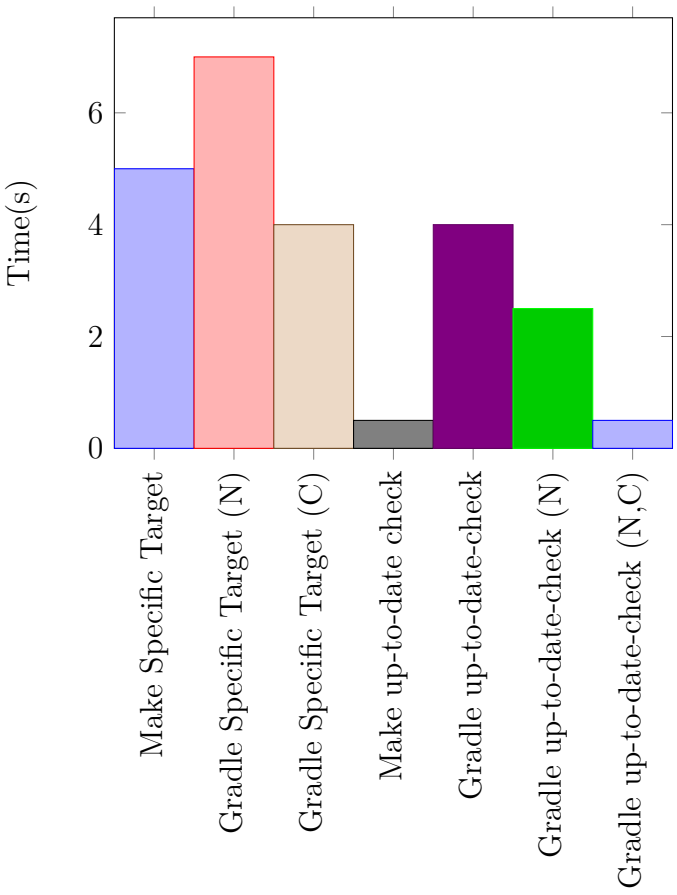

Figure 25: Specific builds without custom plugins applied to the project (N) and enabling configuration on demand (C)

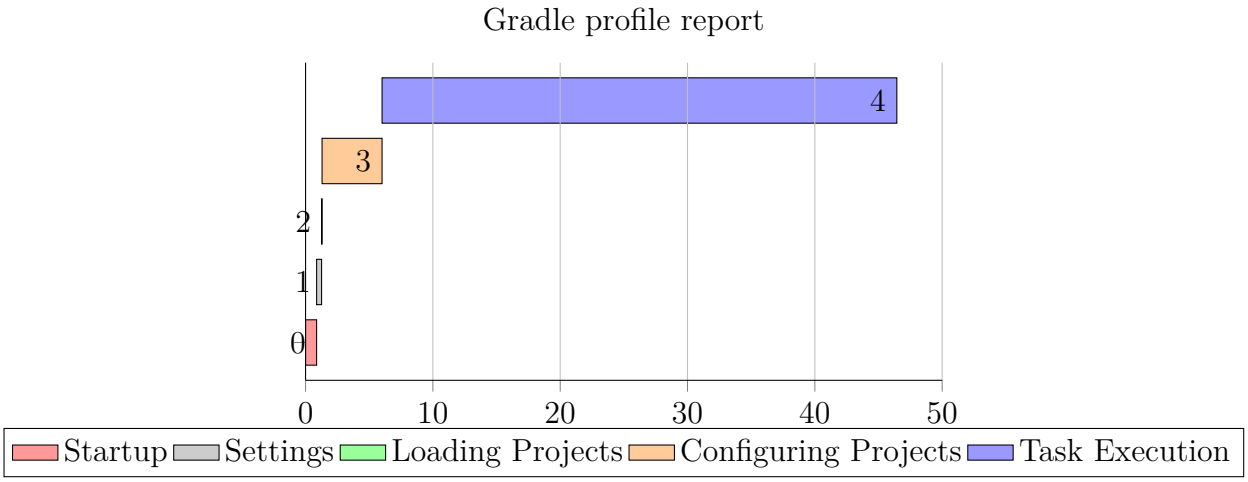

Figure 26: Gradle profiling results for a specific build

The root project's attributes are inherited in the sub-project but the Gradle manual

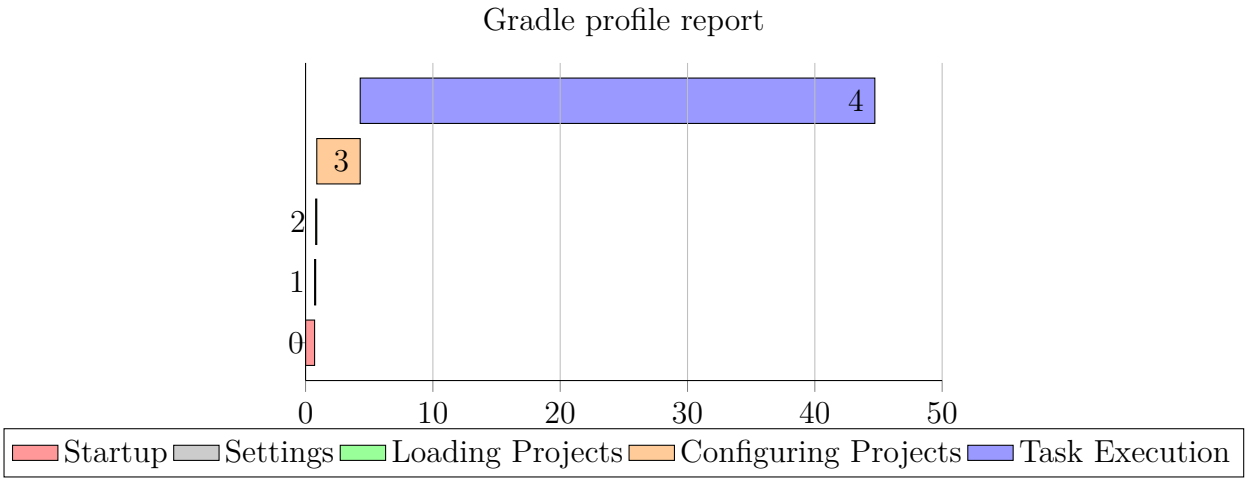

Figure 27: Gradle profiling results for a specific build with the Gradle daemon running

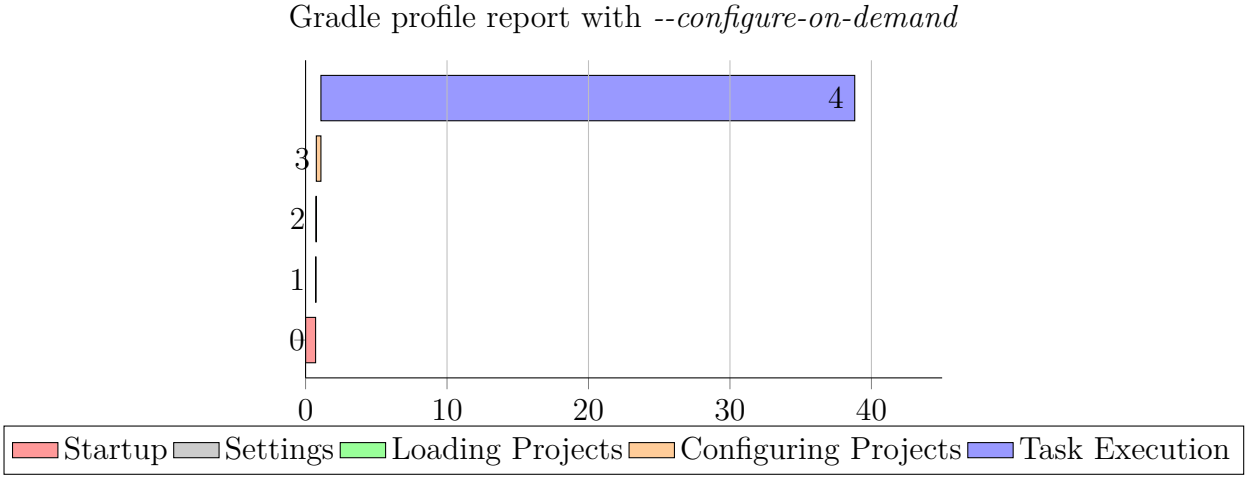

Figure 28: Gradle profiling results for a specific build enabling --configure-on-demand

says that this form of injection does not necessarily lead to coupling. The results of the parallel execution is shown in figure 29. However, careful analysis of parallel Gradle builds need to be carried out to ensure reliability.

The final performance tests were done for a full image build. The results are shown in figure 30. The full image builds with Make had to be done in two steps. The first build for building the SYSDB variant and then the SWDB variant. For Gradle, the design methodology allows a single command to build both variants. In Gradle, tests are done for three different build styles. These consist of builds without the Gradle daemon, builds with the daemon and parallel build. The Make build was carried out using the *-j 4* command line option. A summary of all the profile reports and Make's execution times have been mapped and shown in figure 31.

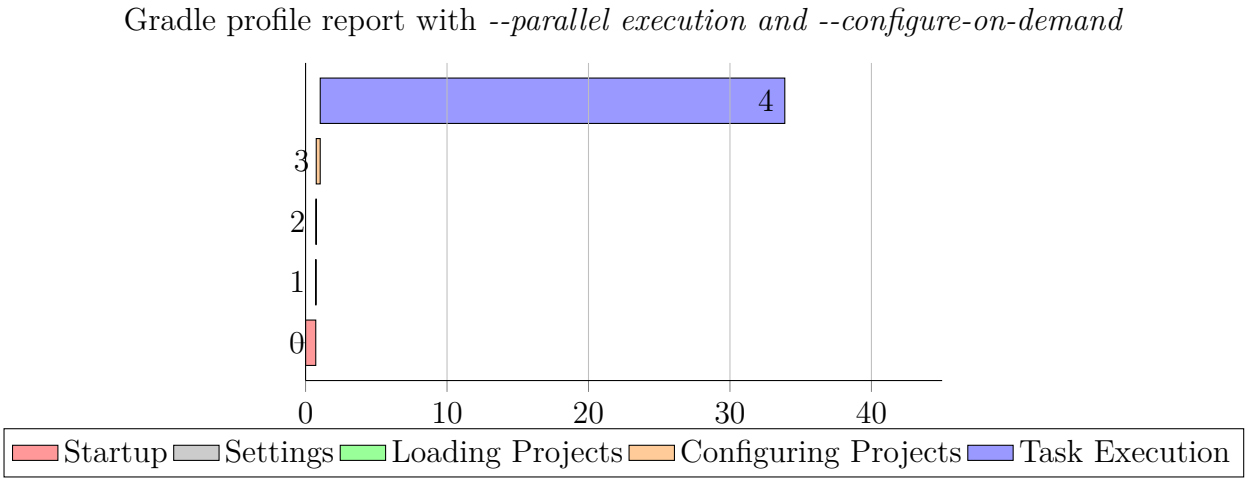

Figure 29: Gradle profiling results for a specific build enabling --configure-on-demand and --parallel execution

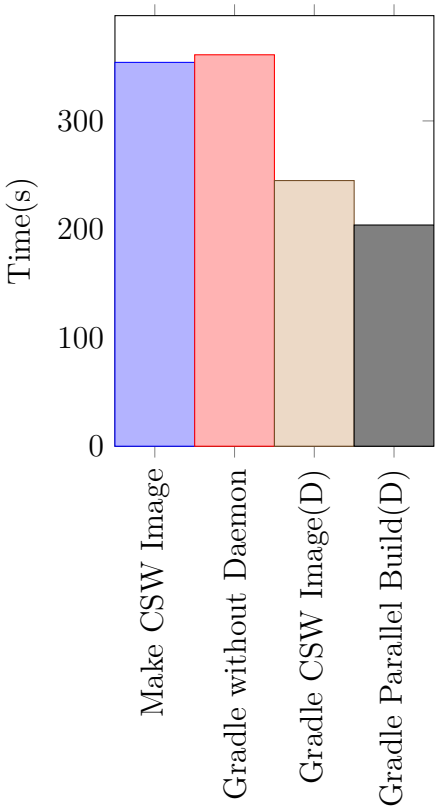

Figure 30: Comparison of full image builds on Make, Gradle without daemon and with daemon(D), as well as parallel Gradle builds

Figure 31: Summary of all profile reports

7.2 Complexity of Build Logic

Maintenance complexity of build files is an indicator for quantifying build systems. This type of complexity is a measure for the overhead that arises with maintenance of build logic. Software code complexity metrics such as Halstead's Complexity measures [46] and McCabe's Cyclomatic Complexity [47] do not provide a good method for measuring maintenance complexity of build logic.

For measuring complexity of Makefiles, literature suggests a theory based on calculating the number of occurrences of indirection in the logic [48]. An indirection requires a developer or build author working with the build script to trace a reference in the build logic to another location in the same build script or another build script. The number of occurrences of indirection are measured and their sum provides a measure of complexity of the logic. In Make based systems, indirections can arise because of using some of Make's features in the build logic. In our study, we consider the following parameters in the study of complexity.

- Number of build files in the project.
- Lines of code of build logic.
- Variable references in the build logic.
- Number of includes of other Makefiles.
- Employment of conditional statements in the build logic.

These parameters are measured for the AS400 project and the results are summarized in table 1. As the build system employs a recursive make build, Makefiles are present in each Collection/Constituent containing sources. However, the build system within the

production repository is not self contained. There exists a separate SDE repository as shown in figure 7. This repository contains a Makefile called *Make.rules* which defines the common build logic for the project. This Makefile is included in all the Makefiles in the production repository. In addition *Make.rules* includes other Makefiles in its build logic. The existing system uses a large number of reference variables in the design. Furthermore, these reference variables are used several times in the build logic. A large number of conditional statements are encountered in the Make based build system. This is primarily because the build author has designed the logic to derive control flows from low-level command line arguments that are provided at the time of invoking Make for the build. Each of these features contribute to indirection, thereby increasing the complexity of build logic.

Table 1: Tabular with parameters for complexity measurement in Makefiles

GNU Make	
Complexity Parameters	
Number of Makefiles	137
Lines of Code of common build logic	1754
Average Lines of Code per Collection/Constituent build script	25
Number of Variable References	61
Includes of other Makefiles	6
Conditionals (ifeq)	35
Conditionals (ifdef)	14

A similar exercise is carried out in the design involving Gradle as the build tool. The results are summarized in table 2. The number of build files are the same in case of both the build systems. However, the build logic implemented using Gradle for the project is self contained within the production repository. From the Gradle Multi-project structure described earlier, it can be inferred that there are no include statements spread across all the build files. There is a Gradle settings file which describes the sub-projects participating in the build. The number of variable references are lower by 83% and the number of occurrences of these references across the build logic are limited. This is attributed primarily to the declarative nature of the build system which requires less effort required from a build author's point of view to define rudimentary details in the build.

From the data aggregated in tables 1 and 2, it can also be observed that the average lines of code in a build script at Collection level is more in Gradle as compared to Make. This is because each build script at these levels define the different binaries that need to be created for the corresponding Collection. Hence, there exists boiler plate code within these build scripts increasing the average lines of code count.

Table 2: Tabular with parameters for complexity measurement in *build.gradle*

Gradle	
Complexity Parameters	
Number of *build.gradle*	137
Lines of Code of common build logic	648
Average Lines of Code per Collection/Constituent build script	37
Number of Variable References	10
Conditionals (if..else)	12

7.3 Feature Comparison

This evaluation block compares the features supported by the Make based and Gradle based build systems in this project. The objective behind this analysis is to determine the value provided by the build systems for an embedded software development project. This section aims to distinguish the build systems by the features which are supported by them within the scope of the project under study.

1. **Build by Convention**

   One of Gradle's strongest suite is the build by convention approach towards build. The build tool possesses knowledge of commonly used build operations and helps the build author in constructing control flows with minimal configuration effort. At the same time, the defaults defined by Gradle are flexible which makes it a good candidate for use in embedded software development. In Make based builds, the build tool is not declarative in nature. The differences in this aspect between the build tools originate from how the build tools are designed. A declarative build paradigm leads to expressive but concise build logics with lesser lines of code.

2. **Integration with CI tools and other build tools**

   This feature is important for projects which have adopted CI for their development. The differences in how the two build systems perform in this analysis can be explained with the help of a real life example.

   The AS400 software development project is aimed at creating a generic product. This means that at some point in development, other projects will be initiated whose architecture and source codes are based on AS400. With Make as automation tool, the interface between the build tool and CI tools was accomplished primarily by using shell scripts. The logic behind implementation of this interface is called the glue logic. When CI server needs to be set-up for the new project, the glue logic needs to me modified accordingly. The effort for this activity depends on the nature of the glue logic that is developed. At the time of design of glue logic for this project, hard coded absolute paths and environmental variables were introduced in the system. This caused the logic to fail on the newly created project. Hence, the workflow suffers from scalability issue.

With Gradle, a dedicated CI task is defined. This task is a container for other tasks which perform the build or test. In the CI realm, different activities such as build, unit tests and validation tests are mapped to corresponding jobs. In the build realms, a mapping is achieved through the set-up of different CI tasks. As Jenkins contains support for running Gradle tasks without any interfacing logic, it was possible for Jenkins to run the corresponding CI tasks. Hence, the CI workflow with Gradle proved to be effective.

In the validation environment of this project, Apache Ant is used for build of Java sources. Gradle integrates with Ant and provides a mechanism to execute Ant tasks from within Gradle. This has been demonstrated for the project and the obtained results were validated by comparison with Ant build results.

3. **Differences in Design of Build Logic**

   Developers uses build tools on a daily basis. Qualitative data was obtained from developers on the different aspects of the build logic which is used, the frequency of the usage as well as the challenges faced. This data was taken into consideration during the design with Gradle. The observations were recorded.

   - The Make builds had a number of command line arguments which needed to be used to obtain the correct build results. An example of a build command is as follows:

     ```
     $ make -j 4 CPU=SCOC3 DATABASE_DIR=swdb allLibs ram
     $ make -j 4 CPU=SCOC3 DATABASE_DIR= allLibs ram
     ```

     These low-level command line arguments arise because the build author creates a logic which defines different control flows to the build based on input from a developer. In our design with Gradle, this approach was discouraged. Alternatives such as creating custom Gradle tasks, separate build directories or Gradle's native control flow design concepts were used.

   - Software projects in C present challenges of including the correct header files at the time of compilation. The build system needs to track the location of headers to trigger re-compilations when required. The existing system using Make defined all possible header file locations through one environment variable. This was later passed on as an argument to the compiler. This approach fails if there are header files with the same name at two different locations. To tackle this in a different way, Gradle's design methodology defines distinct sets of header families. Each family contains locations of header files which will be required for building the various Collections. This results in a more fine-grained approach of handling headers. It also reduces boiler plate code as new header file locations need to be included in only one place in the build logic.

4. **Additional Features**

   To execute Gradle scripts, it is not required to have Gradle installed in the machine.

Instead, it is possible to use Gradle wrappers. The build logic defines a wrapper task which contains the Gradle version to be used for the build. At the time of execution of the build with the wrapper, it will download the corresponding version of the Gradle tool in to the machine and executes the build. This is different from Make where it is required to have the correct version of GNU Make installed in the machines and the *$PATH* variable contains the path to the tool.

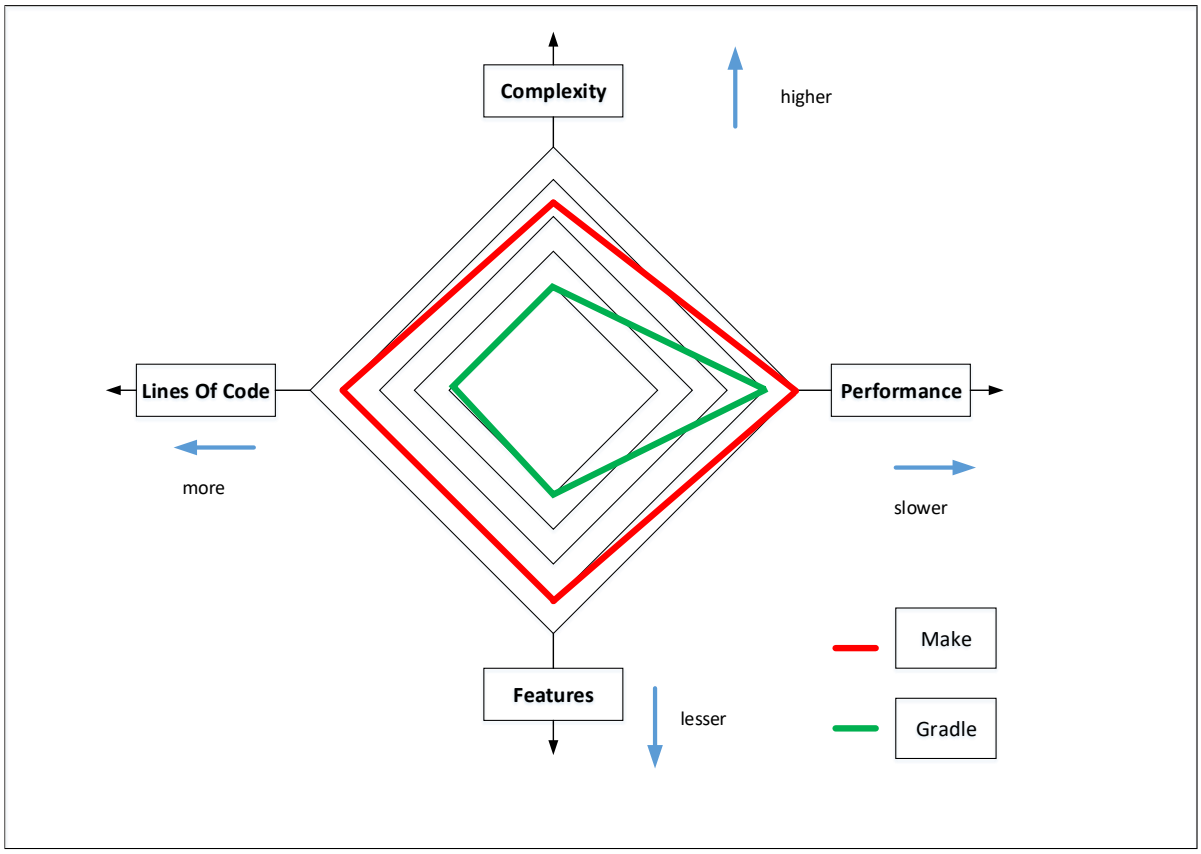

Figure 32: Summary of results - Comparison of evaluation blocks

Build scans help provide insights into build related problems at the time of development. It creates a shareable platform for collaboration between developers of a project to track the status of builds that happened locally on a developers machine. This feature is currently not supported by GNU Make.

The existing build system does not version control the specifications of the compiler and linker that are used to carry out the build. This limitation is handled in Gradle through the use of tool chain definitions within the build scripts. The build scripts are version controlled with the source thereby maintaining the specifications of the SDE tools throughout the life-cycle of the development project.

7.4 Summary

This section presented an overview of the results obtained on evaluating the two build systems under study. Three evaluation blocks were studied. They include performance of the build logic in real life use cases, maintenance complexity of build logic and a comparison of the features provided by the build systems. A summary of the results obtained after evaluation is shown in figure 32. The number of lines of code in Gradle build logic is 61.1% less than in Make. Similarly in terms of comparison of maintenance complexity of build logic, taking into account the number of occurrences of indirection, it can be concluded that the complexity is 51.3% lower in Gradle as compared to Make. In terms of performance analysis, the nature of results obtained are mixed. In certain scenarios, Make scores over Gradle and vice-versa in others. A comparison of the features provided by the build system infers 60% more number of features provided by Gradle over Make.

The functionality provided by the Make system is more comprehensive than of Gradle. However, for the formulation of the results in this study, the functionalities considered in the analysis are present in both the systems. From the obtained results, it can be concluded that Gradle could potentially be a replacement for Make in the project under study.

8 Conclusion

8.1 Summary

This study set out to investigate the design of an effective CI workflow for an embedded space avionics project. To achieve such a design, it investigated the effect build automation tools had in the design. A declarative approach towards build automation was adopted through the use of a tool called Gradle. A part of the existing build logic was modelled using Gradle. The design approach leveraged the use of Gradle's native knowledge about building embedded software such as Multi-project builds and build-by-convention logic which provided default compile,link and clean task invocations. An integration with Jenkins CI server was carried out. This integration did not require the use of a dedicated interface logic.

A framework for setup of CI server was designed. This setup had one instance of a Jenkins master and multiple build slaves. The framework was inspired from the Git-flow workflow used in software development. Job chains were created depending on Git branch types. These job chains had dedicated workspaces where the builds would execute. A large number of the activities were mapped into the CI server from the production domain. These activities were then executed in powerful remote machines in an automated manner. The effort which was needed to be put in by the developer post the commit stage in the development life-cycle is now exported to the CI server.

This study also performed an evaluation on the build automation tools Make and Gradle. The evaluation was done based on three evaluation blocks. Performance, features provided by the build tool and the complexity of build logic. In most of these aspects, Gradle build tool fared better than Make. This study was performed with the requirement that the logic and frameworks designed for this project had to be compatible for similar projects in the avionics domain. For this objective, every aspect of the design was specifically thought out for a very generic usage.

From this research, I conclude that declarative build paradigm using Gradle for an embedded C project is potentially an improvement in the field of building and testing software. In terms of performance, Make scores over Gradle in some aspects. However, the Gradle build tool is incubating to bring out further improvements for building embedded software. For projects adopting CI, Gradle is the modern tool which helps to form an interface with less cost and effort. Taking into account the discussed factors and results, I would recommend to enrich the SDE of the project under study with Gradle and Jenkins.

8.2 Future Work Items

Default plugins were created for some functionalities in the build logic. An example is for creating of Partially Linked Objects. Gradle does not provide an opportunity to create PLOs from the C sources. A cleaner approach towards PLO generation required

the creation of custom plugins. However, this creation had come at the cost of loosing out on Gradle's another benefit, the up-to-date check. The custom plugin was designed so that it could carry some of its activities in Gradle's execution phase. This effectively blocks the up-to-date features on those tasks which use this plugin. A possible solution is to re-work on the plugin to encapsulate the execution time logic into the configuration block. This would result in a significant performance enhancement as compared to existing logic. However, the effort involved in this encapsulation will be significant.

Only a sub-set of activities provided by Make was designed for Gradle. A complete migration would only be possible if tasks such as static code analysis, documentation generation, and framework for validation tests were implemented with the Gradle build tool. For validation testing, Gradle can be used to build Jar files which contain the test cases to be executed on the production code.

As mentioned, the notion of continuous deployment is not defined in this study. However, an extension towards user acceptance testing and other stages beyond the validation of the production code through CI is to be considered.

In the field of embedded software development, there exists a large number of ad-hoc tasks which need to be automated. A mapping of these tasks into CI server would need careful design and implementation practices.

References

[1] D. Cohen, M. Lindvall, and P. Costa, "Agile software development," *DACS SOAR Report*, vol. 11, 2003.

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley professional, 2000.

[3] "Jenkins documentation." URL: `https://jenkins.io/doc/`, 2012.

[4] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works http://www.thoughtworks. com/Continuous Integration. pdf*, p. 122, 2006.

[5] P. M. Duvall, *Continuous integration. Improving Software Quality and Reducing Risk*. Pearson Education India, 2007.

[6] D. Ståhl and J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study," in *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*, pp. 736–743, 2013.

[7] A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE'08. Conference*, pp. 289–293, IEEE, 2008.

[8] A. Debbiche, M. Dienér, and R. B. Svensson, "Challenges when adopting continuous integration: A case study," in *International Conference on Product-Focused Software Process Improvement*, pp. 17–32, Springer, 2014.

[9] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software technology*, vol. 57, pp. 21–31, 2015.

[10] H. H. Olsson, H. Alahyari, and J. Bosch, "Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pp. 392–399, IEEE, 2012.

[11] R. M. Stallman, R. McGrath, and P. D. Smith, "Gnu make manual," *Free Software Foundation*, vol. 3, 2014.

[12] H. Dockter and A. Murdoch, "Gradle user guide," 2015.

[13] S. I. Feldman, "Make — a program for maintaining computer programs," *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979.

[14] "Recursive use of make." URL: `https://www.gnu.org/software/make/manual/html_node/Recursion.html`.

[15] P. Miller, "Recursive make considered harmful," *AUUGN Journal of AUUG Inc*, vol. 19, no. 1, pp. 14–25, 1998.

[16] "Gnu make - auto-dependency generation." URL: `http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/`.

[17] A. Ant, "Apache software foundation." URL: `http://ant.apache.org/manual/index.html`, 2004.

[18] A. Williamson, J. Gibson, A. Wu, and K. Pepperdine, *Ant Developer's Handbook*. Sams Publishing, 2002.

[19] A. Van Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *CIT. Journal of computing and information technology*, vol. 10, no. 1, pp. 1–17, 2002.

[20] D. Koenig, A. Glover, and D. König, *Groovy in action*, vol. 1. Manning, 2007.

[21] "About gradle inc.." URL: `https://gradle.com/about`, 2017.

[22] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to advanced empirical software engineering*, pp. 285–311, Springer, 2008.

[23] B. Dick, *Action research theses*. PhD thesis, Thesis resource paper. You want to do an action research thesis, 2010.

[24] D. J. Greenwood and M. Levin, *Introduction to action research: Social research for social change*. SAGE publications, 2006.

[25] D. I. Sjoberg, T. Dyba, and M. Jorgensen, "The future of empirical methods in software engineering research," in *Future of Software Engineering, 2007. FOSE'07*, pp. 358–378, IEEE, 2007.

[26] C. Seaman, "Using qualitative methods in empirical studies of software engineering," in *Short course. In: VI Experimental Software Engineering Latin American Workshop (ESELAW 2009), São Carlos, Brazil, November*, 2009.

[27] S. J. Taylor and R. Bogdan, "Introduction to qualitative research methods: The search for meaning," 1984.

[28] S. Wagner and D. M. Fernández, "Analysing text in software projects," *arXiv preprint arXiv:1612.00164*, 2016.

[29] A. E. Cooper and W. T. Chow, "Development of on-board space computer systems," *IBM Journal of Research and Development*, vol. 20, no. 1, pp. 5–19, 1976.

[30] S. A. Dart, R. J. Ellison, P. H. Feiler, and A. N. Habermann, "Overview of software development environments," 1992.

[31] "Gitflow workflow." URL: `https://www.atlassian.com/git/tutorials/comparing-workflows#gitflow-workflow`.

[32] J. Fisher, D. Koning, and A. Ludwigsen, "Utilizing atlassian jira for large-scale software development management," in *14th International Conference on Accelerator & Large Experimental Physics Control Systems (ICALEPCS)*, 2013.

[33] "Atlassian documentation - what is an issue." URL: `https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html`.

[34] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*, pp. 712–721, IEEE Press, 2013.

[35] "Getting started with rtems," tech. rep., 2008.

[36] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 01, pp. 31–57, 1992.

[37] S. Chacon, "Git documentation," 2011.

[38] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pp. 1277–1286, ACM, 2012.

[39] R. M. Stallman and R. McGrath, "Gnu make-a program for directing recompilation," 1991.

[40] "Arm compiler toolchain," tech. rep., 2012.

[41] R. M. Stallman and Z. Weinberg, "The c preprocessor," *Free Software Foundation*, 1987.

[42] F. Koebel and J.-F. Coldefy, "Scoc3: a space computer on a chip: an example of successful development of a highly integrated innovative asic," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1345–1348, European Design and Automation Association, 2010.

[43] K. Hjortnaes, S. Mejnertsen, S. Ekholm, P. Hougaard, and J. van der Wateren, "Software validation facilities," in *Data Systems in Aerospace-DASIA 97*, vol. 409, p. 375, 1997.

[44] K. Thulasiraman and M. N. Swamy, *Graphs: theory and algorithms*. John Wiley & Sons, 2011.

[45] "Cpputest manual." URL: `https://cpputest.github.io/manual.html`, 2014.

[46] M. H. Halstead, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[47] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[48] D. H. Martin and J. R. Cordy, "On the maintenance complexity of makefiles," in *Emerging Trends in Software Metrics (WETSoM), 2016 IEEE/ACM 7th International Workshop on*, pp. 50–56, IEEE, 2016.

Appendix A: Jenkins Job Builder

This section presents some YAML files which were created for efficiently setting-up and managing Jenkins jobs on the remote CI server. There are four different types of YAML files which interact with each other to generate Jenkins job XMLs.

Listing 1: *project.yml* for configuring the types of Jenkins jobs

```
################################################################################
############################    Main Job Chains   ##############################
################################################################################
- project:
    name: main_job_for_master
    chain_name: MASTER_BRANCH
    jobs:
        - 'AS400PROD_BUILD_MASTER_BRANCH'

- project:
    name: main_job_for_develop
    chain_name: DEVELOP_BRANCH
    jobs:
        - 'AS400PROD_BUILD_DEVELOP_BRANCH'

- project:
    name: main_job_for_feature
    chain_name: FEATURE_BRANCH
    jobs:
        - 'AS400PROD_BUILD_FEATURE_BRANCH'

- project:
    name: main_job_for_on-demand_build
    chain_name: ON-DEMAND_BUILD
    jobs:
        - 'AS400PROD_ON-DEMAND_BUILD'

- project:
    name: rulechecker_job_for_master
    chain_name: CHECK_MASTER
    jobs:
        - 'AS400PROD_CHECK_MASTER_BRANCH'

- project:
    name: rulechecker_job_for_develop
    chain_name: CHECK_DEVELOP
    jobs:
        - 'AS400PROD_CHECK_DEVELOP_BRANCH'

- project:
    name: rulechecker_job_for_feature_branch
    chain_name: CHECK_FEATURE
    jobs:
        - 'AS400PROD_CHECK_FEATURE_BRANCH'

- project:
    name: workspace_cleaner
    jobs:
        - 'workspace_cleaner'

################################################################################
############################    Checkout Job   ##############################
################################################################################

- project:
    name: checkout_jobs
    chain_name:
        # - MASTER_BRANCH
        - DEVELOP_BRANCH
        # - FEATURE_BRANCH
        # - ON-DEMAND_BUILD
        # - CHECK_MASTER
        # - CHECK_DEVELOP
        # - CHECK_FEATURE
    repo:
        - scripts:
            branch_name: master
            url_name: ssh://git@as400.de.astrium.corp:2222/sdeng/scripts.git
        - as400prod:
            branch_name: "**"
            url_name: ssh://git@as400.de.astrium.corp:2222/as400_smr/as400prod.git
        # - as400val:
            # branch_name: develop
            # url_name: ssh://git@as400.de.astrium.corp:2222/as400_smr/as400val.git
        - rtems:
            branch_name: master
```

```
            url_name: ssh://git@as400.de.astrium.corp:2222/rtems/rtems.git
        - sde:
            branch_name: master
            url_name: ssh://git@as400.de.astrium.corp:2222/sde/sde.git
        # - delivery:
            # branch_name: develop
            # url_name: ssh://git@as400.de.astrium.corp:2222/as400_smr/delivery.git
    jobs:
        - '_AS400PROD_{chain_name}_checkout_{repo}'

################################################################################
#########################    Make Extra Libs - SCOC3   #########################
################################################################################

- project:
    name: make_extra_libs_for_scoc3_job
    chain_name:
        # - MASTER_BRANCH
        - DEVELOP_BRANCH
        # - FEATURE_BRANCH
        # - ON-DEMAND_BUILD
        # - CHECK_MASTER
        # - CHECK_DEVELOP
        # - CHECK_FEATURE
    jobs:
        - '_AS400PROD_{chain_name}_make_extra_libs'


################################################################################
#########################    Unit Test Job - SCOC3   ###########################
################################################################################

- project:
    name: unit_test_jobs_scoc3
    chain_name:
        # - MASTER_BRANCH
        - DEVELOP_BRANCH
        # - FEATURE_BRANCH
        # - ON-DEMAND_BUILD
    test_subsystem:
        - dhLib
        - dhs
        - dms
        - infra
        - io
        - aocs
        - appliCommon
        - platform
    jobs:
        - '_AS400PROD_{chain_name}_unit_fsw_{test_subsystem}'

################################################################################
#########################    Build Job - SCOC3   ###############################
################################################################################

- project:
    name: build_jobs_on_scoc3
    chain_name:
        # - MASTER_BRANCH
        # - DEVELOP_BRANCH
        # - FEATURE_BRANCH
        - ON-DEMAND_BUILD
    build_config:
        - flight_sysdb
        # - flight_swdb
        # - nsvf_sysdb
        # - nsvf_swdb
        # - flight_custom-sgm
        # - flight_generated_swdb
        # - nsvf_generated_swdb
    jobs:
        - '_AS400PROD_{chain_name}_build_{build_config}'
```

Listing 2: *templates.yml* for describing job specific settings and calling macros and defaults

```
################################################################################
#########################    Build Job - SCOC3  ################################
################################################################################
- job-template:
   name: '_AS400PROD_{chain_name}_build_{build_config}'
   node: linux
   concurrent: false
   description: 'Builds as400prod using the respective {build_config} configurations on target SCOC3'
   disabled: true
   logrotate:
```

```
        artifactDaysToKeep: '-1'
        artifactNumToKeep: '-1'
        daysToKeep: '-1'
        numToKeep: '20'
    workspace: ${{CUSTOM_WORKSPACE}}/slaves/${{NODE_NAME}}
    project-type: freestyle
    properties:
    - throttle:
        categories:
            - as400prod_build
        enabled: true
        max-per-node: '1'
        max-total: '2'
        option: category
    defaults: scm_build
    triggers: []
    wrappers: []
    builders:
    - 'build_job_{chain_name}_{build_config}'
    - build_name_setter_build_job
    publishers:
    - 'archiving_{build_config}'
    - 'artifact_deployer'
    - 'log_parser_build_job'
    # - 'editable_email_notification'
    # - stash_notifier
```

Listing 3: *macros.yml* for describing execution commands for each job definition in templates

```
################################################################################
############################    Builders    ####################################
################################################################################

##############################SCOC3#############################

####################  Master Chain  ############################

- builder: # Build custom-sgm job
    name: build_job_MASTER_BRANCH_flight_custom-sgm

    builders:
        - shell: |
            scripts/build.sh --clean-delivery --prepare-environment --ignore-warnings --make-archive --object-dump --
                headless --custom-sgm --line-counter

- builder: # Build sys-db job
    name: build_job_MASTER_BRANCH_flight_sysdb

    builders:
        - shell: |
            scripts/build.sh --clean-delivery --prepare-environment --ignore-warnings --make-archive --object-dump --
                headless --line-counter

- builder: # Build swdb job
    name: build_job_MASTER_BRANCH_flight_swdb

    builders:
        - shell: |
            scripts/build.sh --clean-delivery --prepare-environment --ignore-warnings --make-archive --object-dump --
                headless --with-swdb --line-counter

- builder: # Build nsvf_sysdb job
    name: build_job_MASTER_BRANCH_nsvf_sysdb

    builders:
        - shell: |
            scripts/build.sh --clean-delivery --prepare-environment --ignore-warnings --make-archive --object-dump --
                headless --for-nsvf --line-counter

- builder: # Build nsvf_swdb job
    name: build_job_MASTER_BRANCH_nsvf_swdb

    builders:
        - shell: |
            scripts/build.sh --clean-delivery --prepare-environment --ignore-warnings --make-archive --object-dump --
                headless --for-nsvf --with-swdb --line-counter
```

Appendix B: Interview

As part of this study, interviews were conducted with the developers of the team to capture their inputs on how the build automation systems and the CI chains affect their day to day activities. The questions and excerpts of their answers are discussed in this section. Some notations which are used in the interview include:

- Dx: Denotes the developer in the project.
- CIx: A developer who is also responsible for maintaining CI tool for the project.

Interview Questions	**Category**	**Excerpts of a few selected Replies**
Which division do you work - Production or Validation?	Background	
Are you familiar with concepts of Continuous Integration?	Background	
Have you worked with CI tools such as Jenkins?	Background	
Approximately, how often do you contribute to the central mainline?	Background	*D1: It depends. Sometimes, I like to have a copy of my "under-development" code on the server. Other times, I push my code only after I have completely built and tested the code in my local machine.*
Have you worked with the Gradle build tool or with Groovy?	Background	
How often do you touch the build logic for the project?	Background	*D1: The core build logic is completely transparent to my work. I usually only touch Makefiles on the applications that I work on. I would say roughly 20% of my commits are based on modifications to my Makefiles...* *D2: It depends on the feature I am developing. Sometimes, I need to make the same changes across many Makefiles in the project. At these times, my commit content is mostly build file modifications...*

What challenges do you face when using GNU Make to build and unit test your code?	Data Collection	*D1: The core build logic is too complex, difficult to understand. I find it difficult to bring out some changes to extend the build feature set...* *D2: There are a large number of command line arguments which need to be supplied to build the correct variant. It is difficult to keep track of these arguments and their use...* *D3: There are a large number of activities to perform post my commit stage. There is no one single command to execute all these steps...* *D4: The time required to build all the different image variants and run all the different tests is too high. A large amount of my time is spent on these activities...*
From the demonstration and your usage of Gradle, how do you feel about the tool?	Post Implementation Review	D*1: I think its radically different. It seems to provide a clean way of describing build logic. Seems to be concise...* *D2: Its good to see there are very few environmental variables in the design...* *D3: It is a bit difficult to adjust to this DSL. Maybe the documentation needs to be richer...* *CI1: I see that Jenkins is capable of running Gradle tasks directly without the need for definition of an interface logic...*