TEST AUTOMATION IN CONTINUOUS INTEGRATION FOR HARDWARE VALIDATION

Pedro Dias Faria

Dissertação realizada sob a orientação do *Prof. Rui Filipe Lima Maranhão de Abreu*
na *Synopsys Portugal Lda*

1. Motivação

O processo de validação de hardware está diretamente relacionado com a verificação se as diferentes configurações dos clientes serão cumpridas. Como tal, o processo de validação pode tornar-se num processo subjetivo, uma vez que envolve a avaliação de como o comportamento do hardware irá atuar nas mais diversas aplicações e condições. O processo normalmente consiste em atividades que incluem modelagem de sistemas, prototipagem e avaliação pelo utilizador[1][2].

Com esta dissertação, ajudamos a construir um ambiente de integração contínua para a validação de hardware, desenvolvendo um plugin de Jenkins em forma de *dashboard*, com o objectivo de ajudar as equipas de R&D da Synopsys no processo de prototipagem do hardware.

2. Ojectivos

A subjetividade do processo de validação em qualquer tipo de desenvolvimento de software requer o uso de algum tipo de estrutura e categorização de dados. Isto pode ajudar a um acesso mais fácil na resolução de falhas no desenvolvimento.

Com a nossa solução, a produtividade nas equipes de desenvolvimento deverá aumentar, tendo esta informação exibida de forma simplificada, precisa e direta. Para sua validação, o sistema teve que ser implementado num projeto real e recolhido feedback da equipa para se perceber se o valor gerado é visível.

Os objetivos da dissertação são:

- Definir uma estrutura automática de gestão de testes para a validação de Hardware;
- Definir técnicas para categorização e gestão dos resultados dos testes de validação de Hardware;
- Desenvolver uma aplicação web de suporte ao sistema automático de testes;
- Testar a aplicação e validar a sua utilidade.

3. Descrição do Trabalho

Uma vez que a Synopsys já tinha configurado o servidor de CI em Jenkins, foi decidido desenvolver um Plugin para esta ferramenta. Temos também a vantagem de Jenkins ser muito costumizável, com centenas de *extension points* e plugins disponíveis para suportar as nossas necessidades [3].

3.1. Casos de Uso

Tendo em consideração a organização dos projetos R&D no Jenkins, foi discutido com a equipa colaboradora que a melhor opção era desenvolver o Dashboard como um *View plugin*, implementando a classe *ViewGroup*, com o propósito de agregar diferentes Views dentro da mesma.

No diagrama de Casos de Uso da figura 1 estão representadas as ações que um utilizador pode executar para exibir o nosso Dashboard no ambiente do Jenkins.

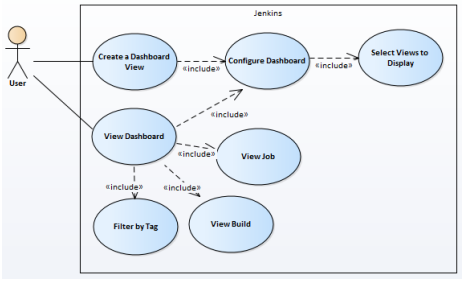

Fig. 1 – Diagrama de Casos de Uso do plugin

O plugin foi desenhado para ser tão simples como criar uma nova *View* dentro da ferramenta, selecionando quais as outras Views a serem exibidas. Depois, o utilizador pode alterar livremente a sua configuração, ou seja, adicionar, remover ou alterar quais *Views* a serem exibidas.

3.2. Jenkins Plugin - Filtered Dashboard View

3.2.1. Mission Control Plugin

Sabendo que queremos criar um Dashboard, encontramos o *Mission Control Plugin* [4], que tem praticamente toda a base para desenvolver, uma *dashboard View* completa contendo:

- Exibição dos estados dos *Jobs*;
- Histórico e filas de *Builds*;
- Extende um objeto *View*, que poderá ser adicionado ao *dashboard* principal.

Apesar de ser um ponto de partida, precisava ser moldado e adaptado às nossas necessidades.

3.2.2. Metadata Plugin

Considerando que um dos requisitos é filtrar os dados exibidos com as informações de cada configuração de teste, e uma vez que Jenkins tem a possibilidade de armazenar informações sobre cada um de seus items em formato XML, então também poderia ser armazenado algum tipo de *metadata*.

Para facilitar isso, usamos o *extension point* que o *Metadata Plugin* [5] fornece, em conjunto com a API Jenkins, para aplicar filtros relevantes em cada *Job* para uma análise mais fácil dessas informações das configurações.

Esta *Metadata* pode ser aplicada a *Jobs* simplesmente através do menu de customização, no qual aplicará a todas as *Builds* futuras.

Ou alternativamente, automatizar o processo chamando simplesmente um comando CLI de *post-build*, fornecido pelo mesmo plugin.

3.3. Server Side Implementation Design

Compreender como o *back-end* em Java interage com Jenkins e como os projetos foram organizados em *Views*, definimos uma estratégia de *Top-down* para obter e analisar as informações dos nossos items de interesse. Isto significa que tivemos que processar *View* a *View* como projetos individuais para relacionar os seus items filho no nosso Dashboard, como demonstrado na figura 2.

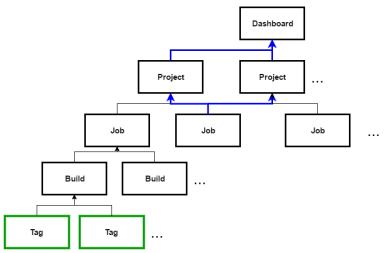

Fig. 2 – Diagrama de abordagem *Top-down*

Finalmente, a adição de *Tags* para filtrar as informações é a principal característica do nosso Dashboard. Este conceito ajuda o rastreamento de cada configuração, com o requisito mínimo de adicionar ou modificar essas classificações dos *Jobs* no *building process*.

Este design permite uma estrutura totalmente escalável, uma vez que não tem limitações no número de *Views* associados ao Dashboard, nem dos seus *Jobs* e *Builds*. Também pode ser extensível para armazenar mais informações em cada uma das suas classes personalizadas, para cálculo de métricas e indicadores adicionais no futuro, se pretendido. Todas estas informações só dependem da API do Jenkins e do que esta suporta.

3.4. Front-End Implementation Design

Para a estrutura da organização da inforação, no *front-end* do Dashboard, baseámo-nos nos mesmos scripts do *Mission Control Plugin*, descritos na seção 3.2.1, juntamente com várias bibliotecas e *Web App frameworks* comuns como Javascript, JQuery e Bootstrap, todos processados em Apache Jelly, um *engine* de processamento de scripts baseados em Java e XML que permite que a UI do Jenkins seja estendida.

Como os dados exportados para a API já estão bem estruturados, o processo de apresentação foi concluído com facilidade, exigindo apenas uma maneira de exibir a *View* de um projeto com uma vista intuitiva.

Para completar o acima mencionado, foi decidido exibir cada *Job* dentro do projeto como uma coluna numa tabela, sendo cada célula da linha as suas *Builds* organizadas em ordem cronológica decrescente. Cada célula tem suas *Tags* associadas para uma referência rápida das configurações utilizadas na respectiva *Build*.

4. Conclusões

No final deste projeto fomos capazes de desenvolver com sucesso o plugin de Dashboard pretendido [6].

Embora reduzindo algumas das características esperadas no início do projeto, como exibir métricas sobre o processo de validação de hardware devido a restrições de tempo entre o fim do desenvolvimento e testes do Plugin, acreditamos que irá ajudar as equipas de desenvolvimento a manter o rastreamento dos seus projectos de forma mais simples e precisa.

Referências

[1] 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`.

[2] Stephan Puri-Jobi. Test Automation for NFC ICs using Jenkins and NUnit. Em *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, páginas 1–4. IEEE, apr 2015.

[3] JenkinsCI. Jenkins continuous integration tool. `https://jenkins.io/`, June 2016.

[4] Andrey Shevtsov. Mission control plugin - jenkins. `https://wiki.jenkins-ci.org/display/JENKINS/Mission+Control+Plugin`, 2015.

[5] Robert Sandell e Tomas Westling. Metadata plugin - jenkins. `https://wiki.jenkins-ci.org/display/JENKINS/Metadata+Plugin`, 2012.

[6] Pedro Faria. Filtered views dashboard plugin - jenkins. `https://wiki.jenkins-ci.org/display/JENKINS/Filtered+Views+Dashboard+Plugin`, 2017.