

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Test Automation in Continuous Integration for Hardware Validation

**Pedro Dias Faria**

DISSERTATION



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Filipe Lima Maranhão de Abreu

Co- Supervisor::

Pedro Moreira - Synopsys Portugal

February 19, 2017



# **Test Automation in Continuous Integration for Hardware Validation**

**Pedro Dias Faria**

Mestrado Integrado em Engenharia Informática e Computação

February 19, 2017



# Abstract

As in software development where bugs ought to be detected and fixed as soon as possible, hardware validation is in the same level of priority when used in applications that seek to dramatically reduce costs and extend systems life.

In these complex embedded systems, various interface systems, such as Field-Programmable Gate Arrays (FPGAs), are distributed across multiple hardware blocks.

The interaction between the various systems involves an integration of firmware and its hardware. With the growing complexity of this systems, specially this interaction, it is imperative to reduce the time of the functional validation process of the hardware.

To achieve this, there is the need to implement an automatic validation hardware environment, allowing systematic and automatic tests on the hardware, as well as showing performance metrics of the hardware with a specific combination of each element in the system board, such as the Driver version on the CPU, the Verilog version configuring the FPGA, and the board version itself.

In this thesis, being conducted at Synopsys Porto, it will be described how an automatic test validation environment in continuous integration is structured and apply it on the context of hardware validation.

It will be also outlined the process to create a dashboard view with the different information related to the systems in testing, with the intent of helping stakeholders within the development teams have a more streamlined view of the process results.

**Keywords:** *Continuous Integration, Hardware Validation*

**Classification:**

- *Software and its engineering → Software creation and management → Software verification and validation;*
- *Hardware → Hardware validation*



# Resumo

Assim como no desenvolvimento de software onde os bugs devem ser detectados e corrigidos o mais rapidamente possível, a validação de hardware está ao mesmo nível de prioridade quando usada em aplicações que procuram reduzir drasticamente os custos e estender a vida útil dos sistemas.

Nestes sistemas embutidos complexos, vários sistemas de interface, tais como os Field-Programmable Gate Arrays (FPGAs), são distribuídos em vários blocos de hardware.

A interação entre os vários sistemas envolve uma integração do firmware e o seu hardware. Com a complexidade crescente desses sistemas, especialmente nessa interação, é imperativo reduzir o tempo do processo de validação funcional do hardware.

Para se alcançar isso, propomos implementar um ambiente de hardware de validação automática, permitindo testes sistemáticos e automáticos no hardware, assim como mostrar métricas de desempenho do hardware com uma combinação específica de cada elemento na placa do sistema, como a versão do Driver no CPU, a versão do Verilog configurada no FPGA, assim como a própria versão da placa.

Nesta tese, realizada na Synopsys Porto, será descrita a estrutura de um ambiente de validação de testes automáticos em integração contínua, e aplicá-lo no contexto de validação de hardware.

Será também delineado o processo de criação um painel com as diferentes informações relacionadas aos sistemas em teste, com o objetivo de ajudar os stakeholders numa equipa de desenvolvimento a ter uma vista dos resultados do processo mais simplificada.

**Palavras Chave:** *Continuous Integration, Hardware Validation*

**Classificação:**

- *Software and its engineering → Software creation and management → Software verification and validation;*
- *Hardware → Hardware validation*





*“Do. Or do not.  
There is no try.”*

Master Yoda



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Problem Statement . . . . .	2
1.3	Motivation and Goals . . . . .	2
1.4	Document Structure . . . . .	3
<b>2</b>	<b>State of the Art</b>	<b>5</b>
2.1	Continuous Integration . . . . .	5
2.1.1	Process . . . . .	5
2.1.2	Components . . . . .	6
2.1.3	Value . . . . .	7
2.2	Continuous Integration Tools . . . . .	7
2.2.1	Jenkins CI . . . . .	8
2.3	Hardware Validation Architectures . . . . .	10
2.3.1	LAVA . . . . .	10
2.4	Conclusions . . . . .	12
<b>3</b>	<b>Research Problem</b>	<b>13</b>
3.1	The System architecture . . . . .	13
3.1.1	Communication Interfaces . . . . .	15
3.1.2	CI-flow . . . . .	17
3.2	Solution Requirements . . . . .	18
3.2.1	Project Views Alignment . . . . .	19
3.3	Project Build Spreadsheet . . . . .	19
3.4	Conclusion . . . . .	20
<b>4</b>	<b>The Dashboard Solution</b>	<b>23</b>
4.1	Use Cases . . . . .	23
4.2	Jenkins Plugin - Filtered Dashboard View . . . . .	24
4.2.1	Auxiliary Plugins . . . . .	25
4.2.2	Server Side Implementation Design . . . . .	27
4.2.3	Front-End Implementation Design . . . . .	30
4.3	Metadata and Pipeline Workaround . . . . .	33
4.4	Conclusion . . . . .	34
<b>5</b>	<b>Evaluation</b>	<b>37</b>
5.1	Setup . . . . .	37
5.2	Requirements . . . . .	37

## CONTENTS

5.3	Results . . . . .	37
5.4	Conclusions . . . . .	38
<b>6</b>	<b>Conclusions and Future work</b>	<b>39</b>
6.1	Contributions . . . . .	39
6.2	Limitations . . . . .	39
6.3	Future Work . . . . .	40
	<b>References</b>	<b>41</b>

# List of Figures

2.1	Continuous Integration Process . . . . .	6
2.2	Common connection scheme [Nen16] . . . . .	9
2.3	LAVA Workflow . . . . .	11
3.1	View of an example Build . . . . .	14
3.2	CI environment for Hardware Validation Context . . . . .	15
3.3	HAPS®-Developer eXpress (HAPS-DX) . . . . .	16
3.4	ARC interface architecture . . . . .	16
3.5	PCI Express (PCIe) interface architecture . . . . .	17
3.6	Project Views organized in the Jenkins system. . . . .	19
3.7	Hardware Test Summary Results . . . . .	19
3.8	Different tools and controller Versions . . . . .	20
4.1	Dashboard plugin Use Case diagram . . . . .	24
4.2	View creation menu . . . . .	24
4.3	Dashboard customization menu . . . . .	24
4.4	Mission Control Plugin UI . . . . .	25
4.5	XML file containing a Builds' information . . . . .	26
4.6	Metadata option inside a Job customization panel . . . . .	27
4.7	CLI command update-metadata . . . . .	27
4.8	<i>Top-down</i> approach diagram . . . . .	28
4.9	Filtered Dashboard View classes diagram . . . . .	29
4.10	Interaction Diagram . . . . .	30
4.11	Dashboard Landing page . . . . .	31
4.12	Overall view of a Project . . . . .	32
4.13	Overall view of the same Project (Fig. 4.12) with filters applied . . . . .	32
4.14	Parameterized project selection . . . . .	33

## LIST OF FIGURES

# Abbreviations

API	Application Programming Interface
CI	Continuous Integration
CLI	Command Line Interface
CPU	Central Processing Unit
FPGA	Field-programmable gate array
HAPS	High-performance ASIC Prototyping Systems
HW	Hardware
IDE	Integrated Development Environment
IP	Intellectual Property
R&D	Research and Development
SoC	System on a Chip
SW	Software
UI	User Interface





# Chapter 1

## Introduction

Hardware validation process is essential when the hardware is used on complex embedded integrated systems in applications that pretend to reduce drastically the system costs and expand its durability[Sco].

In embedded systems, various interfaces are distributed by multiple blocs of hardware. The interaction between these systems wraps an integration of the Firmware and its respective hardware. Examples of these systems are the Reset and Clock, energy management, security and other blocks[ASV].

Although the growing complexity of the systems, namely the interaction between firmware and hardware, the process of functional hardware validation needs to be drastically reduced and become more effective due to the "time-to-market". This requirement is even more relevant on the mobile communication industry, making the validation process a major part of the project development cost[PJ15].

These criteria motivate the creation of an automatic hardware validation environment, with characteristics that permit conducting systematic and automatic tests, as well as providing verification indicators tracking and test quality[SY15][GPD<sup>+</sup>11].

In this document we describe the development of a solution to help development teams track their work progress in their Jenkins projects. It is expected to help the validation process both in Hardware and Software development by creating a certain level of abstraction, so it can be applied to both

This chapter presents an introduction to the document, by making an overview of the problem, explaining our motivation and the goals we pretend to achieve and the dissertation structure.

### 1.1 Context

This Dissertation was proposed by Synopsys Portugal, chosen as the MSc Dissertation in Informatics and Computing Engineering of the Faculty of Engineering of the University of Porto. It

emerged from the need to monitor and exhibit the status of current projects in development to stakeholders not accustomed to the used development tools. In this case, the Jenkins CI.

In order to mitigate the complexity of the problem, new validation techniques will have to be investigated in which functional requirements are verified by stimulating the Hardware with configurations and specific test sequences.

SNPS Portugal Lda office is located at Maia (Tecmaia) and it is one of the offices of Synopsys Inc, the leading American company by sales in the Electronic Design Automation industry.

The Synopsys IPK R&D team, the one collaborating with this project, is the team responsible for the validation process of the developed IP designs [Moy], following a specific set of requirements and specifications to be verified accordingly a series of compliance tests defined by the technologies consortia [HLA] [PS].

This team raised the need of having a more centralized and reliable source of information regarding the validation process of their projects. These projects consist of developing and optimizing the firmware utilized in HDMI and PCI-express sockets.

## 1.2 Problem Statement

The subjectivity of validation process in any kind of software development requires the use of some type of structure and categorization of data. This can help an easier access in troubleshooting flaws in the development.

The hypothesis is that with our solution, the productivity in development teams will increase by having this information displayed in a streamlined, precise and straightforward way.

In order to validate this, the system has to be implemented in a real project and feedback collected by the team to understand if the value added is noticeable.

## 1.3 Motivation and Goals

The hardware validation process is directly related with verifying if the different clients configurations will be fulfilled. As such, the validation process can become a subjective process, since it involves assessing how the behavior of the hardware will operate in the most diverse applications and conditions. The process typically consists of activities which include system modeling, prototyping and evaluation by the user.

With this dissertation, we helped to build up a continuous integration environment for hardware validation by developing a Jenkins plugin in form of a dashboard in order to help Synopsys R&D teams on the hardware prototyping process.

To achieve this, the goals for the dissertation are:

- Define an automatic test management structure for Hardware validation;
- Define techniques to label and manage the Hardware validation test results;

- Development of an web application to support the automatic test system;
- Test the web application and validate its usefulness.

### **1.4 Document Structure**

In addition to the introduction, this dissertation report contains 5 other chapters. Chapter 2 describes the state of the art and present related work. Chapter 3 presents the system we want to base our study on and test. Chapter 4 describes the implementation of our application. Results and evaluation are shown in chapter 5. Finally, in chapter 6, is presented the conclusions and future work suggestions.

## Introduction

## Chapter 2

# State of the Art

In this chapter is described the state of the art of related work and technologies. It is divided in three parts. Section 2.1 is explained the Continuous Integration process. In section 2.2 is presented the different tools for continuous integration in software development for future contextualization in hardware validation. In section 2.3 is presented one hardware validation architecture in which will be based our solution.

### 2.1 Continuous Integration

Continuous Integration (CI), in Software Engineering, is the process of merging all the working copies of developers to a shared mainline several times a day. It was first mentioned by Grady Booch in his 1994 book "Object-Oriented Analysis and Design with Applications" [Boo94], although he did not advocate integrating several times a day.

Extreme programming (XP) adopted this concept, when Kent Beck described XP [Bec99] as an agile methodology designed for improving productivity, flexibility and teamwork in projects, and did advocate integrating more than once per day, not leaving "unintegrated" code for more than a couple of hours, thus presenting CI as one of its practices.

CI was set to resolve prevent integration problems, referred to "Integration Hell" in early characterizations of XP, where developers used to divide work in modules, that once completed, would be integrated to create the application as a whole. It was a costly process in terms of time and effort, arising a number of bugs in the process.

The idea behind CI is to reduce this issues by integrating early and often, so as to avoid the "integration hell".

#### 2.1.1 Process

Continuous Integration is a cyclic process, as seen in the figure 2.1, divided in the following steps:

1. **Commit** - Developer modifies the code, tests it and commits to the repository;
2. **Build** - Code is fetched from the repository, integrated and built;
3. **Test** - Tests are performed on the build, followed by a report of the outcome;
4. **Report** - Developers are notified about the build and tests results

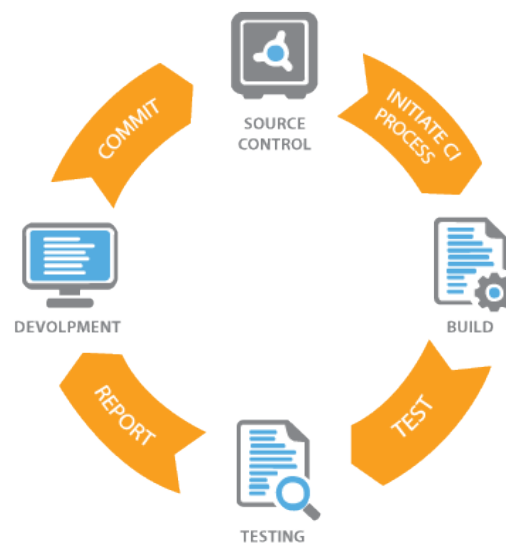


Figure 2.1: Continuous Integration Process

The process is dependent of the build and tests duration. It is expected to be fast so it can be run several times a day.

### 2.1.2 Components

In this section are enumerated each component in the CI process.

#### 2.1.2.1 Developers

Code developers are responsible for the creation of software. Continuous Integration is backed by several important principles and practices [PSA07]:

- **Check in frequently** - Frequent commits means that smaller amount of code changed, which leads to an easier process of finding bugs;
- **Create automated tests** - Tests should be automated and have high code coverage;
- **Don't check in broken code** - Before commit, developers should test the code locally so errors aren't unnecessary introduced;

- **Don't check in when the build is broken** - If a build is broken, it should be made top priority an effort in fixing it, so the errors do not propagate.

Many teams develop rituals around these policies, meaning the teams effectively manage themselves, knowing that if software is broken, they will receive immediate feedback.

#### **2.1.2.2 Repository**

A central repository with the project code using a Source Code Management (SCM) tool or Version Control System (VCS) is needed to perform a successful CI, maintaining the code reachable so the developers can get the most recent version of the source code. These systems are important for software development as they serve as backup for programmers. It is possible to revert changes or pull previous code, allowing its use without the risk of breaking working code. Repositories should contain the source code and everything else needed to create a build, such as: test files and scripts, database schemas, libraries and install scripts. Everything needed to run the software in a new machine.

#### **2.1.2.3 CI Server**

The CI server is the instrument behind Continuous Integration. It is used to implement continuous process of applying quality control of the software, by pulling the latest code from the repository, create a build and report the developers a feedback of the outcome. There are several CI tools implementing this system, which we will talk in section 2.2 with more detail.

### **2.1.3 Value**

By adopting Continuous Integration, you not only reduce risks and catch bugs quickly, but also move rapidly to working software.

With low-risk releases, you can quickly adapt to business requirements and user needs. This allows for greater collaboration between ops and delivery, fueling real change in your organization, and turning your release process into a business advantage.

## **2.2 Continuous Integration Tools**

In order to set up a successful agile software development environment, it is essential to include a method of continuous integration in the process [AF12]. This practice can be easily adapted to the hardware validation process, since it operates the same way as in software development:

1. Hardware logic is modified;
2. Firmware is built and deployed on the physical hardware;
3. Automated tests are run;

4. Hardware is validated.

To contextualize this process into the hardware validation process, we must first choose a continuous integration tool before designing the architecture of the system. It is known that there is a wide range of choices for continuous integration tools available, but for this project it was selected Jenkins as the preferred one.

### 2.2.1 Jenkins CI

Jenkins [Jen16] is a continuous integration and continuous delivery tool application, used to build and test software projects and help developers integrate new changes and features to the project. It also provides ways to delineate customized build pipelines and integrate with several testing and deployment technologies.

It is the application of choice to create our system thanks to the several features it offers, such as:

- **Open Source** - Jenkins can be extended and modified in its majority, and makes uncomplicated to develop new plugins. This allows the creation of the dashboard plugin proposed;
- **Automated Jobs / Jobs Management** - Jenkins can integrate with most software configuration management and build tools available. With this it is possible an easy script management of jobs to build, deploy and test the firmware developed;
- **Build Reports** - The tool can automatically receive and generate logs of successful or failed builds, which can then be parsed (for example into an XML) and eventually used in a plugin or external application;
- **Distributed System** - Jenkins can distribute builds and test loads to specific machines with the requested configurations.

#### 2.2.1.1 Jenkins Pipeline Jobs

In modern environment, delivering innovative ideas in a fast and reliable manner is extremely significant for any organization to better respond the dynamic market requirements [Son15].

Recently, inside the Jenkins tool, was created a new ecosystem that allows implementing jobs in a domain specific language. It is referred to as Pipeline [Cib] (formerly known as Workflow).

Pipeline is a powerful tool available for Jenkins users to implement various software delivery pipelines in code.

Since any peripheral hardware device can be attached to a Jenkins node, and these nodes requiring Java only, almost every development machine can be attached. A Sample scheme can be seen in the figure 2.2.



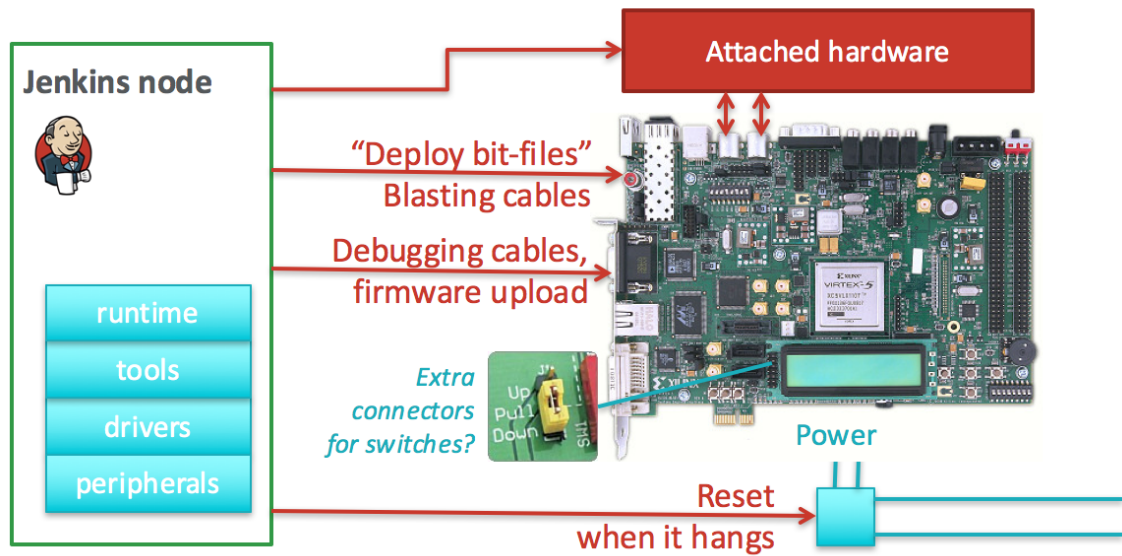


Figure 2.2: Common connection scheme [Nen16]

When connected, Jenkins jobs could invoke common EDA tools via command-line interfaces, which can be easily done by a `Execute shell` build steps in free-style projects.

Pipeline as Code is an approach for describing complex automation flows in software lifecycles, e.g: build, delivery, deployment.

In Jenkins there are two most popular plugins: Pipeline [Cib] and Job DSL [JADM]. JobDSL Plugin internally generates common freestyle jobs according to the script, so it's functionality is similar to the classic approaches. Pipeline is fundamentally different, because it provides a new engine controlling flows independently from particular nodes and workspaces. So it provides a higher job description level, which was not available in Jenkins before.

Accordingly to Oleg Nenashev [Nen16], there are several Pipeline features that may be of value to embedded and hardware projects.

- Robustness against restarts of Jenkins master.
- Robustness against network disconnects. `sh()` steps are based on the Durable Task plugin [Gli], so Jenkins can safely continue the execution flow once the node reconnects to the master.
- Possibility to run tasks on multiple nodes without creating complex flows based on job triggers and copy artifact steps. Achieved via combination of `parallel()` and `node()` steps.
- Ability to store the shared logic in standalone Pipeline libraries

In conclusion, Jenkins is a powerful automation framework that can be used in many areas. Although it has no dedicated plugins for test runs on hardware, it provides diverse general-purpose "building blocks", that allow the implementation of any flow.

Pipeline as code can greatly simplify these complex implementations in Jenkins. It continues to evolve and extend support of use-cases.

## 2.3 Hardware Validation Architectures

Subsequently to deciding the CI tool to use, we must start designing the architecture of the target environment. The framework we came across that accomplishes our goal to some extent was the LAVA project.

### 2.3.1 LAVA

The Linaro Automated Validation Architecture (LAVA) project [Lin16] is a "continuous integration system for deploying operating systems onto physical and virtual hardware for running tests". It can run simple boot loading and system level tests, where the results can be exported and inspected subsequently.

It is composed by two main components, as seen in the figure 2.3:

- The LAVA Server: a web application that provides a central scheduler, dashboard for job monitoring and test result visualization;
- The LAVA Dispatcher: a sub-system that manages communication to the physical test devices.

The overall idea is allowing us to make continuous testing, quality control and automatic validation for projects of all sizes.

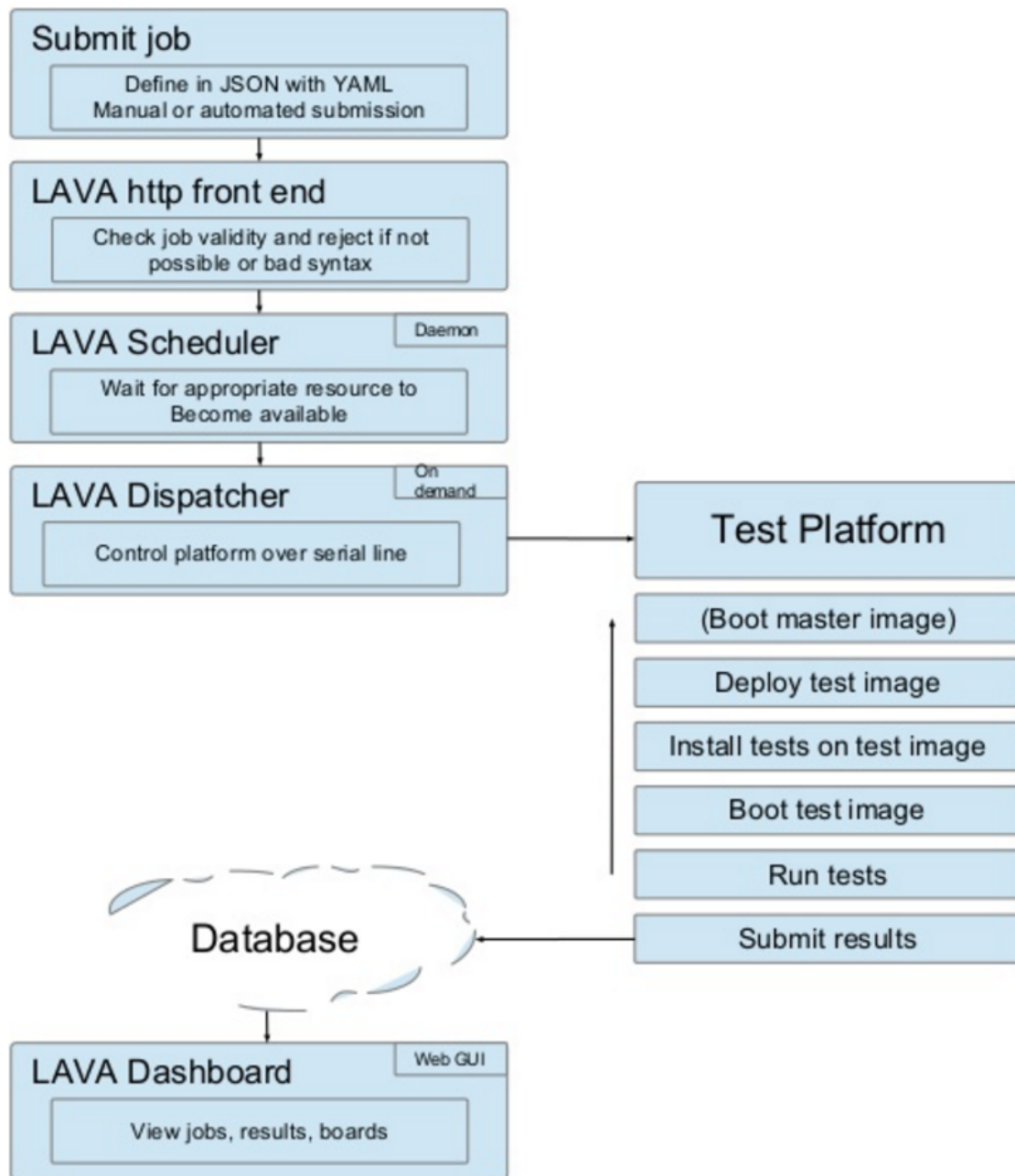


Figure 2.3: LAVA Workflow

It is to note that LAVA itself doesn't run the tests, it is only an enabler as it does not define what can be run. It is a merely black-box to CI, since the test jobs still have to be provided by an external source (usually Jenkins).

The framework serves to test the boards from the Linaro validation lab in Cambridge, where all devices have the same LAVA interface.

## 2.4 Conclusions

The market is full of CI tools available to use, but it is not very explored in the hardware validation context.

Since Synopsys already started to implement a Continuous Integration environment using Jenkins, with automated jobs by this time created in the application, and it providing key features for the goal system, makes sense to continue this approach to develop a new plugin for the tool.

Concluding, we can adapt the LAVA architecture to Synopsys interfaces using Jenkins, by using the job management and distributed system features as the central scheduler and dispatcher, and creating the dashboard plugin to supervise the test results.

## Chapter 3

# Research Problem

In this chapter, we present how the Continuous Integration System of the R&D IPK team in Synopsys Porto is arranged and what needs to be improved.

We will describe its architecture in section 3.1, with a little description of its different communication interfaces in section 3.1.1 and current continuous integration flow in section 3.1.2. It is also listed the requirements we will implement in the solution in section 3.2, following by the current method of displaying the project builds information in section 3.3. Finally a conclusion in section 3.4 about the problem in study.

### 3.1 The System architecture

Most compliance tests take long periods to finish (aprox. 4 hours, as displayed in the figure 3.1 top-right corner). In addition to setting up and recreating the test environment for each run, it is felt as a waste of time. The current test automation environment, due to its manual labor has some issues such as the inconsistency between tests and the lack of traceability among product versions in testing. Hereupon, we want to increase efficiency and productivity of the R&D teams in HW/SW testing and improve reporting to these teams.

To this goal, it is desirable to create an automated test architecture that eliminates the need of employing an engineer to configure the equipment by using automated test equipment, which will allow the continuous testing of different features configurations through automation. By consequence this permits an improved and robust test coverage, as well as increasing their number and velocity. This will summarize in a more refined and consistent testing process, with an improved degree of trust and certainty on the results.

This architecture should meet the following requirements:

- Speed up testing to allow for accelerated releases which:
  - Reduces testing costs;
  - Reduces time in testing phase.
- Allow testing IP's features continuously;

## Research Problem

- Improve test coverage;
- Ensure consistency;
- Improve the reliability of testing;
  - Consolidate the testing process



Figure 3.1: View of an example Build

The block diagram shown in the figure 3.2 shows the hardware testing architecture which is analogous to a common continuous integration software development setup.

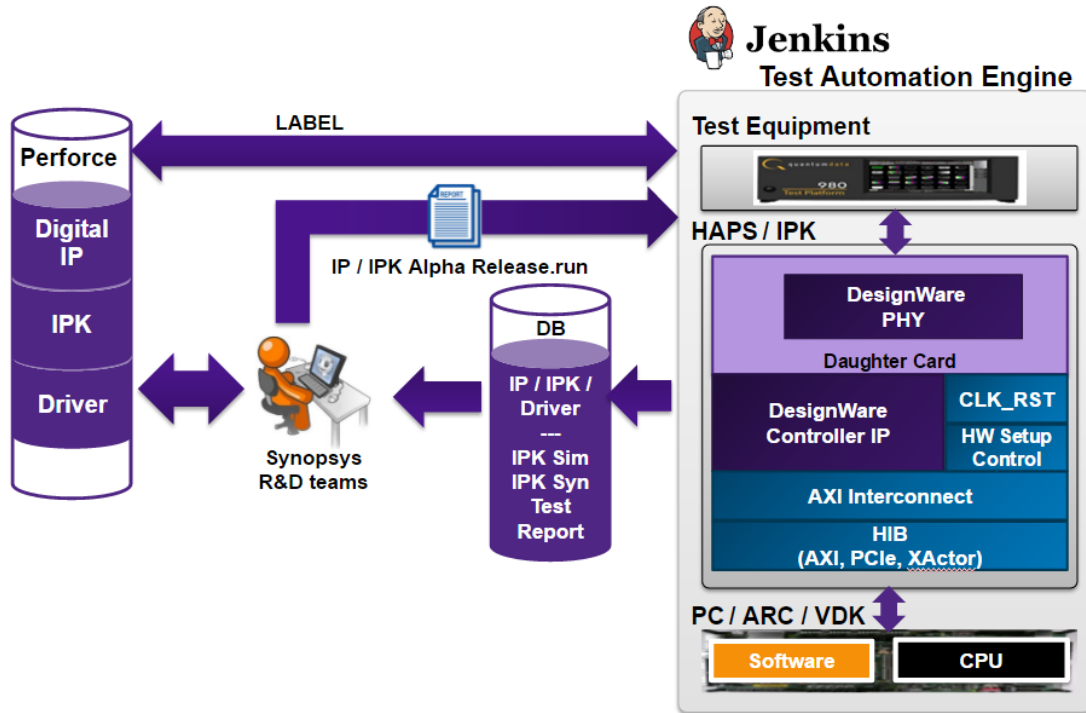


Figure 3.2: CI environment for Hardware Validation Context

This architecture contains all the components used for a successful CI environment, as described in section 2.1.2, being the object of test and analysis the multiple IP configurations, for the different communication interfaces, namely PCIe and ARC (see sections 3.1.1.3 and 3.1.1.2).

### 3.1.1 Communication Interfaces

Here are succinctly described the HAPS platform and the different communication interfaces utilized as test objects by the IPK team.

#### 3.1.1.1 HAPS

HAPS (High-performance ASIC Prototyping Systems) is an integrated and scalable hardware-software solution utilized by design and verification teams to improve their ASIC design schedules and avoid costly device re-spins[Sync].

This prototyping solution consists of a collection of modular, easy-to-use products for ASIC and SoC prototyping that include HAPS hardware components supported by an integrated software tool flow for design planning, FPGA synthesis, and debug.

## Research Problem

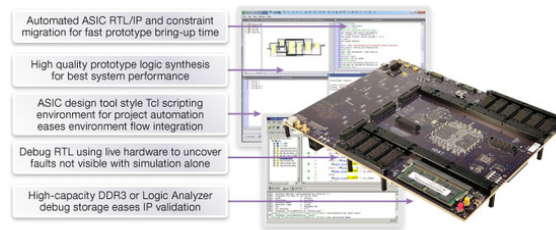


Figure 3.3: HAPS®-Developer eXpress (HAPS-DX)

### 3.1.1.2 ARC

Synopsys' DesignWare® ARC® Processors[Syna] are a family of 32-bit CPUs that SoC designers can optimize for a wide range of uses, from deeply embedded to high-performance host applications in a variety of market segments, such as Automotive and Industrial, Internet of Things or mobile.

Designers can adapt their products by using configuration technology to tailor each ARC processor instance to meet specific performance, power and area requirements. These processors are also extendable, allowing designers to add their own custom instructions to increase performance.

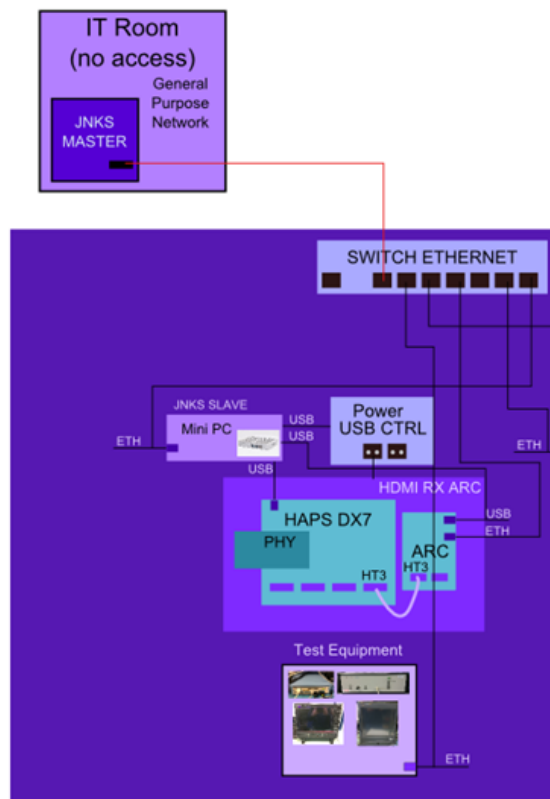


Figure 3.4: ARC interface architecture



### 3.1.1.3 PCIe

PCI Express (Peripheral Component Interconnect Express), is a high-speed serial computer expansion bus standard, designed to replace the older bus standards[[Wikb](#)][[Synb](#)]. Recent revisions of the PCIe standard provide hardware support for I/O virtualization.

The PCI Express electrical interface is also used in a variety of other standards, most notably in ExpressCard as a laptop expansion card interface, and in SATA Express as a computer storage interface.

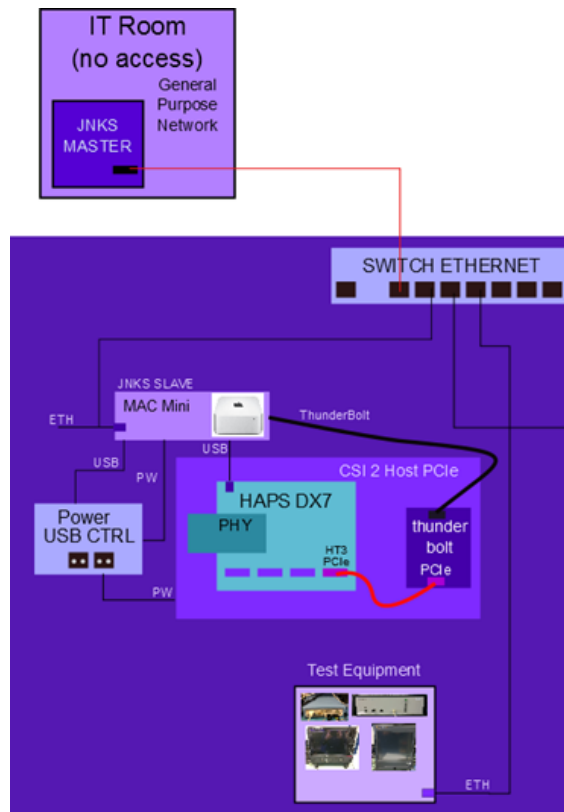


Figure 3.5: PCI Express (PCIe) interface architecture

### 3.1.2 CI-flow

To better understand how the test automation for hardware/software validation works, the rundown of the process is described as follows:

1. The R&D teams create an hardware image with a specific IP configuration to be deployed to the HAPS platform. At the same time, a firmware is packaged with its specific drivers to be deployed in the target interface, e.g. ARC and PCIe;
2. These files are committed to a VCS repository;

## Research Problem

3. The Jenkins Master node receives notification of these changes and proceeds to run the associated Job. The Job is dispatched to the appropriate slave node, as seen in the figures 3.5 and 3.4, in a test automation rack that will trigger a set of sub-steps:
  - 3.1. program the HAPS FPGA with the submitted hardware image;
  - 3.2. boot the host interface and loads the firmware;
  - 3.3. run a set of tests (IP compliance tests) from the test equipment and a log is generated;
4. The log is then parsed and a spreadsheet (detailed in section 3.3), containing all the relevant information is created. The spreadsheet is then validated by the R&D teams.

The last step of this flow is is faulted mainly by three dissatisfactory characteristics:

- The use of a spreadsheet file, which we detail its faults thoroughly in section 3.3, that has no correlation with the Jenkins environment;
- Jenkins default views of the IP projects do not display enough information to reach conclusions about the test results with ease;
- Jenkins build views do not contain specific information about the configuration of the tested hardware, as seen in the figure 3.1.

Ultimately, these are the points we want to correct in order to improve the last step, with this dissertation work.

## 3.2 Solution Requirements

For the display of the desired information in order for a better test result analysis, it was decided to implement a dashboard plugin as solution. Since dashboards allow stakeholders to monitor the contribution of the various elements in the organization. They allow the display of specific data points of the system in question, provided in a single "snapshot" [Wika].

Presently, R&D teams utilize an Excel spreadsheet created by its IP Prototyping Team, where most of the time the data is manually input with the test results obtained from a test compliance log. This method is a poor solution since it does not provide consistency between tests, nor automatic traceability among the different tested versions, as explained in section 3.3.

The dashboard plugin will serve as a project monitoring view where the R&D teams can constantly observe the results of the hardware validation phase testing. The main features of the dashboard include:

- View customized to each development team;
- Information of each test configuration;
- Display of test results and build performance indicators (e.g. build average time of each configuration).

## Research Problem

### 3.2.1 Project Views Alignment

The R&D Jenkins environment is currently organized by different Views, each referencing a different project. As seen in the figure 3.6, a project View is very unidimensional, not displaying enough information.

Aware that each R&D team can have more than one Project associated to it, this is one requirement we had in mind while creating our solution, as explained in section 4.2.2.

Bluetooth DDR EQOS XGMAC <b>HDMI</b> IPK MIPI PCIe SATA Software USB +							
S	W	Categorized - Job ↓	Primary Owner	Último Sucesso	Última falha	Duração da Última	
🔴	🔴	DWIPK HDMI RX PIPE	pmmoreir	2 mês 0 dias - #11	N/A	2 h 0 min	
🟢	🔴	DWIPK HDMI RX dotRun_test	psousa	1 yr 0 mês - #38	1 yr 0 mês - #31	5 h 30 min	
🟡	🔴	DWIPK HDMI RX multijob	pmmoreir	22 dias - #147	1 mês 5 dias - #144	2 h 29 min	
🟢	🔴	DWIPK HDMI RX multijob_HW_arc	psousa	22 dias - #132	1 mês 11 dias - #122	1 min 22 seg	
🟢	🔴	DWIPK HDMI RX multijob_HW_haps	pmmoreir	22 dias - #144	1 mês 5 dias - #141	17 seg	
🟢	🔴	DWIPK HDMI RX multijob_HW_quantum	pmmoreir	22 dias - #118	1 mês 8 dias - #112	2 h 23 min	
🟢	🔴	DWIPK HDMI RX multijob_p4sync	pmmoreir	22 dias - #167	3 mês 16 dias - #95	2 min 17 seg	
🟢	🔴	DWIPK HDMI RX multijob_p4trigger	psousa	3 mês 21 dias - #27	N/A	42 seg	
🔴	🔴	DWIPK HDMI RX multijob_VS	unknown	N/D	3 mês 22 dias - #2	2 min 18 seg	
🟢	🔴	DWIPK HDMI RX multijob_VS_p4	unknown	3 mês 22 dias - #10	3 mês 22 dias - #6	1 min 0 seg	
🟢	🔴	DWIPK HDMI QPRX	pmmoreir	6 mês 25 dias - #1	N/A	4 min 12 seg	
🔴	🔴	HDMI Compliance Automation	hfaria	N/D	1 yr 6 mês - #4	30 seg	
🔴	🔴	HDMI_TX_IPK	unknown	2 mês 9 dias - #80	6 dias 13 h - #89	28 seg	

Ícone: S M L

Legenda RSS para todos RSS só para falhas RSS só para últimas builds

Figure 3.6: Project Views organized in the Jenkins system.

### 3.3 Project Build Spreadsheet

The IPK team uses an Excel Spreadsheet to track the overall status of each job with different test configurations, illustrated in figures 3.7 and 3.8.

	A	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	AllentGen3 PTC - Gen3																	
2	Overall Result	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PA
3	Agilent Gen3 PTC Software version	8.73	8.73	8.62	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.73	8.7
4	Detected Error Window	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PA
5		PASS	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
6		FAIL																
7	DIL_04_01_02 To check that receiver ignores the reserved fields of the received DLLPs (ReservedFieldsDLLPReceive).	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PA
8		PASS	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
9		FAIL																
10	DIL_05_02_01 The intent of this test is to ensure that a DUT will retransmit a transaction for which a NAK has been issued (RetransmitOnNAK).	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PA
11		PASS	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
12		FAIL																
13	DIL_05_02_011 The intent of this test is to ensure that the DUT's REPLAY_TIMER is working properly by not sending neither an ACK nor a NAK (ReplayTimerTest).	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PA
14		PASS	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
15		FAIL																
Summary		rGen1PTC	rGen2PTC	rGen3PTC	rPCIeCV	rLTSSM	rSynth	rSim	rVersions	rConfig - Customer	rConfig - R&D	rConfig - Nico	Data Thr ...					

Figure 3.7: Hardware Test Summary Results

Figure 3.7 depicts an example build run information in a PCIe interface. It is structured by the following elements:

## Research Problem

- The labels in line 1 contain the project information, about the hardware build, namely the host interface, the PHY[[Wike](#)] and top level hardware configurations, and date of the build;
- The subsequent rows contain the test details, with the columns displaying information about the test result.

	A	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Test Versions	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...	ahbata_h70k_770k_sppcs_151209_022657_snp...
2	DWC_pcie	4.60g	4.60g	4.70g-est02	4.60g	4.60g	4.60g	4.60g	4.60g	4.60g	4.60g	4.60g	4.60g	4.60g
3	vcs	2012.09-3	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1	2014.12-SP1
4	synplify	2013.03-SP1-1			2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1	2014.03-SP1
5	identify	2013.03			2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1	2014.09-SP1
6	protocomp		2014.09-SP2	2015.12	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2	2014.03-SP2
7	xilinx	14.4		14.7										
8	vivado		2013.3	2015.4	2014.3	2014.3	2014.3	2014.3	2014.3	2014.3	2014.3	2014.3	2014.3	2014.3
9	ct	2013.03-SP1-2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2	2015.06-SP2

Figure 3.8: Different tools and controller Versions

In figure 3.8 it is represented more information about the same build run, in a different sheet, showing the Controller Version in the first line, followed with the used tool versions for each configuration.

This exposes various problems, such as:

- **Traceability** - A new spreadsheet has to be created each time a new data-set of test results is created.
- **Susceptible to Human Error** - If an error while populating the spreadsheet data is encountered, a manual fix is needed to correct it.
- **Availability** - Every stakeholder in the process needs a copy of the spreadsheet for analysis.
- **Difficulty to troubleshoot** - The need of correcting a failed test leads the user to search for it in the system.

A dashboard inside Jenkins will resolve this issues by aggregating the information in a single snapshot.

## 3.4 Conclusion

Accuracy and speed need to be improved in the hardware validation process. Having a more centralized information access would greatly aid this improvement.

Despite the fact that the organization of each project is well structured in Views, there is the need of a factor that facilitates its analysis, eliminating some of irrelevant information displayed.

The current solution is inefficient, with information disperse and more than required for the time being.

## Research Problem

With this, it is safe to assume a View in form of a dashboard inside Jenkins is one optimal way to achieve this goal. Jenkins has a great API that allows access of its items inside its various Extension Points [[CIA](#)].

This will eliminate the issues of using a spreadsheet, since all the information is retrieved automatically from the Jenkins server in a reliable manner, and displayed available to every user inside the tool. It will make the validation process faster, concentrating the relevant information in a single view.

## Research Problem

## Chapter 4

# The Dashboard Solution

In this chapter, we will describe the Dashboard solution, its implementation and methodology under its development.

Since Synopsys had already setup the CI server in Jenkins, it was chosen to develop the Plugin for it. We also take the advantage of Jenkins being very customizable, with hundreds of extension points and plugins available to support our needs [Jen16] [CIa].

Nonetheless, it is not a simple task to develop a plugin due to lack of documentation. In order to find out how plugins work, it was crucial to look into other plugins source code.

Our plugin, named **Filtered Dashboard View** plugin [Far17], will be referenced from here on simply as **Dashboard**.

### 4.1 Use Cases

Taking into consideration the R&D project organization of Jenkins, described in section 3.2.1, it was discussed with the collaborating IPK team that the best option was to develop the Dashboard as a View plugin, implementing the class ViewGroup, in order to aggregate different views inside it.

In the Use Case diagram in the figure 4.1 are pictured the actions a user can perform to display our Dashboard in the Jenkins environment.

The plugin is designed to be as simple as creating a new View inside the tool, selecting which other Views are requested to display, as shown in the figure 4.2. Afterwards, the user can freely change its configuration, namely add, remove or change which views will be displayed, demonstrated in the figure 4.3.

It is important to note that all kinds of Views should be allowed to display inside the Dashboard, with the exception of itself and other instances of the same View Class, so no loops are originated from this process.

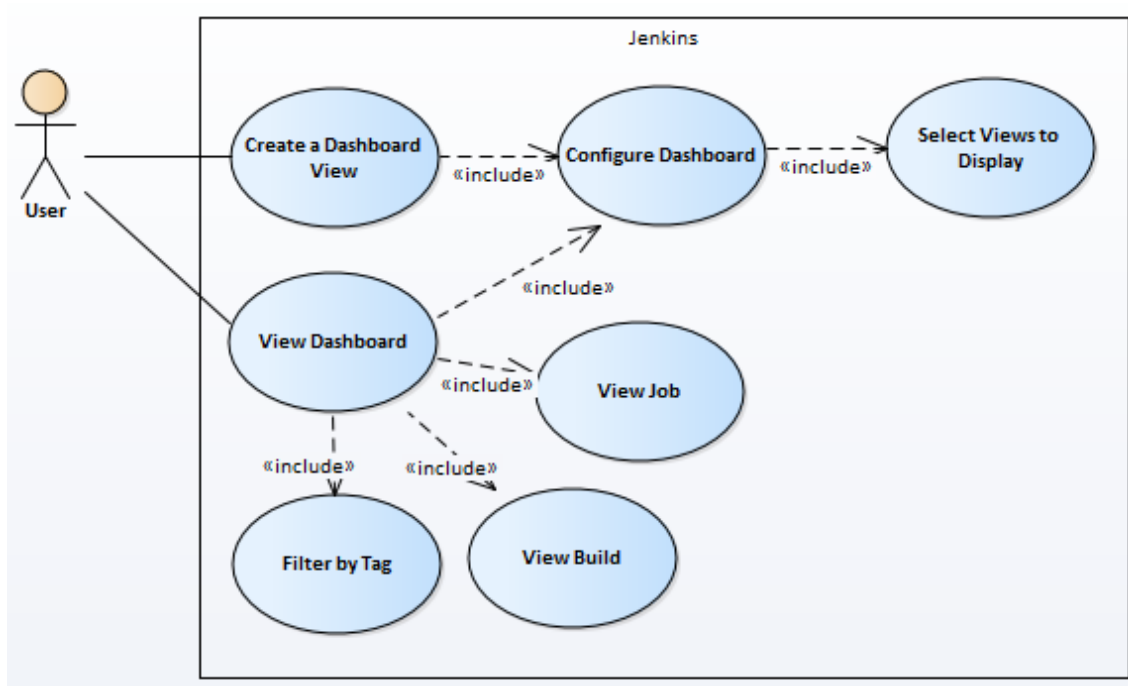


Figure 4.1: Dashboard plugin Use Case diagram

View name

- ☐ All  
This view shows all the jobs on Jenkins.
- ☐ Build Pipeline View  
Shows the jobs in a build pipeline view. The o
- ☐ Categorized Jobs View  
Besides showing items in the list format, allow
- ☐ Dashboard  
Customizable view that contains various portl
- ☐ Delivery Pipeline View  
Shows one or more delivery pipeline instance
- ☒ Filtered Views Dashboard  
Full screen dashboard view featuring build his

Figure 4.2: View creation menu

Name

**Views Filter**

Select views to display:

- ☐ ARC
- ☐ ARC\_UIIMAGE
- ☐ Bluetooth
- ☐ Buildroot
- ☐ DDR
- ☐ EQOS XGMAC
- ☒ HDMI
- ☐ MIPI
- ☒ PCIe
- ☐ SATA
- ☐ Software
- ☐ Templates
- ☐ USB

**UI Settings**

Consider only last 'n' builds

Figure 4.3: Dashboard customization menu

## 4.2 Jenkins Plugin - Filtered Dashboard View

In this section it will be described the thought process of designing our plugin, including the research of similar and auxiliary plugins in 4.2.1, our implementation design in section 4.2.2, and



finally the display of the User Interface of the final product in section 4.2.3.2.

## 4.2.1 Auxiliary Plugins

Since Jenkins [Jen16] supports plugins, with an abundance of extension points which allow it to be extended to meet specific needs of individual projects, it was appropriate to investigate what similar plugins already exist that can be extended or modified as base to our own solution.

In this section, it will be described 2 plugins used to help achieve our final application.

### 4.2.1.1 Mission Control Plugin

Accounting that we want to create a Dashboard, we came across the Mission Control Plugin [She15], that has practically all the base to work with, a full dashboard view that features:

- Job status display;
- Build history and build queue;
- Extends a View Object [Kaw], which can be added to the main dashboard.

Despite being a starting point, the information displayed in the figure 4.4, needed to be trimmed and adapted to our requirements.

Nevertheless, after making a profound source code analysis of this plugin, we had a good understanding on how we could interact with the Jenkins API in various ways, such as retrieving information of all necessary items in our Jenkins instance, exportation of this information and its interaction with the front-end.

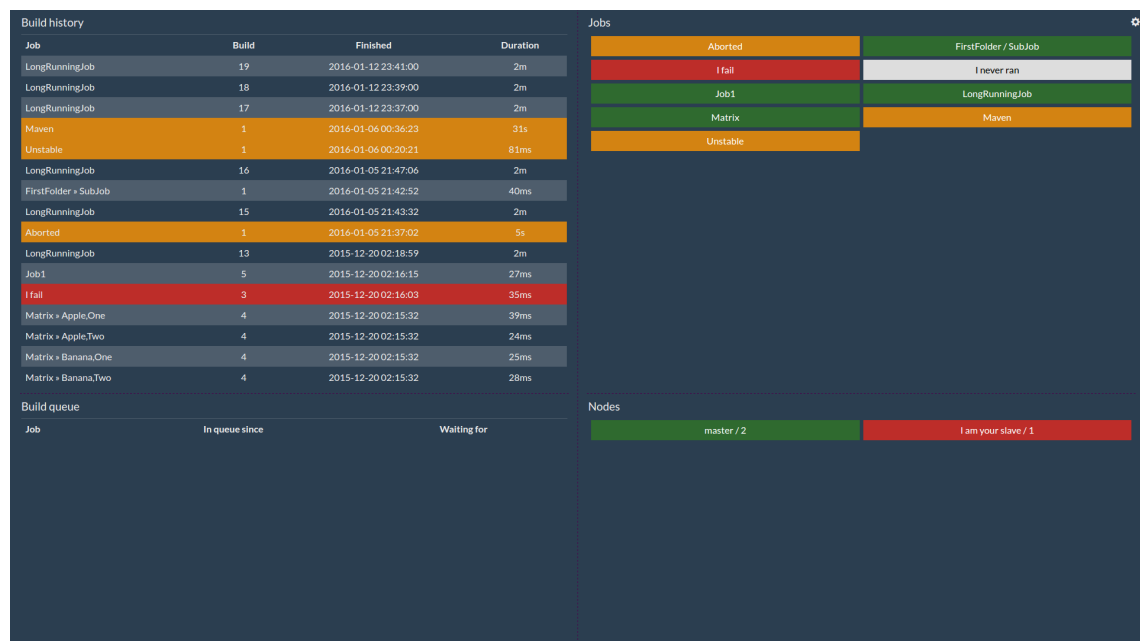
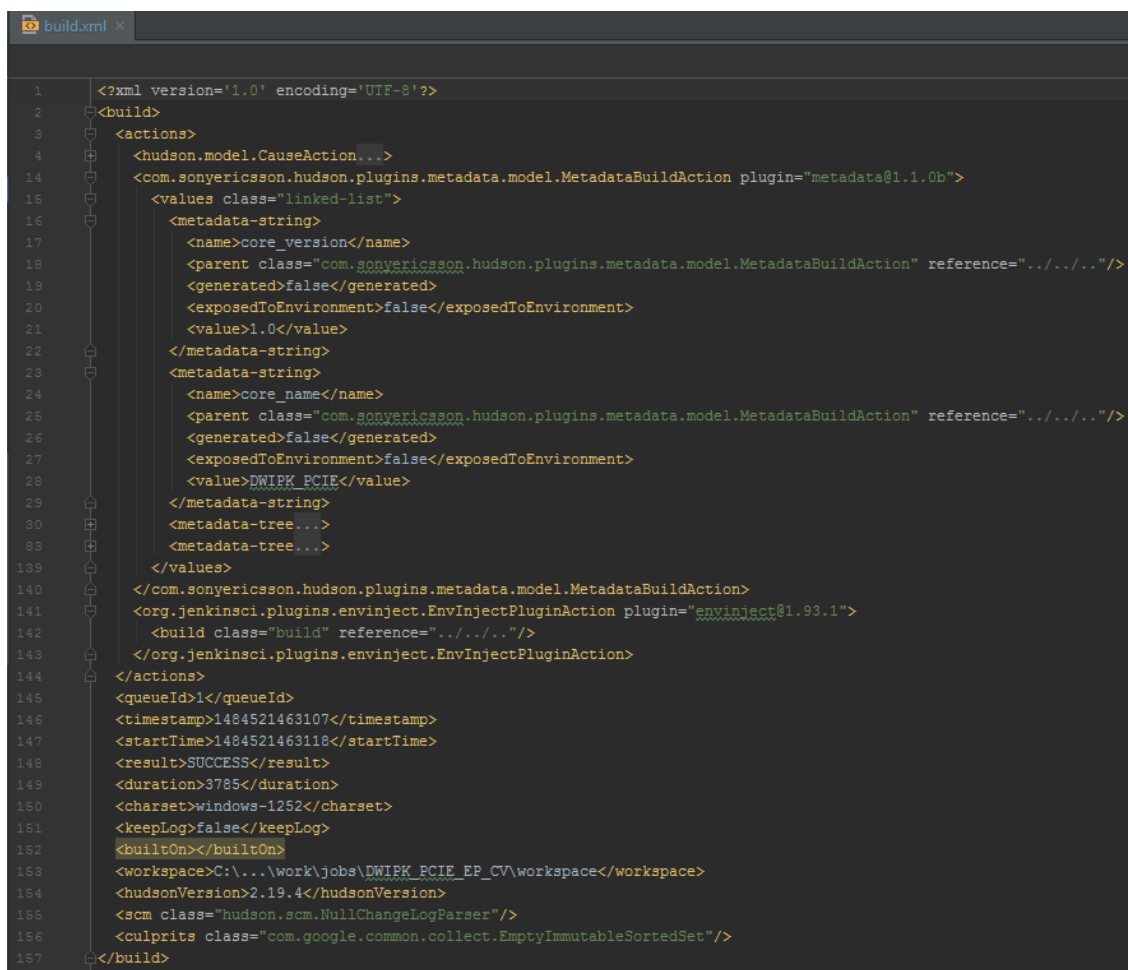


Figure 4.4: Mission Control Plugin UI

#### 4.2.1.2 Metadata Plugin

Considering that one of the requirements is filtering the data displayed with the information of each test configuration, as said in section 3.1, and since Jenkins has the possibility to store information on each of its CI items in XML format, exemplified in the figure 4.5, then it could be stored some type of metadata as well.

To facilitate this, we used the extension point that Metadata Plugin [RT12] provides, in conjunction with the Jenkins API, to apply relevant filters on each job for an easier parsing of these configurations information.



```

1  <?xml version='1.0' encoding='UTF-8'?>
2  <build>
3    <actions>
4      <hudson.model.CauseAction...>
14     <com.sonyericsson.hudson.plugins.metadata.model.MetadataBuildAction plugin="metadata@1.1.0b">
15       <values class="linked-list">
16         <metadata-string>
17           <name>core_version</name>
18           <parent class="com.sonyericsson.hudson.plugins.metadata.model.MetadataBuildAction" reference="../../.."/>
19           <generated>false</generated>
20           <exposedToEnvironment>false</exposedToEnvironment>
21           <value>1.0</value>
22         </metadata-string>
23         <metadata-string>
24           <name>core_name</name>
25           <parent class="com.sonyericsson.hudson.plugins.metadata.model.MetadataBuildAction" reference="../../.."/>
26           <generated>false</generated>
27           <exposedToEnvironment>false</exposedToEnvironment>
28           <value>DWIPK_PCIE</value>
29         </metadata-string>
30         <metadata-tree...>
31         <metadata-tree...>
139      </values>
140    </com.sonyericsson.hudson.plugins.metadata.model.MetadataBuildAction>
141    <org.jenkinsci.plugins.envinject.EnvInjectPluginAction plugin="envinject@1.93.1">
142      <build class="build" reference="../../.."/>
143    </org.jenkinsci.plugins.envinject.EnvInjectPluginAction>
144  </actions>
145  <queueId>1</queueId>
146  <timestamp>1484521463107</timestamp>
147  <startTime>1484521463118</startTime>
148  <result>SUCCESS</result>
149  <duration>3785</duration>
150  <charset>windows-1252</charset>
151  <keepLog>false</keepLog>
152  <builtOn></builtOn>
153  <workspace>C:\...\work\jobs\DWIPK_PCIE_EP_CV\workspace</workspace>
154  <hudsonVersion>2.19.4</hudsonVersion>
155  <scm class="hudson.scm.NullChangeLogParser"/>
156  <culprits class="com.google.common.collect.EmptyImmutableSortedSet"/>
157 </build>

```

Figure 4.5: XML file containing a Builds' information

This Metadata can be applied to Jobs simply through the customization menu, as seen in the figure 4.6, which will apply it to every future Build after this point.

## The Dashboard Solution

Project nome

**Meta Data**

Preset Values

Values

**String**

Name  Value

Expose to environment ☐ [?](#)

**Apagar**

**String**

Name  Value

Expose to environment ☐ [?](#)

**Apagar**

**Save** **Apply**

Figure 4.6: Metadata option inside a Job customization panel

Or alternatively, automate the process simply by calling a post-build CLI command that is provided from the same plugin, using the format demonstrated in the figure 4.7.

```
java -jar jenkins-cli.jar -s http://localhost:8090/jenkins/ update-metadata [-build N] [-data VAL] [-job VAL] [-node VAL] [-replace]

Update or add metadata to a node or job.

-build N : The build number of the job to add data to.
-data VAL : File or URL to the document containing the data to add.
            If no data argument is provided it is assumed to come on stdin
-job VAL  : The name of the job to add data to.
-node VAL : The name of the node/computer to add data to.
-replace  : If existing values should be replaced/merged.
```

Figure 4.7: CLI command update-metadata

### 4.2.2 Server Side Implementation Design

Understanding how the back end in Java interacts with Jenkins and how projects were organized in Views, as explained in section 3.2.1, we defined a *Top-down* strategy to obtain and parse the information of our items of interest. This means we had to process View by View as individual projects to relate their child items in our Dashboard as seen in the figure 4.8.

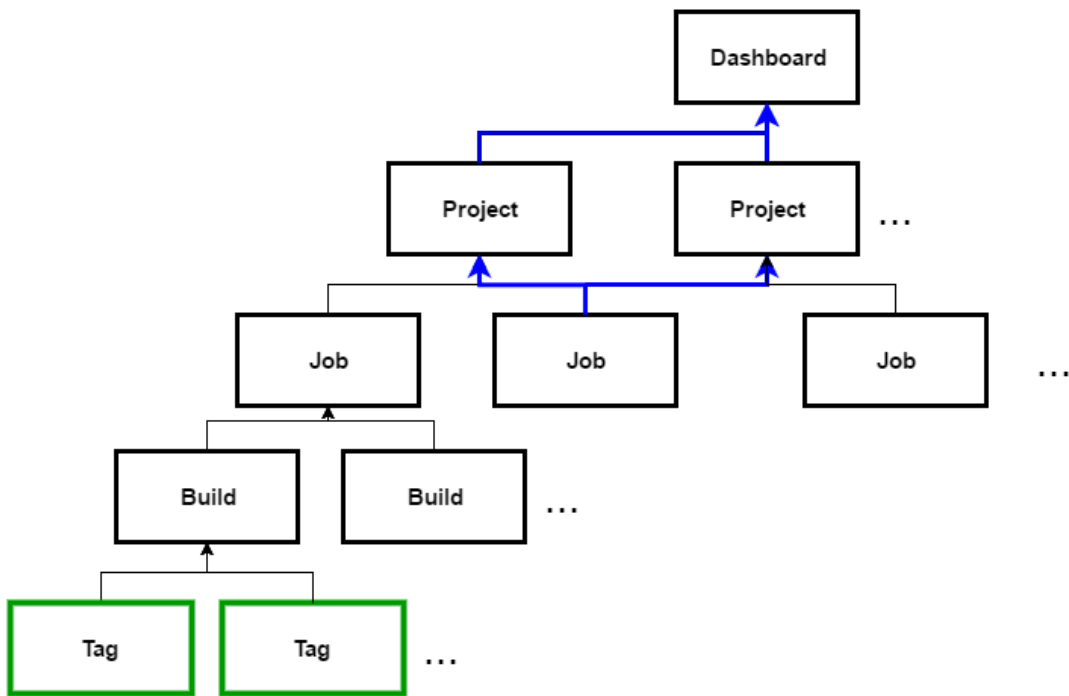


Figure 4.8: *Top-down* approach diagram

It was took in consideration the association between Views and Jobs inside Jenkins, namely that one Job is not associated to any View. Therefore each Job could be displayed multiple times, being represented by the same object.

We also needed a way to store information about the filters that can be applied, so we decided to implement a new Tag child in each Build.

Based on this, our final plugin consists in this five different classes, depicted in the figure 4.9, that work together in parsing the information to display:

- **FilteredDashboardView** - The main class of the Dashboard. It collects all the Data from the Views selected and parses them into our other custom classes. After this collection, it exports the information to Jenkins API, which will be used to render and display in the front-end.

This class refreshes all the information when a new request is made by the front-end, which happens in a predefined time with an auto-refresh to capture new changes in the Jenkins instance, as schemed in the diagram 4.10.

We also ensure concurrency of the Views between all the life cycle of the class with a simple `CopyOnWriteArrayList<View> views = new CopyOnWriteArrayList<View>()` on its instantiation, which safeguards any View modification or deletion in the server.

To ensure our approach scheme is followed, and prevent unnecessary duplicates of Jobs, these mapped in a variable `Map<String, JobData> jobsMap`, with their name serving as identifier, as well each Project mapped with its respective Jobs as `Map<String, ArrayList<String>> jobsInProjectMap`.

## The Dashboard Solution

- **Project** - Custom class representing a View. Contains a list of all Jobs inside the View as `ArrayList<JobData> projectJobs`. It is defined by its name `projectName` and overall project status `projectStatus`. This status is calculated after all jobs are parsed and is set as the worst case scenario between them (i.e. if one Job fails, the whole project is set as Fail).
- **JobData** - Custom class representing a Job. Contains a list of all Builds made by its represented Job as `ArrayList<Build> builds`.

It has a subclass **JobVar** that keeps information of its variables, such as its URL, name and status. This concept was chosen for an easier access in the front-end, where we simply iterate through all these variables for display.

- **Build** - Custom class representing a Build. Contains a list of Tags as `ArrayList<Tag> tags` associated to this Build.
- **Tag** - Class that represents a Metadata, from the plugin described in section 4.2.1.2, `label` and `value` contained in a Build, that will be used to apply filters in the Dashboard. Each **Tag** is the depiction of the configuration parameters utilized during the test process in section 3.1.2.

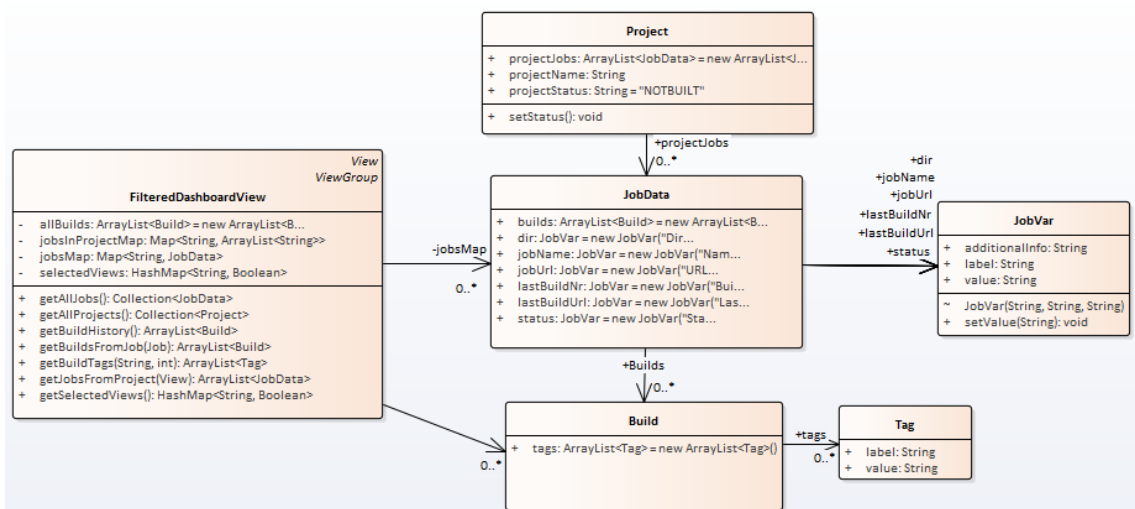


Figure 4.9: Filtered Dashboard View classes diagram

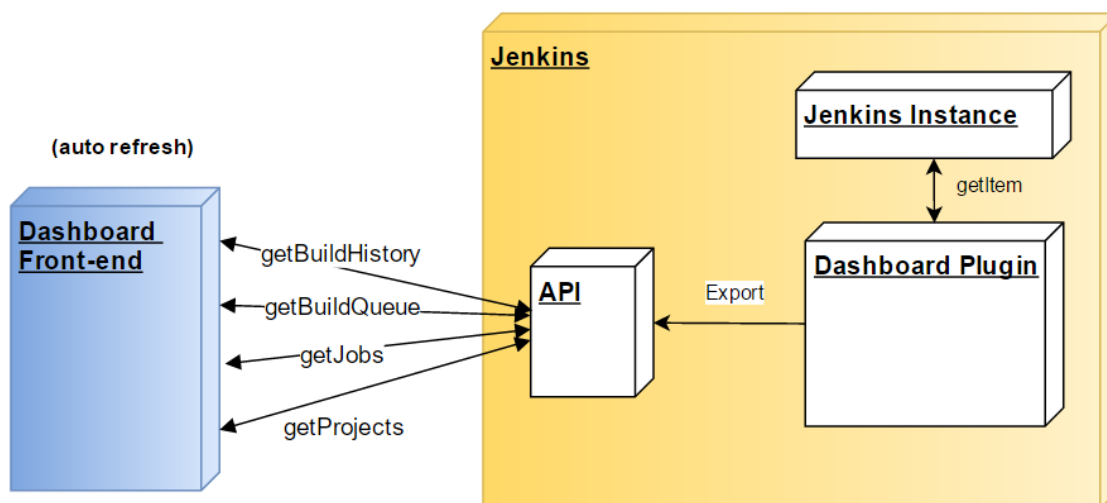


Figure 4.10: Interaction Diagram

To eliminate some clustering of information, in the front-end we display only the last  $N$  builds of each job. This number can be increased in the customization panel from figure 4.3, however it will also increase its processing time proportionally.

Finally, the addition of Tags to filter the information is the key feature of our Dashboard. This concept helps to maintain track of each configuration as requested in section 3.2, with the minimum requisite of adding, or modifying, these labels to Jobs in the building process.

This design permits a totally scalable structure, since it has no limitations on the number of Views associated to the Dashboard, nor its Jobs and Builds. It can also be extendable to put more information in each of its custom classes for additional metrics and indicators in the future if desired. All this information only depends on the Jenkins API and what it supports.

### 4.2.3 Front-End Implementation Design

After the development of the server side, it was needed some brainstorming on how to organize the information structured and display it in the browser.

For this, in the front-end display of the Dashboard, we based on the same scripts of the Mission Control Plugin, described in section 4.2.1.1, along with several common Web Application libraries and frameworks such as Javascript, jQuery and Bootstrap, all processed in Apache Jelly [SF], a Java and XML based scripting and processing engine that allows Jenkins its UI to be extended by plugins.

Since the data exported to the API is already well structured, the presentation process was completed with ease, only requiring a way to exhibit the view of a project with an intuitive display.

To accomplish the aforementioned, was decided to display every Job inside the project as a column in a table, using the library referenced in section 4.2.3.1, being each row cell its Builds organized in a descending chronological order. Each cell has its Tags associated for a quick reference of the configurations utilized in the respective Build.

## 4.2.3.1 Auxliary Library: DataTables - jQuery

DataTables [Ltd] is a plugin for the jQuery Javascript library. It is a highly flexible tool that helped to create and add advanced interaction controls to any HTML table.

Is one of the key components of the front-end that made possible and simple the creation of a display table within the Dashboard, as well the search interaction with the added Tag filters.

## 4.2.3.2 User Interface

Once the Dashboard is selected, it displays every View selected previously, as seen in the figure 4.2, and the overall status of those projects. It also shows all the Jobs inside those Views, and a Build history resulted from these, organized chronologically from the most recent. It is to note that every one of this items is linked directly to its object inside the Jenkins tool, eliminating the need of searching for them in the system.

In the figure 4.11 its is shown the Synopsys IPK R&D team's Dashboard, which are associated both HDMI and PCIe projects.

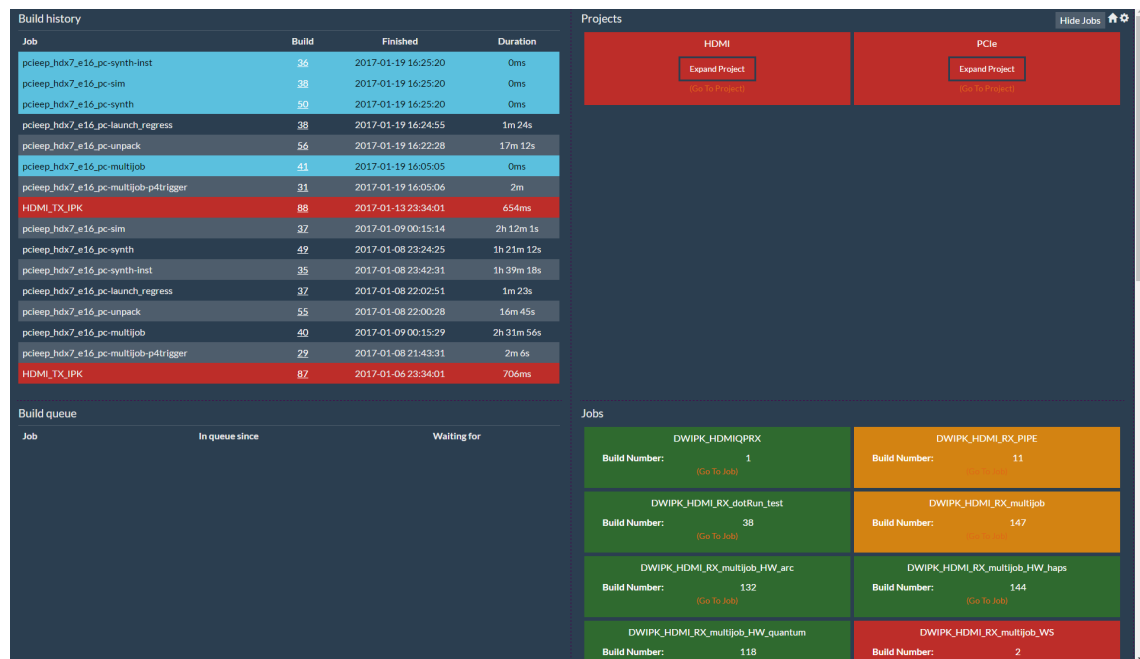


Figure 4.11: Dashboard Landing page

It displays a "straight to the point" facet, that helps to identify exactly which Job or Build is not corresponding to the expected results.

Inside of each sub-view, the user can see information about Jobs, Builds and overall Project status, as well as the different filters defined by the Metadata inputs, referring to the different test configurations, displayed in the figure 4.12.

## The Dashboard Solution

**HDMI**

Go Back

Filter by:

CORE\_VERSION: 1.0 ☐

CORE\_NAME: dwipk\_hdmirx ☐

Show: 10 entries

Search:

Row#	DWIPK_HDM_LRX_dotRun_test	DWIPK_HDMI_RX_multijob	DWIPK_HDM_LRX_multijob_HW_arc	DWIPK_HDM_LRX_multijob_HW_haps	DWIPK_HDM_LRX_multijob_HW_quantum	DWIPK_HDM_LRX_multijob_p4sync	DWIPK_HDM_LRX_multijob_p4trigger	DWIPK_HDM_LRX_multijob_WS	DWIPK_HDM_LRX_multijob_WS_p4	DWIPK_HDM_LRX_PIPE	DWIPK_HDM_IQPRX	HDMI_Comp_liaance_Auto_mation	HDMI_TX_IP_K
1	SUCCESS #38	UNSTABLE #147 Tags: 1.0 dwipk_hdmirx	SUCCESS #132	SUCCESS #144	SUCCESS #118	SUCCESS #167	SUCCESS #27	FAILURE #2	SUCCESS #10	UNSTABLE #11	SUCCESS #1	FAILURE #4	FAILURE #88
2	SUCCESS #37	UNSTABLE #146 Tags: 1.0 dwipk_hdmirx	SUCCESS #131	SUCCESS #143	SUCCESS #117	SUCCESS #166	SUCCESS #26	FAILURE #1	SUCCESS #9	UNSTABLE #10	NO DATA	FAILURE #3	FAILURE #87
3	SUCCESS #36	UNSTABLE #145 Tags: 1.0 dwipk_hdmirx	SUCCESS #130	SUCCESS #142	SUCCESS #116	SUCCESS #165	SUCCESS #25	NO DATA	SUCCESS #8	UNSTABLE #9	NO DATA	FAILURE #2	FAILURE #86
4	SUCCESS #35	FAILURE #144 Tags: 1.0 dwipk_hdmirx	SUCCESS #129	FAILURE #141	SUCCESS #115	SUCCESS #164	SUCCESS #24	NO DATA	SUCCESS #7	UNSTABLE #3	NO DATA	FAILURE #1	FAILURE #85
5	SUCCESS #34	FAILURE #143 Tags: 1.0 dwipk_hdmirx	SUCCESS #128	FAILURE #140	SUCCESS #114	SUCCESS #163	SUCCESS #23	NO DATA	FAILURE #6	NO DATA	NO DATA	NO DATA	FAILURE #84

Showing 1 to 5 of 5 entries

Go Back

First Previous 1 Next Last

Figure 4.12: Overall view of a Project

For a quick filtering through the Tags, the user simply needs to select them in the respective section (Fig. 4.13). This search is made following a regular expression using the AND operator through all the selected Tags, displaying only the Builds that follow it. With this, we attain the desired streamlined perspective to troubleshoot and validate the configurations.

**HDMI**

Go Back

Filter by:

CORE\_VERSION: 1.0 ☒

CORE\_NAME: dwipk\_hdmirx ☒

Show: 10 entries

Search:

Row#	DWIPK_HDMI_RX_multijob
1	UNSTABLE #147 Tags: 1.0 dwipk_hdmirx
2	UNSTABLE #146 Tags: 1.0 dwipk_hdmirx
3	UNSTABLE #145 Tags: 1.0 dwipk_hdmirx
4	FAILURE #144 Tags: 1.0 dwipk_hdmirx
5	FAILURE #143 Tags: 1.0 dwipk_hdmirx

Showing 1 to 5 of 5 entries

Go Back

First Previous 1 Next Last

Figure 4.13: Overall view of the same Project (Fig. 4.12) with filters applied



### 4.3 Metadata and Pipeline Workaround

One issue that was discovered while developing our plugin, was the incompatibility between the Metadata Plugin, from section 4.2.1.2, and the Pipeline Jobs [C1b]. This occurs because the Metadata Plugin only supports Abstract Jobs, while Workflow Jobs (created in Pipeline) are not inherited from this class, since it was created in a pre-Workflow era.

To workaround this problem, it was created a function, exemplified in the snippet 4.1, that can be patched in Pipeline Jobs groovy scripts and still maintain their complete functionality while being used in our Dashboard.

For this, one has to allow Pipeline builds to be Parameterized, as seen in figure 4.14, and add String Parameters with the following structure: **metadata:name**, where *name* can be any string the user wishes to reference the name of the metadata. This is valid because the inclusion of both Metadata and Parameters in Builds, are extended from the `Action` extension point of Jenkins. Afterwards, only this function is needed in the Groovy script.

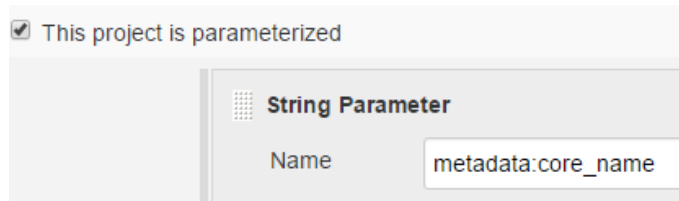


Figure 4.14: Parameterized project selection

```

1  import hudson.model.*
2
3  @NonCPS
4  def setMetadata(map) {
5      def npl = new ArrayList<ParametersAction>();
6      for (e in map) {
7          npl.add(new StringParameterValue(e.key.toString(), e.value.toString()));
8      }
9      def newPa = null
10     def oldPa = currentBuild.build().getAction(ParametersAction.class)
11     if (oldPa != null) {
12         currentBuild.build().actions.remove(oldPa)
13         newPa = oldPa.createUpdated(npl)
14     } else {
15         newPa = new ParametersAction(npl)
16     }
17     currentBuild.build().actions.add(newPa);
18 }
19 def map = [:]
20
21 //Example to populate the map:
22 map['metadata:core_name'] = 'DWIPK_HDMI_RX';

```

## The Dashboard Solution

```
23 map['metadata:core_version'] = '1.0';  
24 setMetadata(map);
```

Listing 4.1: Groovy Script Workaround

This information will be parsed inside our dashboard with a condition statement, displayed in our snippet 4.2 function, that creates exactly the same output for Pipelines as if other Job was being processed.

```
1  public ArrayList<Tag> getBuildTags(String jobName, int buildNr) {  
2      ArrayList<Tag> tags = new ArrayList<Tag>();  
3      Job job = Jenkins.getInstance().getItemByFullName(jobName, Job.class);  
4      Run build = job.getBuildByNumber(buildNr);  
5  
6      if (job.getClass().getName().equals("org.jenkinsci.plugins.workflow.job.  
7          WorkflowJob")) {  
8          ParametersAction parameterActions = build.getAction(ParametersAction.  
9              class);  
10         if (parameterActions != null) {  
11             for (ParameterValue parameter : parameterActions.getAllParameters()  
12                 ) {  
13                 if (parameter.getClass().equals(StringParameterValue.class)) {  
14                     if (StringUtils.substring(parameter.getName(), 0, "metadata  
15                         :".length()).equals("metadata:") &&  
16                         !parameter.getValue().equals("")) {  
17                         String name = parameter.getName().replace("metadata:",  
18                             "");  
19                         tags.add(new Tag(name.toUpperCase(), parameter.getValue  
20                             ().toString().toLowerCase()));  
21                     }  
22                 }  
23             }  
24         }  
25     }  
26     }  
27     }  
28     }  
29     }  
30     }  
31     }  
32     }  
33     }  
34     }  
35     }  
36     }  
37     }  
38     }  
39     }  
40     }  
41     }  
42     }  
43     }  
44     }  
45     }  
46     }  
47     }  
48     }  
49     }  
50     }  
51     }  
52     }  
53     }  
54     }  
55     }  
56     }  
57     }  
58     }  
59     }  
60     }  
61     }  
62     }  
63     }  
64     }  
65     }  
66     }  
67     }  
68     }  
69     }  
70     }  
71     }  
72     }  
73     }  
74     }  
75     }  
76     }  
77     }  
78     }  
79     }  
80     }  
81     }  
82     }  
83     }  
84     }  
85     }  
86     }  
87     }  
88     }  
89     }  
90     }  
91     }  
92     }  
93     }  
94     }  
95     }  
96     }  
97     }  
98     }  
99     }  
100    }
```

Listing 4.2: Java Condition Statement Workaround

## 4.4 Conclusion

In this chapter the Dashboard solution was fully detailed along with the fixes one must do to make it fully functional with Pipelines. It is divided in simple classes that can be used as base for future extensions to the main View.

We can safely disclose that every obstacle in the previous Spreadsheet solution, detailed in section 3.3, was surpassed:

## The Dashboard Solution

- **Traceability** - The Dashboard is automatically updated when a new test is made.
- **Susceptible to Human Error** - All the information comes from inside Jenkins, so there is no human interference in the process of displaying the information.
- **Availability** - The only requirement is being a user inside the Jenkins tool and having permission to see the View.
- **Difficulty to troubleshoot** - Every item on the Dashboard is linked directly to the original inside Jenkins, making the process a "one click" distance.

The plugin is made abstract enough so it can be used not only by our main target environment, but to help other software development teams equally.

## The Dashboard Solution

## **Chapter 5**

# **Evaluation**

This project was made with the support and constant review of the collaborating team at Synopsys, adapting every iteration towards the final goal product. For a final validation process, it was asked the IPK R&D team leader and manager short considerations about the whole project and process itself.

### **5.1 Setup**

A prototyping kit is quite a complex design in great extent due to its nature of multiple configurations. To automate the process of test compliance of the prototyping kits and with it reducing considerably the effort needed during the testing phase, jobs were implemented in Jenkins to automate the testing of the kits during their compliance testing phase. Nonetheless, the configurations to be tested scale quite exponentially and maintain their traceability can be a time consuming effort. In addition, the life time of the IPK with new versions occurring at a very fast pace makes the analysis of data extremely cumbersome, erroneous and time consuming.

### **5.2 Requirements**

The requirements for the development of the Jenkins' Dashboard were to improve the analysis, readability, usability and traceability during the lifecycle span of the IPKs. The Dashboard should arrange and consolidate the most need information related with the parameters of a specific project build, this way the product development can be monitored in a single snapshot.

### **5.3 Results**

The Dashboard developed appears to be a powerful tool to help reducing the analysis of the state of specific IPK configuration and respective version. It fills the requirements of filtering and

keep traceability of the specific relevant information of each configuration. In addition, it is very effective as it is accurate and complete in showing the defined important metadata information of the specific product configuration. Accurate, because it shows the job state of the builds to the defined information (metadata), namely product version or configuration of the image tested. Complete, since the data is well presented. The efficiency is also improved as the time required to find the successful and unsuccessful builds is now reduced. Moreover, it is easier to categorize which are the IPK product configurations that require less effort to successfully pass the compliance test during the testing phase.

### **5.4 Conclusions**

The overall Dashboard satisfies greatly all the requirements proposed, some works need to be done to improve the cosmetic aspect of the Dashboard to make it more appealing to the eye. It is now a work of the IPK team to update the jobs so that the Dashboard can be used at its full extent.

## Chapter 6

# Conclusions and Future work

In the end of this project we were able to develop successfully the intended Dashboard application.

Although narrowed down some features expected in the beginning of the project, such as displaying metrics about the hardware validation process due to time constraints between the end of development and testing of the Plugin, we believe it will greatly help development teams to keep track of their projects in a simpler and streamlined way.

### 6.1 Contributions

This Dashboard Plugin gathers all the needed information for any development team using Jenkins, for a quick glance at the status of their projects. It was designed considering people that are not used to Jenkins and its UI, so they can track this process with ease and improve the team productivity.

The fact that it is all maintained inside of Jenkins, without the need of updating any file or distributing information by other means, has the advantage of giving anyone within the system access to this information.

With the use of Metadata and applying tags to projects, creates the habit of keeping these organized for quick information access at any time.

The plugin can also be obtained within the Jenkins plugin repository, under the name **Filtered Dashboard View Plugin** [\[Far17\]](#).

### 6.2 Limitations

As referenced in the Implementation in section 4, the only limitation is the impossibility of directly addition of metadata inside a Pipeline Job, but it is provided a workaround for this issue in section 4.3.

## 6.3 Future Work

The Dashboard Jenkins plugin is completely functional and will immediately help Synopsys development teams, as well as whoever Jenkins user that wants to utilize it. However some continuous work is needed to improve it to fulfill future demands.

As any Jenkins plugin, there will be the need of maintaining it. Every new version of the tool can introduce unanticipated bugs. This is also true for the interaction with other untested plugins.

Part of the maintainability includes the improvement of the UI. Due to time restrictions, it was not put a lot of effort on it, since we prioritized the functionality of the plugin.

It would add great value as well to include some metrics inside each project (e.g. average build time, average number of builds to pass in tests). For this, new parameters should be added and calculated inside the Dashboard custom classes.

Finally, we would like to contribute even more to the Jenkins community by helping in the maintenance of the Metadata Plugin [\[RT12\]](#) so the workaround provided in section [4.3](#) is not needed anymore, and covers all types of Jobs for a total functionality of the Dashboard filtering and organizing feature.



# References

- [AF12] Fazreil Amreen Abdul and Mensely Cheah Siow Fhang. Implementing continuous integration towards rapid application development. In *ICIMTR 2012 - 2012 International Conference on Innovation, Management and Technology Research*, 2012.
- [ASV] Yael Abarbanel, Eli Singerman, and Moshe Y Vardi. Validation of SoC Firmware-Hardware Flows: Challenges and Solution Directions.
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.
- [Boo94] Grady Booch. *Object-Oriented Analysis and Design with applications*. Addison-Wesley, 2 edition, 1994.
- [CIa] Jenkins CI. Jenkins extension points. <https://wiki.jenkins-ci.org/display/JENKINS/Extension+points>.
- [CIb] Jenkins CI. Jenkins pipeline. <https://jenkins.io/doc/book/pipeline/>.
- [Far17] Pedro Faria. Filtered views dashboard plugin - jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Filtered+Views+Dashboard+Plugin>, 2017.
- [Gli] Jesse Glick. Durable task plugin - jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Durable+Task+Plugin>.
- [GPD<sup>+</sup>11] Shekhar Gupta, Msc Presentation, Prof Dr, Ir. Arjan, J. C. Van Gemund, Prof Dr, C. Witteveen, Dr. Ir, Rui Abreu, Dr. Ir, and Stefan Dulman. Antares: Automatic diagnosis of software/hardware systems, 2011.
- [HLA] Inc HDMI Licensing Administrator. Hdmi consortium, test compliance requirements. [http://www.hdmi.org/manufacturer/testing\\_policies.aspx](http://www.hdmi.org/manufacturer/testing_policies.aspx).
- [JADM] Justin Ryan, Andrew Harmel-Law, Daniel Spilker, and Matt Sheehan. Job dsl plugin - jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin>.
- [Jen16] JenkinsCI. Jenkins continuous integration tool. <https://jenkins.io/>, June 2016.
- [Kaw] Kohsuke Kawaguchi. Jenkins class view. <http://javadoc.jenkins.io/hudson/model/View.html>.
- [Lin16] Linaro. Linaro automated validation architecture (lava). <https://wiki.linaro.org/LAVA>, June 2016.

## REFERENCES

- [Ltd] SpryMedia Ltd. Datatables, table plug-in for jquery.
- [Moy] Bryon Moyer. Synopsys's ip initiative. <http://www.eejournal.com/blog/synopsys-ip-initiative/>.
- [Nen16] Oleg Nenashev. Automating test runs on hardware with pipeline as code. <https://jenkins.io/blog/2016/04/07/pipeline-for-runs-on-hardware/>, April 2016.
- [PJ15] Stephan Puri-Jobi. Test Automation for NFC ICs using Jenkins and NUnit. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4. IEEE, apr 2015.
- [PS] PCI-SIG. Pcie consortium, compliance workshop. <https://pcisig.com>.
- [PSA07] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.
- [RT12] Robert Sandell and Tomas Westling. Metadata plugin - jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Metadata+Plugin>, 2012.
- [Sco] Troy Scott. Modern fpga-based prototyping systems accelerate the transition from stand-alone ip block validation to integrated systems. <https://www.synopsys.com/designware-ip/newsletters/technical-bulletin/modern-prototyping-systems.html>.
- [SF] The Apache Software Foundation. Jelly: Executable xml. <http://commons.apache.org/proper/commons-jelly/>.
- [She15] Andrey Shevtsov. Mission control plugin - jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Mission+Control+Plugin>, 2015.
- [Son15] Mitesh Soni. End to End Automation On Cloud with Build Pipeline- The case for DevOps in Insurance Industry. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 85–89. IEEE, nov 2015.
- [Syna] Synopsys. Designware arc processor cores. <https://www.synopsys.com/designware-ip/processor-solutions/arc-processors.html>.
- [Synb] Synopsys. Designware ip solutions for pci express. <https://www.synopsys.com/designware-ip/interface-ip/pci-express.html>.
- [Sync] Synopsys. Haps® family of physical prototyping solutions - physical prototyping made easy. <https://www.synopsys.com/verification/prototyping/physical-prototyping/haps.html>.
- [SYP15] Shixiao Yan, Yu Zhao, and Ping Chen. Automated test platform for FPGA Software Validation. *2015 China Semiconductor Technology International Conference*, pages 1–3, 2015.
- [Wika] Wikipedia. Dashboard (business) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/Dashboard\\_\(business\)](https://en.wikipedia.org/wiki/Dashboard_(business)).
- [Wikb] Wikipedia. Pci express — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/PCI\\_Express](https://en.wikipedia.org/wiki/PCI_Express).

## REFERENCES

- [Wikc] Wikipedia. Phy (chip) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/PHY\\_\(chip\)](https://en.wikipedia.org/wiki/PHY_(chip)).