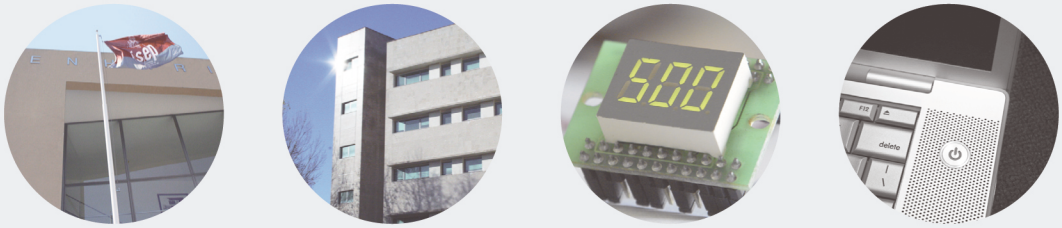
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Infraestrutura de regressões para Synopsys DesignWare DDR-PHY IP

LUCAS TUCKER
Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Politécnico do Porto
Instituto Superior de Engenharia do Porto

Regression infrastructure for Synopsys DesignWare DDR-PHY IP

Lucas Tucker

Master's Degree in Electrical and Computer Engineering
Specialization Area of Telecommunications

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

October, 2021

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the Master's Degree in Electrical and Computer Engineering, Specialization Area of Telecommunications.

Candidate: Lucas Tucker, No. 1160559, 1160559@isep.ipp.pt

Scientific Guidance: Manuel Gradim de Oliveira Gericota, mgg@isep.ipp.pt

Company: Synopsys, Inc.

Advisor: Diogo Sousa, diogo@synopsys.com

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

October, 2021

Acknowledgements

This project counted on the support and incentives of many individuals, without whom it would not have been possible and to which I will be forever grateful.

Firstly, I would like to thank the entire Synopsys team with which my internship took place, who granted me with an opportunity to carry out this project in such a prestigious organisation. Your professionalism and team spirit will not be forgotten.

To my Synopsys buddy and advisor, Diogo Sousa, for all the technical support provided during the internship as well as the tireless dedication shown in helping me solve problems that inevitably arose throughout my work.

To my supervisor, Manuel Gradim de Oliveira Gericota, for all the constructive criticisms given throughout the project and, in particular, for the remarkable and quick availability provided in clarifying all of my doubts and questions.

To my colleagues and friends who accompanied me throughout my academic journey. I have no doubt that without you all I would not have reached the point at where I am today.

To my girlfriend, for the constant motivation and inspiration given, which continues to drive me to this day in reaching new goals and achieving new heights.

Last but not least, I would like to thank my parents for the unconditional support given to me in every facet of my life. Your love, effort and devotion invested in me, independently of my career or study choices, made it possible for me to be here.

Abstract

Verification regarding integrated circuit designs is an essential step in assuring that the intent of these designs is preserved in their implementation, thus meeting their initial specifications. In this context, regression testing is used in order to assert the working order of newly introduced features in a design and assure previously implemented functionalities have not been hindered. However, regression testing often generates vasts amounts of data, including failures, calling for a process in which to appropriately categorise and prioritise these so as to enable the most effective means of fixing them. This process is known as failure triage.

In this project, a developed regression infrastructure for regression running and triaging purposes is presented. This aims to surpass the limitations associated with the previous solution employed at Synopsys, which possesses an in-house nature and, consequently, involves maintenance efforts on the company's side in order to keep its infrastructure up to date and in working order. The proposed solution involves the application of a standardised regression test execution tool in VC Execution Manager. The developed work was executed alongside this tool, where customisations regarding its user interface were performed in order to provide the same functionalities as with the previous solution. Validation aimed at the developed infrastructure was executed by means of testing the customised interface's functionalities, upon obtained regression results stemming from regression runs completed with the Execution Manager tool.

Keywords: Builds, Coverage, Design flow, Execution Manager, Failure triage, JavaScript, Python, Regression testing, Regular expressions, Test case, Testlist, Verification, Verification Plan.

Resumo

O processo de verificação no projeto de circuitos integrados é um passo crucial em garantir que as funcionalidades destes projetos são preservadas na sua implementação, cumprindo as especificações iniciais. Neste contexto, o uso de regressões é aplicado de modo a averiguar o correto comportamento de novas funcionalidades integradas no projeto, bem como assegurar que a integração destas não tenha prejudicado o comportamento de funcionalidades anteriormente presentes. Porém, a aplicação de regressões é caraterizada pela vasta quantidade de dados criada, os quais incluem falhas no projeto. Com base nisto, a correta categorização e priorização destas falhas torna-se importante de modo a alcançar a forma mais eficaz de as solucionar. Este processo é alcançado por intermédio de triagem.

Neste documento, é apresentado um projeto no qual foi desenvolvida uma infraestrutura que viabiliza a execução de regressões para posterior vizualização de resultados e sua aplicação em contextos de triagem. Este projeto visa ultrapassar as limitações relativas à solução prévia empregue na Synopsys, desenvolvida para uso interno na empresa e que, consequentemente, requere manutenção de modo a manter a infraestrutura funcional e atualizada. A solução proposta envolve a utilização de uma ferramenta standardizada para a execução de regressões, designada VC Execution Manager. Desta forma, o trabalho realizado foi baseado na utilização e alteração da interface desta ferramenta de modo a incluir nela todas as funcionalidades presentes na solução prévia. Validações efetuadas a toda a infrastrutura desenvolvida foram executadas por intermédio de testes à interface alterada e às funcionalidades nela introduzidas. Isto foi alcançado após obtidos resultados derivados de regressões, as quais foram executadas com a designada ferramenta.

Palavras-Chave: *Builds*, *Coverage*, *Design flow*, Execution Manager, JavaScript, Lista de testes, Plano de verificação, Python, Regressões, Expressões regulares, Testes, Triagem, Verificação.

Contents

List of Figures

List of Tables

List of Acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CAD	Computer-Aided Design
CSV	Comma Separated Value
DUT	Design Under Test
EMan	VC Execution Manager
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HTML	Hypertext Markup Language
IC	Integrated Circuit
ISEP	Instituto Superior de Engenharia do Porto
IT	Information and Technology
JSON	JavaScript Object Notation
RTL	Register Transfer Level
SGE	Sun Grid Engine
SSI	Small Scale Integration
UI	User Interface
URL	Uniform Resource Locator
UVM	Universal Verification Methodology
VLSI	Very Large Scale Integration
YAML	Yet Another Markup Language

Chapter 1

Introduction

1.1 Contextualisation

This project results from a proposal presented by Synopsys, upon my interest in partaking in an internship there. The team where the internship took place focuses on the verification of chip designs produced by the company, a process which is performed in order to guarantee that design functionalities are in working order and satisfy the requirements demanded by clients prior to fabrication and dissemination. This is achieved through digital simulation runs to validate designs at the Register Transfer Level (RTL) level using test strategies aiming at exercising and verifying all possible features and their integration. Therefore, the main goal lies in achieving total coverage of designs by eliminating any existent discrepancies between the initial requirements and the final product.

1.2 Motivation

The current verification setup at Synopsys relies on tools which aid engineers in the several tasks involved with the triage process. Triage is a procedure in which levels of priority are assigned to different tasks or individuals in order to determine the most effective approach in which to deal with certain problems and issues within a certain environment. These tools, despite fulfilling the intended goals and needs of the team, were developed for in-house use and, thus, involve maintenance on the team's side. Therefore, integrating new features in these tools and repairing

possible flaws that may arise require a specific engineer in order to perform the duties associated with these jobs. Consequently, the team's time, effort and resources are diverted towards these tasks.

The presented proposal corresponds to a project aiming to develop and test a new regression infrastructure solution to allow for result extraction and gathering. Furthermore, the project also aims to work alongside bug trackers and project management tools for future integration in the team's verification flow. The main advantage of the new proposed regression infrastructure setup to be developed is the complete lack of maintenance required, unlike with the previously used solution. This results from the fact that the new infrastructure is built upon an outsourced tool as opposed to the previous in-house setup. Therefore, regression infrastructure maintenance on the verification team's side is no longer required, since the outsourcing of the tool implicates that these duties are now delegated to a particular team assembled for the job. Another advantage regarding this outsourced tool is its standardisation in the industry, being used by several teams and companies outside of Synopsys for design verification purposes. Thus, time and resources previously expended towards maintenance tasks may be put to use in the verification process.

1.3 Goals

Since the main objective of the project lies in the development and test of a regression infrastructure solution, the use the outsourced design verification tool, namely VC Execution Manager, was proposed. Therefore, in order to aid the attaining of the previously mentioned objective, a coherent work plan was devised. This plan is subsequently outlined:

- analysis of regression environments/tools;
- specification of regression requirements;
- implementation of regression infrastructure;
- verification of implemented features;
- documentation and presentation.

The first goal involves the familiarisation with the preexisting regression environment as well as the exploration of the VC Execution Manager tool. Next, a list of regression features for the infrastructure is formulated including integration requirements with external tools. The subsequent step corresponds to the implementation of the new regression infrastructure based on the VC Execution Manager as well as the integration with external tools previously mentioned. Subsequently and parallel to this last goal, verification of the implemented features is done, focusing on the

debugging and fixing of potential implementation issues. Finally, documentation on the project activities and results are written as well as the presentation to the relevant personnel.

1.4 Report structure

This report is organised according to a coherent order of contents, in order to allow an unequivocal association of the addressed subjects. In Chapter 2, a literature review is presented, addressing several relevant topics regarding the design and verification of integrated circuits. In Chapter 3, the problem at hand is characterised and, so as to solve the issues regarding it, requirements regarding a new proposed solution are specified. Chapter 4 reveals the overall solution's architecture through means of a block diagram, detailing the purpose for each of its integrating elements in accomplishing the established requirements. In Chapter 5, the implementation of the project is addressed, detailing all the tasks performed in order to obtain the desired results, which are subsequently focused on in Chapter 6. Finally, conclusions and future work respecting the developed project are present in Chapter 7.

Chapter 2

Literature Review

This chapter focuses on giving an introduction to the design and verification process. First, a generic design flow is explained with respects to phases belonging to the frontend or backend flow. Subsequently, the verification process is addressed, in regards to its importance and overall purpose, followed by an analysis regarding the basics of the testing process. The final section of this chapter aims to study the failure triage process, its connection to regression testing and the tasks it comprises.

2.1 Design Flow

Throughout history, the Integrated Circuit (IC) industry has undergone a series of transformations and revolutions in the way chips are designed. Since the late 1950s, these circuits have experienced an increase in size and complexity, leading to several IC generations of scale integration, from Small Scale Integration (SSI) circuits up to the current Very Large Scale Integration (VLSI) circuits. While the former solely included up to a dozen gates, the latter and present generation of VLSI circuits hover the 100 million gate mark, where it is proclaimed that approximately a third of these modern designs surpasses this value [4].

During its production, a digital IC design goes through a variety of different phases in which it is subjected to a number of transformations, which allow it to progress from an original set of specifications and develop into the desired final product. The entirety of the previously mentioned phases constitute what is known as the circuits' design flow. Figure 2.1 illustrates a conceptual and somewhat abstract diagram of this flow, following a simplified top-down framework.

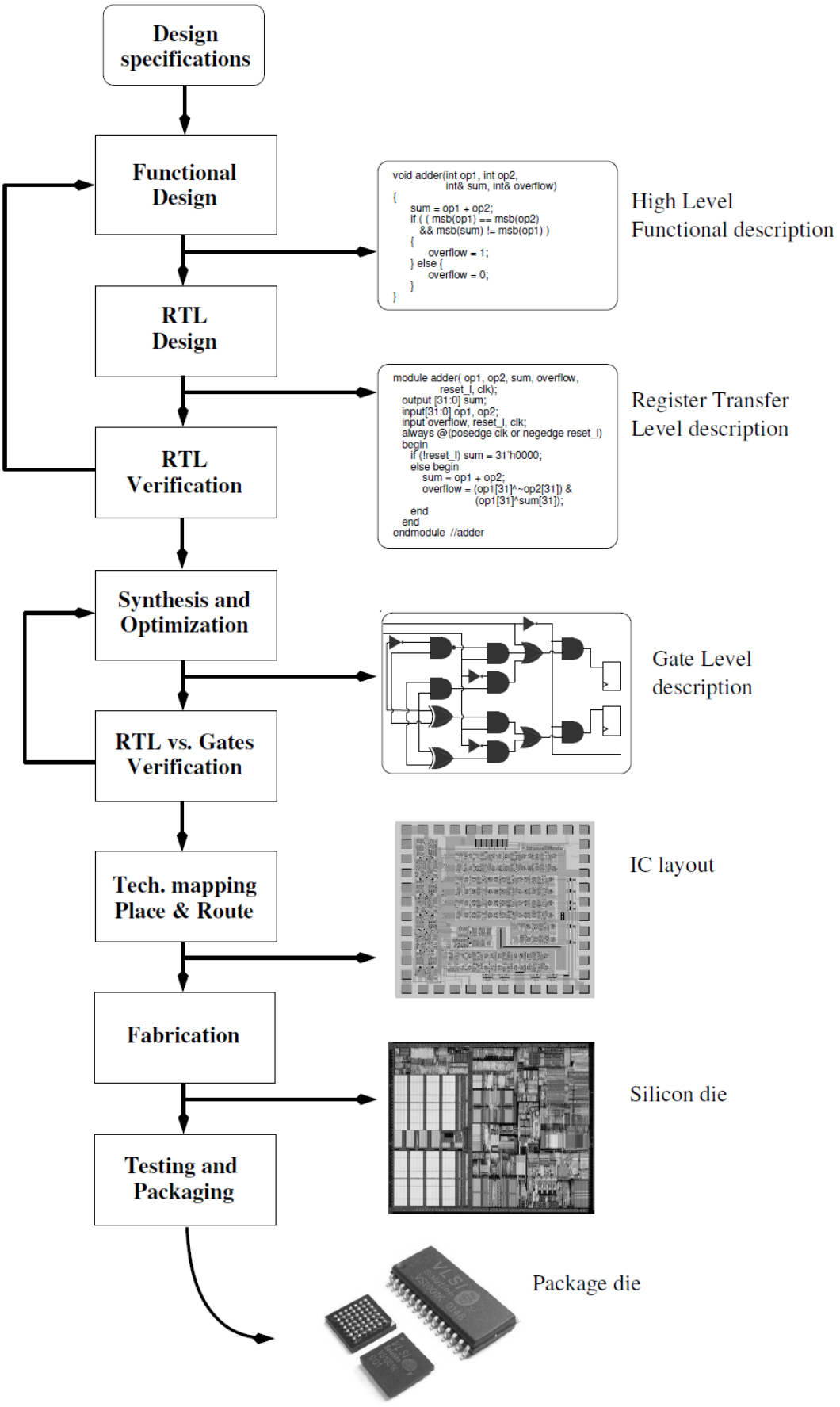


Figure 2.1: IC design flow [1]

In reality, the industrial development of the digital design from specification to final product is much more complex and challenging, requiring a multitudinous of iterations through various phases of the flow. Each of the modifications experienced at each of these phases corresponds, coarsely, to a different description of the system, which grows in detail the further down it finds itself in the flow. In the end, the conceived design takes shape according to the initial specification requirements regarding functionality, area, timing power and cost. As will be covered in more detail in Subsections 2.1.1 and 2.1.2, the design flow is divided into two separate parts, namely the frontend design flow and backend design flow, which together allow the creation of a conceptual IC from scratch to production [1, 6].

2.1.1 Frontend flow

The frontend design flow comprises all the stages from the design specifications up to the RTL Verification phase of the IC design flow. Therefore, the main responsibility of this section of the flow is to establish a solution to a given problem/opportunity and transform it into an RTL circuit description. It is also technology independent, meaning that this section of flow is follow whether you are designing Application-Specific Integrated Circuits (ASIC) or Field Programmable Gate Arrays (FPGA), for example. A brief description of each of the phases that embody the frontend flow are subsequently given, following the order displayed in Figure 2.1:

- design specifications - This stage, as is common with any project, rises from the need to solve a problem, capitalise on an opportunity or simply to improve upon a given process/product. Generally, design specifications are presented as a document outlining a set of functionalities that the final solution should provide, as well as a number of constraints it must abide by. This document then becomes the basis for the next stages of design along with verification work.

- functional design - Following the given specifications, the next step encompasses the deriving of a potential and feasible solution from these, which, in other words, is termed functional design. This process, also referred to as modelling, aims to architect the system in high level modules, each of which possesses a well defined functionality for the overall design, as well as determine the way they communicate mutually through well defined input/output interfaces and communication protocols. Among the results of this design phase is a high level functional description, often a software program in C or similar programming language. Additionally, lower level design details about each functional block implementation are designed, amounting to the micro-architecture design. Given the large scale of the problem at hand, this stage is

commonly accomplished by way of a hierarchical approach, enabling designers to focus on a certain portion of the model at any given time.

- RTL design - Subsequent to the functional design, we find the RTL design phase. It is here where the previously mentioned micro design is described using a Hardware Description Language (HDL) such as Verilog or VHDL. Each module functionality is then converted into constructs specific to the particular language in use, so that further on in the flow the RTL model can be input into a synthesis tool, whose responsibility is to map the design into a gate level netlist. The RTL model developed in this phase is used as a golden reference, meaning that it is against it on which all gate level simulations shall be compared to.

- RTL verification - Once the RTL design of the system is completed, the RTL verification stage begins. In this stage, further discussed in Section 2.2, the functional aspects of the design are simulated with the end goal of acquiring a reasonable amount of confidence that the final circuit will function correctly, this is, according to the initial specifications. In order to test if the prior developed RTL code meets the functional requirements, every block is verified, whereby and whenever each time errors are found, the model needs to be modified to embody the specified behaviour. But when every block meets its specifications, they are integrated in a top-level to verify the functionality of the entire system. For this purpose testbenches are used, composed of numerous suites of tests which stimulate the block or top-level functionality via test vectors in order to ascertain no errors are detected and no incongruities relative to the original set of specifications are present, which otherwise lead to updates in the RTL design. Thus, the underlying purpose for the RTL verification stage is to remove all possible design errors before advancing to the expensive chip manufacturing process.

2.1.2 Backend flow

On the other hand, contrarily to the frontend flow, whose objective is to obtain a circuit description, this portion of the flow mainly targets the physical implementation of the circuit and thus is technology dependent. The backend design flow spans from the synthesis and optimisation stage up until the fabrication of the chip:

- synthesis and optimisation - In the next phase of the design flow we find synthesis and optimisation of the RTL design. The overall intent in this stage is to convert the earlier RTL model into a further detailed description of the circuit, known as a gate level based netlist, which described the design in terms of its basic logic components and memory elements. This circuit description

is optimised in order to meet a number of design constraints regarding power consumption, timing and IC area, an increasingly challenging activity with the rising complexity of designs.

- RTL versus gates verification - This stage, commonly known as equivalency checking, is apace with synthesis and optimisation. Optimisation of the circuit description, as previously stated, is a difficult process which in turn may entail new functional errors that require additional RTL verification. The purpose here is then to guarantee that no errors have been introduced during the synthesis phase. Consequently, the pre-synthesis RTL description is compared to the pos-synthesis gate level description in order to assure equivalence between both models.

- technology mapping and place & route - This backend stage converts the gate level netlist produced during synthesis into a physical design, this is, a description of the circuit in terms of geometrical layout. Although the name suggests for two phases, this stage can be divided into five distinct phases: design planning, placement, clock tree synthesis, routing and chip finishing. Finally the design is fabricated and the chips are tested and packaged, fit for customer use.

The presented design flow is, as mentioned, a theoretical approximation as to what occurs in reality. Unpredictable changes or overlooked aspects related to the design specifications, as well as the detection of incongruities and errors during RTL verification imply multiple iterations of synthesis and, as a result of this, each new version of the design goes through the same subsequent design phases. Additionally, dealing with the increasing demand for designs with faster clock cycles forces engineers to push the boundaries of designs, requiring ever further optimisations.

2.2 The Importance of Verification

Verification, as was explained in Subsection 2.1.1, refers to the process of determining whether or not an implementation of an IC meets its initial specifications. In other words, it is a means to assure the intent of a design is preserved in its implementation and that the hardware model fulfils the requirements. It is often misconceived that the purpose of verification is to find bugs and errors in the design, a partially correct statement. In reality, the end goal is to make sure that the device's functionalities satisfy the prerequisites, performing its tasks as intended. The errors and bugs found along the way are the stepping stones toward this goal, symbolising discrepancies between the design and what is expected of it [2].

Figure 2.1 shows the RTL verification stage as an isolated phase of the design flow. However, in practicality, verification takes place throughout the evolution of

the product. Verification activities are commonly conducted alongside the design creation process, often lasting until the circuit layout. Initially, a designer reads the hardware specification for a block, interpreting the human language describing it, then creates the corresponding logic, usually in the form of RTL code. In this context, given the constant changes in the RTL to accommodate for new functionalities, verifiers need to assure that [7]:

1. the new functionality is in working order;
2. no damage to previous implemented design features has been done.

Thus, whenever a part of a design is altered, verification not only needs to determine if the changes are working but that these have not broken other working pieces. Hence, the correctness of IC designs are a major factor to consider, often being achieved through a strenuous RTL verification process. This laborious activity is motivated by the increasing costs of IC manufacturing, where functional errors must be prevented from escaping to the tape-out stage, given that consequences of flaws going unnoticed in system designs until after the production phase are dauntingly expensive. However, in today's era of multi-million gate ICs, validating their functionalities is an increasingly challenging task due to the growing complexity of the designs [1, 8].

Indeed, in the process of design and verification, it is the latter task which has been shown to dominate the time scales, with reports stressing that verification poses a significant bottleneck in productivity, occupying up to 70% of the modern design flow cycle's efforts and resources [1, 9, 10, 11, 8, 12]. Given the magnitude of effort required by the verification challenge, coupled with the frequent shortage of verification engineers, it comes as no shock that verification often finds itself in the critical path of many projects. Moreover, verification is oftentimes thought about upon the completion of the design and after the integrity of the planned schedule has been violated, escalating the already arduous problem. With this scenario in mind, verification can be seen as a necessary evil. On the one hand, it is indispensable in the assurance that a design is functionally correct in order to conceive a marketable product, providing benefits that customers require and in turn, generating revenues. On the other hand, verification in itself is not the one directly generating these dividends. The design that is verified is the one that is sold and that will ultimately make money.

Another note regarding the verification process is that, from a theoretical standpoint, it is never truly complete. This derives from the fact that one cannot prove that no discrepancies between the design and its requirements are left. New design errors will be found granted enough time and therefore, ensuring a completely error-free design is not possible [2, 10]. But, as time spent on verification increases, fewer

errors are detected given the same amount of effort, meaning that as the process advances, rewards associated with it decrease, costing more to find each remaining error. Consequently, the issue thus becomes if the severity of a single error justifies the effort spent working on it.

2.3 Testing

The testing process is often confused with verification. While the purpose of the latter, as previously detailed in Section 2.2, is to ensure that a design meets its functional intent, testing aims to determine if the design was correctly manufactured without defects, according to a specific implementation. Testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results. Thus, to test means to compare an actual result to a predefined standard [10, 12, 13, 14].

2.3.1 Testbench and test suites

In order to test a certain design a verification environment is used, more commonly known as a testbench. Figure 2.2 illustrates a testbench and its interaction with a Design Under Test (DUT).

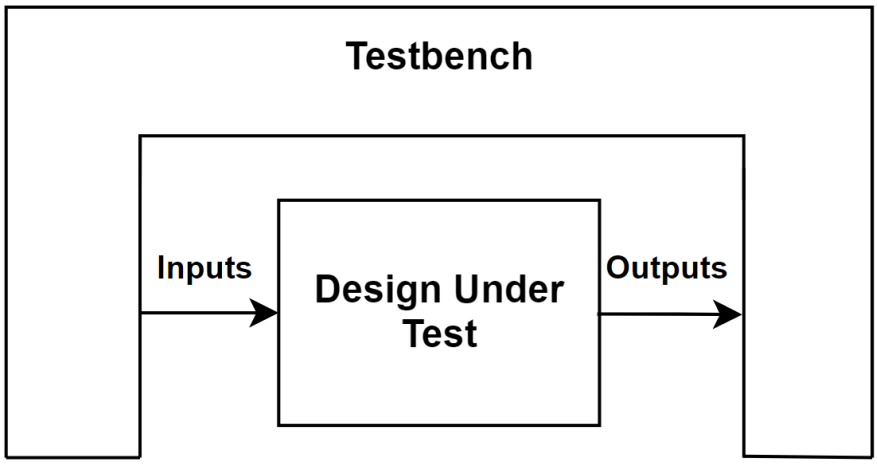


Figure 2.2: Testbench and Design Under Test (adapted from [2])

The testbench usually refers to simulation code, which wraps around the DUT and works over a wide range of levels of abstraction to determine its correctness. This is accomplished through the following steps [2]:

1. generating stimulus;
2. applying stimulus to the DUT;

3. capturing the response;

4. checking for correctness;

5. measuring progress against the overall verification goals;

Therefore, the testbench is a complete verification environment formed by several constituents, providing predetermined input sequences of bits (called test vectors) to the design and observing the outputs given. However, these input sequences are determined through test cases, which use the infrastructure provided by the testbench to achieve the verification objective [7, 10]. This is accomplished by driving the design using stimuli, constraints to restrain the inputs to legal values (generating the scenarios of interest) and other tools for monitoring progress and validating outputs. The objective of these test vectors is not to exercise functions but to exercise physical locations in the design in order to assess if everything is in working order. Consequently, the challenge is to determine what input sequences to supply to the design so that most of the design's locations are tested and working correctly. For the sake of this challenge, test suites are used, which consist of a set of two or more test cases aiming to exercise as many physical locations of the DUT as possible.

2.3.2 Coverage metrics and the Verification Plan

The quality of a test case belonging to a suite is then determined by the number of locations it is capable of testing, which is measured through coverage. And so, coverage can be defined as the ratio of physical locations tested to the total number of such locations in a design [7, 10]. Thus, the goal to create an effective test suite is to satisfy given coverage metrics. Given this goal and the constant changes to the RTL code responsible for the addition of new features in a design, tests can be characterised against the coverage they provide in the following way [7, 15]:

- obsolete;
- redundant;
- retestable;

Test cases are designated as obsolete if they are no longer valid to the modified program, becoming irrelevant to the verification process since they generate coverage below a certain threshold. Redundant test cases solely provide coverage on functionalities unrelated to the changes made in the design and, although they comprise valid test cases (i.e., not obsolete), they can be excluded from a test suite without jeopardising the quality of testing. Lastly, retestable test cases are those which exercise affected areas of the design and should be run after modifications, contributing to coverage metrics in a significant manner.

With this being said, certain verification goals may be to satisfy certain coverage metrics (or other metrics) at a specific level and with an acceptable likelihood. These goals are defined in the verification plan, a document containing several items organised in various sections defining the verification scope [3]. These items are obtained through the design's specification document, through which detailed features are identified and listed, being mapped to quantifiable metrics (such as coverage) so that the verification progress can be observed. The verification plan is essential in the process of verifying the designs since it contains specific details on the methods and resources used along the way as well as the work which needs to be accomplished. The verification plan is also responsible for settling controversies regarding discrepancies found between responses expected by the testbench and the ones produced by the DUT. Finally, the RTL code is said to be an accurate representation of the specifications once all of test cases have been surpassed and the satisfactory levels regarding the defined metrics have been achieved [10].

2.3.3 Regression testing

In the context of software maintenance, this is an activity which inherently includes modifications such as optimisations, error corrections and the deletion of obsolete capabilities. And, no matter how well conceived and tested the software is before being released, it will eventually require altering in order to perform one or more of these modifications. Such modifications may cause the software to work incorrectly and/or may affect previous sections of developed software [16].

In the case of the previously discussed RTL code this is no exception. The RTL code suffers constant transformations during the life cycle of the design and, in order to guarantee the integrity of the design as the RTL is worked on, not only must it be assured that newly introduced functionalities behave as expected but also that these functionalities have not adversely affected previously working functions. Verifying directly modified sections of code is carried out with resolution tests, while unchanged portions that may be affected by the code changes are handled through regression testing.

Therefore, regression testing is a process to verify if proposed changes made while enhancing or fixing the design have not introduced any defects by obstructing or influencing the previously validated code's functions. In other words, it is the process of testing or retesting a system to verify that modifications have not caused unintended effects and that the system still complies with its specified requirements [3, 7, 15, 16]. Consequently, the regression testing process involves a repeated execution of determined suites of tests every time the design's RTL changes. Figure 2.3 illustrates the regression testing scope process within a metric driven verification flow, granting a global view of the verification process. Additionally, it encompasses

all the components previously detailed in this section in order to provide a better understanding of the mentioned flow.

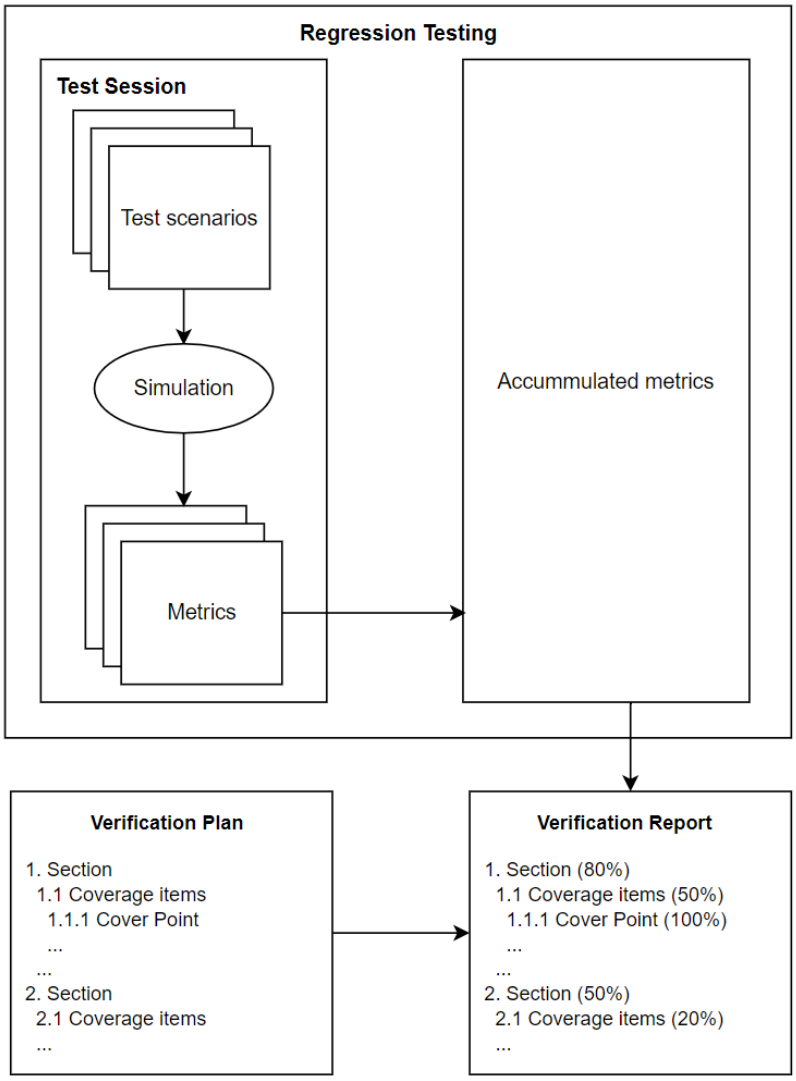


Figure 2.3: Regression testing and metric driven verification flow [3]

Firstly, within the regression testing domain, a test session may be run in which several test cases producing different scenarios of interest are submitted to simulation through means of a testbench. From here, metrics such as coverage are generated but, as is the case with regression testing, often several test cases need to be rerun and simulated again in order to achieve the desired metric based goals. Thus, the obtained metrics from completed tests, which correspond to the verification plan elements, are accumulated. Finally, once coverage goals are achieved, thus completing the testing, the verification report containing metric data mapped to the verification plan is generated and verification closure is achieved.

2.4 Failure Triage

Over the past years, regression testing has experienced a dramatic boost in capabilities as a result of several developed verification techniques and tools which aid in this process to an exceedingly great extent. However, regressions often generate vasts amounts of data, including failures, which need to be dealt with and eventually debugged. The amount of failures exposed by each regression run is frequently extensive, requiring an emerging need to appropriately categorise, prioritise and distribute them to the correct personnel for detailed analysis. This process, whose role is illustrated in Figure 2.4, is commonly know as failure triage [4, 8, 17] and is an essential pre-processing debugging step.

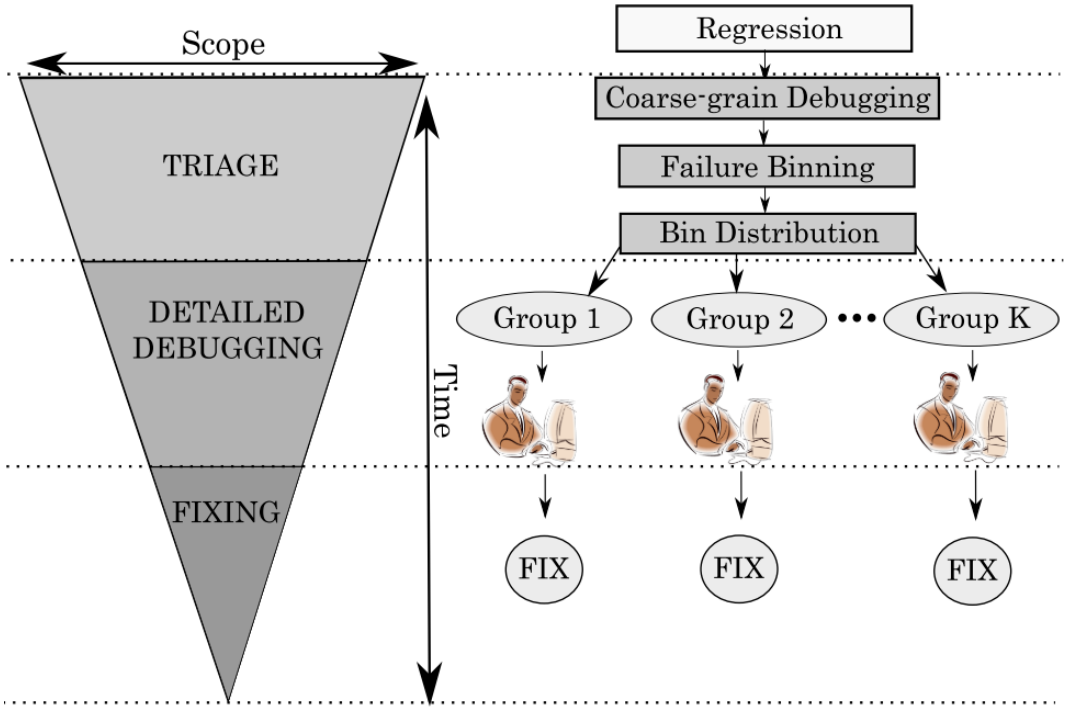


Figure 2.4: Triage in design debugging flows [4]

Broadly speaking, verification can be performed either one of two ways: on-line or in regression mode. In the case of on-line verification, each failure is analysed in an individual manner to determine potential design flaws that could be the cause for certain failures, a process coined as detailed debugging. On the other hand, failure triage is the process that follows regression testing, consisting of a high level debugging task that has a twofold purpose. Firstly, it deals with newly found failures, seeking to group (bin) these together based on their root causes. Secondly, with the aid of these groups, it identifies and assigns the most suitable engineer to perform detailed debugging on each of these and provide a fix.

As depicted in Figure 2.4, failure triage commences after a regression is complete and precedes detailed debugging, being performed through three main tasks [4, 8]:

- coarse-grain debugging;
- failure binning;
- bin distribution.

Coarse-grain debugging is the first failure triage sub-phase. It constitutes of an early stage analysis with two goals in mind: gain an idea of the approximate location of the root causes responsible for each of the exposed failures in the design as well as generate a ranking based on priority for each of these potential causes. The latter involves identifying the design locations most likely to be the actual source for a given failure. Consequently, a higher or lower priority associated to a certain root cause shall have a higher or lower likelihood of being the culprit for the failures in question, respectively. This sub-phase can considerably reduce the engineering effort required for detailed debugging and serves as a means of generating error signatures which aid in the subsequent two triage sub-phases. The next triage sub-phase is failure binning. The meaning behind the name of this sub-phase lies in the fact that the main objective is to group together individual failures into what are called bins. This goal is accomplished through leveraging information from coarse-grain debugging to discover correlations between failures and, based on the similarities between any two failures, bin these together if they are likely to share the same error source or bin them separately if the opposite situation occurs. The final task of triage is failure bin distribution, which usually starts immediately after the binning stage has finished. Bin distribution, as the name suggests, aims to allocate these bins for debugging, assigning these to the best suited engineers for the job. Ideally, the individuals appointed are the ones most familiar with the failures within the specific bin to which they have been assigned and, from this point on, detailed debugging is performed, consisting of a detailed root cause analysis in which the cause of the erroneous behaviour is pursued through a set of design locations. These design locations are able to provide vital suggestions regarding as to where the actual error lies in the design, leaving engineers with the job of identifying the exact location of the error and eventually fixing it. Any fix provided by each of these engineers can potentially eliminate multiple failures belonging to the same bin, as its common that different failures are caused by the same design error.

Organising the number of failures exposed by regressions is a time consuming process and, despite being declared a fast growing regression problem in the industry due to its resource intensive nature, triage solutions remain a predominantly manual process. With current studies indicating that triage may occupy up to 30% of the debugging efforts [4, 17], the semiconductor industry is on the constant lookout for

Computer-Aided Design (CAD) solutions to automate the triage process. Conventional in-house triage techniques often depend on scripts which rely on parsed error messages in order to group the observed failures. Alternatively, a single engineer can be appointed to monitor and analyse error on a daily basis with the intent of assigning other engineers for further debugging. On the one hand the scripting approach frequently fails to detect failures sharing the same root cause. On the other hand, the manual nature of binding a specific engineer to the triage task entails significant time costs and counts on the engineer's intuition and knowledge of the design's behaviour. Often, links between these failures and their root cause are hard to identify, causing confusion among the engineering design and verification teams, since the designs erroneous behaviour is transparent to some but incomprehensible to others resulting in failures being constantly re-assigned to various engineers until the most suitable one is found. This may also cause inaccuracies in the tasks of binning and bin distribution and therefore, if failures are not appropriately categorised into smaller related groups, then multiple engineers could be investing efforts into resolving the same design errors and thus wasting valuable resources.

This Chapter aimed to address several topics regarding the design and verification of integrated circuits. Firstly, a general design flow was presented, addressing the different phases that enable a design to evolve from an initial set of specifications up until its fabrication. The importance regarding the verification process was also talked about, in that it determines whether a design meets its functional intent by assuring new functionalities work and that these have not hindered previously implemented features. This, as was seen, is accomplished through regression testing, which uses a series of metrics such as coverage in order to satisfy delineated goals present in the verification plan. In the last topic, failure triage was discussed, consisting of a method to prioritise failures found while running regressions in order to find the most effective way in which to deal with these issues.

Chapter 3

Problem Characterisation and Solution Requirements

This chapter addresses the problem at hand associated with the current solution employed by the company and the aspects it encompasses. Furthermore, upon a detailed analysis of this solution, a series of requirements is listed, which are considered in order to develop and take part in the new solution.

3.1 Problem at hand

The current solution used by the verification team at Synopsys involves the use of a central database dedicated to managing regression data. This database, allied with a Web User Interface (UI), allows the team's developers to easily access that data. Here, it is possible to consult and analyse information regarding triage purposes by means of a failure signature table, similar to the one illustrated in Figure 3.1.

Failures	Test name	Signature	Triage	Assigned To	JIRA ID	Comments
10	Test #1	Signature_X	Triage	Person A	JIRA_LINK_123	Checking
2	Test #2	Signature_Y	Triage	Person B	JIRA_LINK_456	Nearly solved
12	Test #3	Signature_Z	Triage	Person C	JIRA_LINK_789	Testing issue

Figure 3.1: Current Solution's Failure Signature Table

As seen, the failure signature table possesses a number of different columns, each of which exhibits a particular piece of information, granting developers with the necessary knowledge to fix specific design errors found at the RTL level. Therefore, in order to gain a better understanding regarding the purpose of this table, we must analyse the purpose of each table column individually:

- Test name - contains the name of a test case applied in order to stimulate the design and exercise its features;
- Signature - provides a general string or pattern which many different errors may match, allowing separate errors with the same suspected root cause to be binned together;
- Failures - contains the number of failures obtained in that specific rows's bin;
- Assigned To - displays the engineer/developer to whom the current row's bin was assigned;
- JIRA ID - comprised of a hyperlink to a project management/bug and issue tracking software tool. In this case, that tool is named JIRA;
- Comments - column for any additional comments related to that row's issue;
- Triage - column containing a single triage button which, when pressed, redirects the user to a new page where triage information may be modified.

In the case of the last column mentioned, the new page to which a user is redirected contains a new table, analogous to the one presented in Figure 3.2.

Signature	Signature_X
Test name	Test #1
Assignee	Person A
Assign to	Assign to ▼ Email assignee
Current status	Assigned
Alter status	Status ▼
JIRA ID	JIRA_LINK_123
Comments	Checking for bugs

Figure 3.2: Triage information for a given signature

As stated above, this table grants the ability to manage triage information associated with a particular signature of the failure signature table so that emails may be sent and data such as assignees, issue statuses, bug tracker hyperlinks and comments may be altered when required. However, although this solution contains all features and functionalities required for the verification process, it possesses an in-house nature, meaning that it was specifically developed for use within the verification team at Synopsys. Consequently, it does not consist of a tool widely used across the verification industry and does not enable for standardised verification practices. Moreover, due to its application in an in-house manner, maintenance associated to this solution demands effort on the company's side so as to keep its infrastructure in working order and up to date with the current projects. Hence, in order to surpass these limitations, a new solution must be devised, including all the mentioned functionalities of the current solution. Therefore, the previously mentioned functionalities constitute indispensable requirements in the solution to be developed, so as to continue satisfying the needs of the verification team.

3.2 Solution requirements

Keeping in mind the motivation explained in Section 1.2, it is important to note that the main objective to achieve with this project involves the setup of a new regression infrastructure to aid with the verification process and its associated triage tasks as well as solve the problems encountered in the previous solution. Furthermore, continuing the thought process with which Section 3.1 was ended, and having analysed the current solution's features and components, it is now possible to form a list of requirements to constitute the new solution's regression infrastructure. These requirements include:

- allow for simple and efficient regression running;
- produce error signatures based on test errors;
- assign error signatures to specific developers;
- assign error signatures to a bug tracker hyperlink;
- allow on the fly changes to the previous two items;
- integrate email functionalities;
- allow for a straightforward switch between solutions.

The project must then possess an uncomplicated approach to regression running in order to allow subsequent data to be analysed and acted upon for triage purposes. Secondly, the solution must be able to output error signatures based on the test

errors. These signatures, as mentioned in Section 3.1, consist of general strings which many test error messages may match. The purpose of these matches is to allow for binning of errors which have a high probability of possessing the same root cause, thus allowing for their fixing process to be conducted in a more efficient and effective manner. Additionally, the project should enable error signatures to be assigned to specific developers of the company as well as bug tracker links, allowing for these assignments to be modifiable on the fly (i.e: through a provided UI). Furthermore, the integration of email functionalities is also a crucial requirement for the solution, since notifications and updates regarding the issues to be fixed may be sent to developers through this mean. Finally, the developed infrastructure should allow for a straightforward switch between the previous and proposed solutions. Therefore, not only should the new solution provide the same triage functionalities but also present the regression information in a similar manner as with the former infrastructure. This last requirement aims to diminish the time required for the verification team to become acquainted with the newly developed infrastructure, enabling a more dynamic and productive transition between tools.

This Chapter mainly focused on the characterisation of the problem which the project aims to resolve, as well as the requirements to be integrated in the new solution so as to achieve this goal. It was seen that the main issue regarding the previous solution was its in-house nature which, consequently, lead to required maintenance efforts in order to keep the infrastructure up to date. Thus, requirements for a new solution based on a standardised tool were devised in order to enable effective regression running as well as allow for its straightforward adoption by replicating the previous solution's interface as much as possible.

Chapter 4

Proposed Solution

The present chapter focuses on the structure of the proposed solution. Firstly, in order to gain a general understanding of the solution, its architecture is illustrated in the form of a block diagram with all its main constituents as well as the links existent between them. Secondly, the purpose behind each block is explained in further detail as to clarify its need and, consequently, its presence in the established architecture. Finally, having understood the purpose for all the constituent blocks, it is demonstrated how the proposed solution is able to meet all the intended requirements demanded of it.

4.1 Solution architecture

Having presented the problem at hand with the current solution, specified the main characteristics composing this solution's infrastructure and interface as well as listed the main requirements needed going forward in order to solve the previously encountered issues, it is now possible to delineate an architecture for the new regression infrastructure. This architecture, which is based on a widely used tool for regression environments, is presented in Figure 4.1 in the form of a block diagram. It encompasses all aspects of the new proposed infrastructure by putting nine main components to use and associating them together. These elements will be addressed in Subsections 4.1.1 through 4.1.5 in order to establish their functionality and need within the new solution.

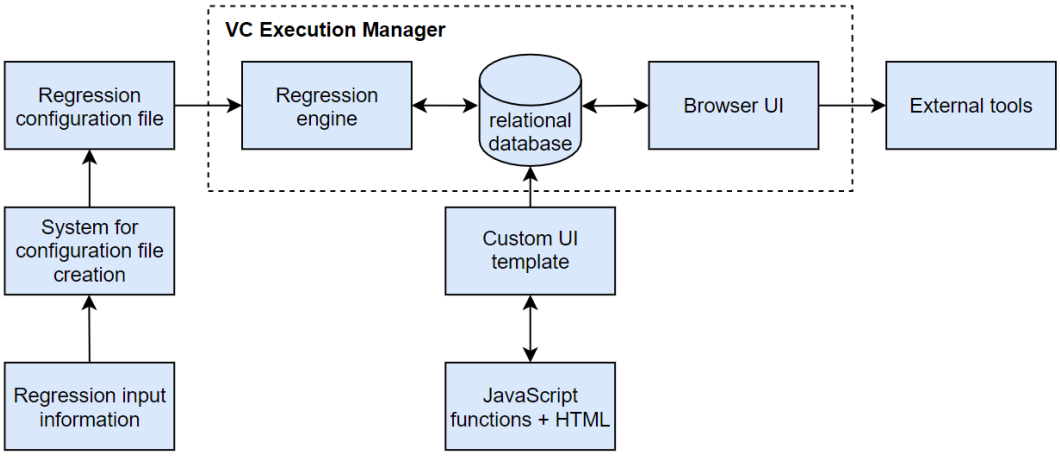


Figure 4.1: Solution's general architecture block diagram

4.1.1 Relational database, regression engine and browser UI

The first blocks to be analysed are at the centre of the solution and, without these elements, none of the other conceived components could possibly function together. This focal point in the architecture is composed of three elements, all of which are integrated in an already existing tool, namely VC Execution Manager (EMan) [5]. The VC Execution Manager tool, whose environment is illustrated in Figure 4.2, aims to provide support for managing compilation, regression test execution, data collection, reporting and tracking of design's verification processes.

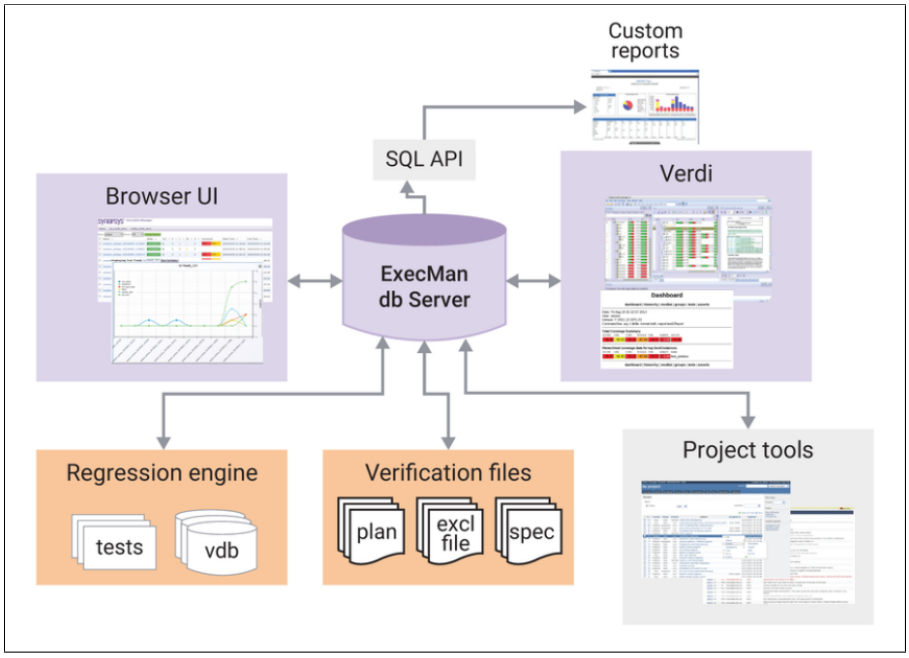


Figure 4.2: VC Execution Manager environment [5]

EMan is a widely used tool in the verification industry which automates coverage driven verification flows, tracks and collects regression results data in a relational database, and supports the annotation of coverage results with the aid of a custom planning management tool (Verdi). Additionally, EMan's comprising infrastructure also offers features such as [5]:

- regression efficiency - by optimising the execution of regressions in regards to time and costs;
- scalability - by enhancing the ability of teams to coordinate efforts and accurately estimate schedules, providing an Application Programming Interface (API) to store and track regression data in a relational database, supporting regression server and serverless capabilities as well as supporting regression job execution on local hosts or widely-used grid managers in a flexible manner;
- debugging productivity - by providing a plan-driven grading application to rank tests, regression monitoring, debug triage, regression trend charting and automatic rerun of regression failures;
- verification closure - by including flexible automated infrastructure management that improves time to coverage closure, allows the integration of in-house scripts and by supporting automatic coverage merging and automatic debug reruns.

EMan can also scale to support any number of projects and users, providing flexibility in terms of interface and report customisation. Thus, returning to the devised block diagram architecture of Figure 4.1, the three paramount elements contained within EMan's infrastructure are:

- regression engine;
- relational database;
- browser UI.

In regards to the EMan regression engine, it is responsible for running the regressions with given regression information and outputting the obtained results. It automatically merges coverage data from each test run, generating coverage reports as well as other output files. It is driven through a configuration file, which is delved into more detail in Subsection 4.1.2, storing all generated files and information through means of the relational database. The relational database is effectively the heart of the architecture, being responsible for storing regression results such as passed and failed test data, metrics or any custom value of an arbitrary user-defined type. Also, as mentioned, it supports the storing of coverage reports and

other output files obtained after regression running, being designed to connect to any regression engine in a straightforward manner. Lastly, gathered regression results may be viewed through the provided web-based UI, which is organised into a series of Web pages following the hierarchy present in Figure 4.3.

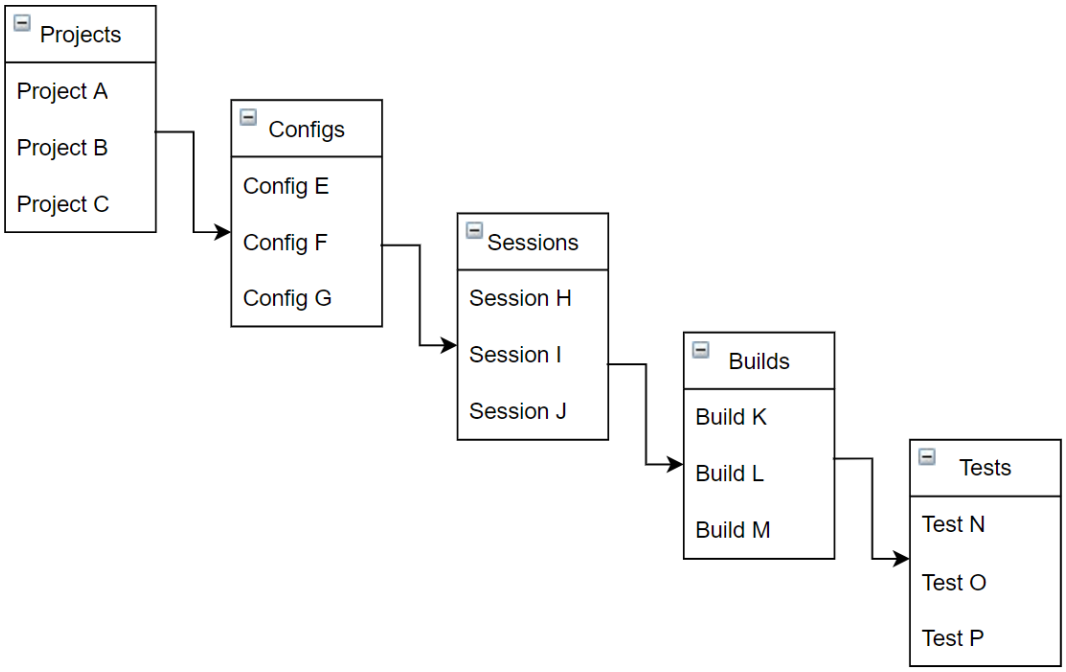


Figure 4.3: VC Execution Manager web page hierarchy

As Figure 4.3 shows, the EMan's UI is organised into five main web pages, which are ordered hierarchically and are responsible for displaying distinct regression data to users. This is done through tables of content present in each one of these pages, allowing users to navigate through the hierarchy in search of desired information. Following this hierarchy in a top down approach, it is seen that the project is at the top, followed by configurations. Projects may contain any number of configurations required, enabling the organising of tests. Configurations are automatically created during test execution by specifying the name of the configuration they should be associated to. Next, within each configuration, a certain number of sessions may be present, which are created every time the EMan tool is invoked and ordered by default from latest to oldest. The last two elements, namely builds and tests, are detailed in Subsection 4.1.2 and are what ultimately allow for regression running.

4.1.2 Regression input information and configuration file

Next in the architecture we find the regression input information and regression configuration file, separated by the system for configuration file creation. The latter

is a structure that allows for the creation of configuration files in an efficient manner before driving the regression engine when running regressions. This system is presented in further detail in Subsection 4.1.3. In the case of the regression input information block, as the name suggests, it must consist of one or more information sources containing the required regression information to be input to the regression engine in order to obtain regression results. The regression information contained in these sources is mainly of three types:

- software builds;
- test cases;
- options.

Software builds, which, for simplicity's sake, shall be referred to as builds from now on, consist of compiled RTL code and, therefore, are testable versions of the developed software. Test cases (or tests), as introduced in Chapter 2, generate the scenarios of interest in the testing process through means of stimuli, which can be applied to the previously mentioned builds in order to verify the presence or absence of errors in these. In regards to the previous two items, it is important to note that the EMan tool only allows for unique build names as well as unique test names to be run on a determined build. This is an important fact to take into consideration, as the script detailed in Subsection 5.2.2 must preserve the build and test case names to be used with the in-house scripts (even if repeated) but guarantee name uniqueness with EMan. Lastly, information regarding options aims to adapt the regressions so as to improve their effectiveness in achieving the metric based goals defined in the verification plan by utilising assisting tools for time and resource optimisation.

As mentioned in Subsection 4.1.1, EMan's works alongside a configuration file which is used to run regressions by driving the regression engine. This file uses the previously detailed regression input information by specifying the design builds, describing the tests in the regression and how these tests are run. Additionally, the configuration file uses error patterns so as to ascertain if builds have compiled properly and test cases have passed or failed. Error patterns use regular expressions [18], which enable pattern detection in log files created by in-house scripts for build compiling and simulation. The syntax for declaring a single error pattern within the configuration file is presented below:

```
pattern:::base_status:::custom_status:::precedence
```

Thus, four fields are responsible for making up an error pattern. As seen in the error pattern syntax shown above, each one of these fields is separated by the ":::" string, serving a specific purpose:

- pattern: consists of a developed regular expression for detecting patterns in the previously mentioned log files so as to determine the status of a build compilation or run test case;
- base_status: provides standard EMan statuses which consist of built-in fields to identify the outcome of a run test case. These statuses are mainly of three kinds, namely pass, fail or warning and are colour coded by EMan in its UI's table cells, turning them green, red or yellow, respectively, for ease of identification;
- custom_status: optional field that serves the same purpose as the base_status field but can override the presented standard statuses in the UI with a custom message instead. If used, this field is colour coded in the UI in the same manner as the according base_status inserted in the error pattern. On the other hand, if left blank, the base_status is used;
- precedence: numeric value which establishes a pattern's priority when determining the status of a build or test. For a pattern, the lower the precedence value, the higher the priority, with 0 being the highest priority possible.

Another note regarding the configuration file is that it should be saved with the *.emc filename extension and must be formatted according to a Yet Another Markup Language (YAML) syntax [19]. YAML is a serialisation language for all programming languages commonly used for configuration file creation due to its noteworthy readability and user friendliness. Roughly speaking, the YAML syntax consists of a series of key and value pairs that uses new lines and indentation of whitespace to denote structure. Furthermore, the data types supported by these files are of three kinds:

- mappings;
- sequences;
- scalars.

Mappings (which can be simple, sequence, nested or mixed) allow for the association of key-value pairs, where order is arbitrary but the uniqueness of each key must be assured. Secondly, sequences consist of lists which include values listed in a specific order and contain any number of items needed. Sequences start with a dash followed by a space in the file, while indentation enables separation from the parent. Finally, scalars represent arbitrary data that can be used as values such as strings, integers, dates, numbers or booleans. Examples regarding these configuration files may be found in Chapter 5, where the previous data type values specific to the EMan environment are described.

4.1.3 System for configuration file creation

Having clarified the significance behind the regression input information and regression configuration file, the system for configuration file creation can now be discussed. Although configuration files may be manually devised by developers, the repetitive nature of regressions would turn this process time consuming, dull and, in certain cases, bothersome. Consequently, the concept for a system for configuration file creation was devised and integrated within the solution's architecture. The workings of this system are illustrated in more detail in Figure 4.4.

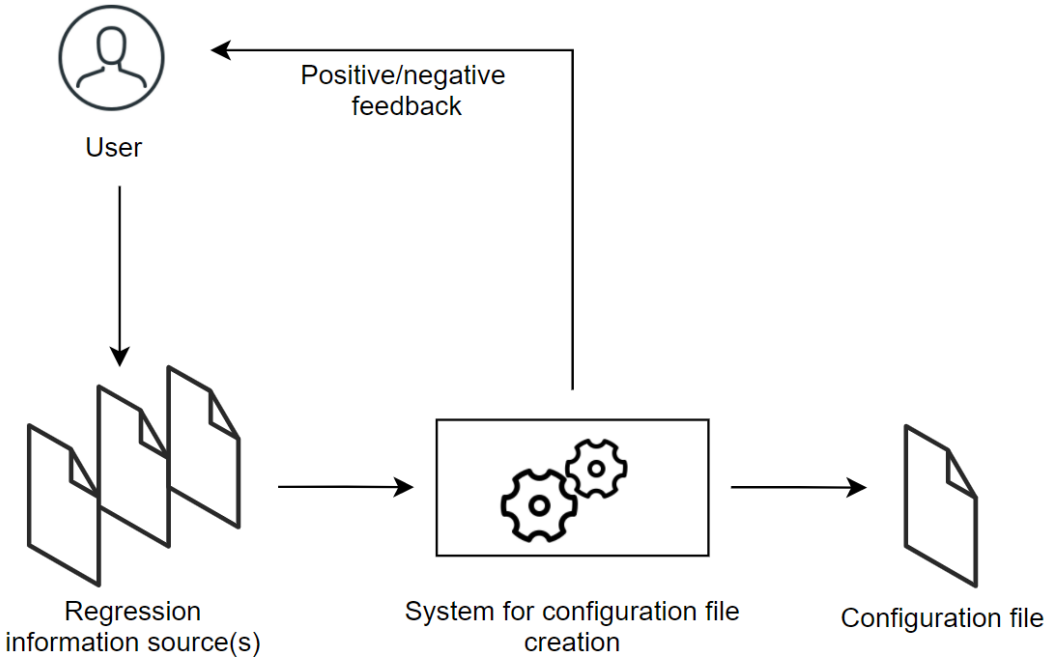


Figure 4.4: Configuration file creation system overview

Therefore, the purpose of this system is to create configuration files in a simple and effective manner through the input regression information provided by the user, carrying out the first requirement for the solution brought to light in Section 3.2 of Chapter 3. The system must then process and organise this information in such a way that an output consisting of a configuration file containing the corresponding input information is created in an efficient manner with little to no difficulty. Additionally, this system must be full proof, ensuring that any errors associated with the input information or input information source's syntax are fed back to the user through this system. On the other hand, absence of errors should also be communicated, assuring the user that the configuration file was successfully created and that a regression may be run with this file.

4.1.4 Custom UI template and JavaScript functions with HTML

Having gone through the architecture's section regarding regression running, it is now possible to consider the blocks responsible for achieving the latter six requirements defined in Section 3.2. These blocks are the custom UI template block as well as the Hypertext Markup Language (HTML) injected code block, which work in unison in order to achieve the following goals:

- customise the browser UI;
- save custom values into the database.

Firstly, regarding the browser UI customisation, EMan supports the addition of custom columns or buttons to default tables containing regression data, enabling the addition of information of arbitrary user-defined nature to be visualised. The addition of new columns and buttons to the tables is accomplished through custom UI template files, whose addition or removal to the EMan server is realised through a shell's command line. Control over the rendering of buttons or cells belonging to these columns added by the custom UI templates is supported by JavaScript scripts, which are outsourced by the template and contain functions with HTML code to be used in the UI. These templates can be applied to projects created in the EMan database server and handling of these templates in regards to their use, addition or removal from the database is exclusive to individuals with granted admin privileges. These templates, similarly to the configuration files, must be formatted according to a YAML syntax, however, their filename extension must be of the *.emd type (as opposed to the previous *.emc extension). On the other hand, custom values may be saved into EMan's database using provided predefined JavaScript APIs which aid in transferring the custom information between the application and the database server. This system is subsequently illustrated in Figure 5.7 of Chapter 5 in order to gain a better understating of its working principle.

Therefore, the customisation of the browser interface as well as the capability of saving custom values into the database for posterior analysis enables a plethora of possibilities regarding the information that can not only be presented to users but also inserted and saved by them. This enables the formation of new columns for error signatures, their assignment to developers and bug tracker links. Additionally, elements made available through the HTML language allow for on the fly changes in the previously mentioned assignments and, furthermore, the use of JavaScript can allow for email functionalities, which may be sent through the custom built interface to the correspondingly assigned developers. Finally, with the possibility of altering the browser's UI, attempting to reproduce a similar interface as the previously used solution can aid the team in an effortless transition between solutions, hereby completing all the established requirements.

4.1.5 External tools

One final element belonging to the solution's architecture is the external tools block. As seen in the solution's architecture illustrated in Figure 4.1, this block does not possess a direct link to the database, having solely one connection to the browser UI. This situation aims to convey the idea of users manually accessing these external tools from the browser UI. Consequently, bug trackers or external project management tools foreign to EMan's environment and framework, should be accessible to users through the custom interface devised. This, as mentioned for the case of bug tracking tools, may be accomplished through the use of hyperlinks, which should be inserted by users and possess the ability to redirect peers to the said tool and corresponding issue/project.

The architecture of the proposed solution was the main focus of this Chapter. As was seen, the heart of this architecture lied in three main components which were all part of the VC Execution Manager tool. Other components regarding the regression running process were also discussed, namely the input information, system for configuration file creation and the configuration file itself. Additionally, elements used for interface customisation were also talked about, such as the JavaScript functions file and the custom UI template. A final element regarding external tools to be used alongside EMan was also briefly explained.

Chapter 5

Implementation

This chapter focuses on the implementation of the project. Firstly, the achieved environment set up is addressed regarding the taken steps in order to correctly configure the EMan tool for use. Furthermore, the system devised in order to achieve efficient configuration file creation is explained with respect to all its integrating elements and processes applied so as to achieve this goal. Finally, the procedure regarding the customisation of EMan's User Interface and the saving of custom values in the database is analysed.

5.1 Environment setup

The first step in order to commence the implementation phase of the project consisted in the setting up of the environment in which the work was going to be carried out to fulfil the established goals. So as to achieve this objective, completing a series of compulsory tasks in the machine's workspace was first necessary. These tasks included:

1. installing the VC Execution Manager tool;
2. setting required paths and variables;
3. setting the correct grid;
4. creating/selecting a project;

5. setting up the web server and obtaining its Uniform Resource Locator (URL).

In regards to the installation of the EMan tool, and as a consequence of the practises adopted at Synopsys which prevent the self installation of all software on machines on the employees' side, this first task was carried out by the Information and Technology (IT) department at the company. Therefore, upon consulting with the IT team and expressing the need for the mentioned tool, permission to install EMan on the desired machine was granted and later carried out. This installation process was done through means of a script solely for this purpose and whose invocation is exclusive to the IT team, performing the necessary tool configurations and setup actions.

Second in the outlined task list, was the setting of required paths and variables for EMan to work correctly on the desired machine. In order to achieve this, a second script was required, namely sourceme.csh. This script arises as a byproduct of the installation process, being created upon invocation of the installation script. However, in the case of the sourceme.csh script, invocation on the client's side is required in order to set the required paths and environment variables so as to setup EMan in the current shell. A noteworthy environment variable which is set upon invoking the sourceme.csh script is EMAN_HOME, whose value, as the name suggests, consists of a path in which all files and information associated with the EMan tool are stored by default. The sourcing of this script is done through the command line in a similar fashion to the following:

```
$ source $EMAN_HOME/path/to/file/sourceme.csh
```

Upon configuring the environment so as to enable the correct working of the EMan tool, the next steps concerned the setting up of the regression environment so as to enable regression running and the analysis of regression results. Therefore, in order to properly run regressions, setting a grid was indispensable. A grid, also referred to as a compute farm, consists of a series of machines that allow for simultaneous test running, thus allowing for task parallelization by means of efficient allocation of computing resources. This task involved the use of a setup file, designated synopsys_eman.setup, which was manually edited in order to select the required grid. EMan provides support for several distinct compute farms, however, in the case of the verification team, the compute farm used was Sun Grid Engine (SGE). The synopsys_eman.setup possesses a YAML syntax, which echoes the supporting of all these grids by including a distinct mapping destined for grid options. As seen in the file snipet below, the main key (called grid) contains a sequence of further mappings belonging to each individual grid. The selection of a specific grid encompasses the adjustment of a boolean scalar value, according to the

wanted grid. Furthermore, the last key value pair of each element of the sequence allows for a path to a setup file for that specific grid:

```
...
grid:

  # SGE settings
  - SGE: True
    setup_file: /path/to/file/sge_file

  # LSF settings
  - LSF: False
    setup_file: /path/to/file/lsf_file

  # RTDA settings
  - RTDA: False
    setup_file: /path/to/file/rtda_file
...
```

The fourth task in the overall environment setup demanded the creation or selection of a project in which the regressions would be run. In order to achieve this, setting a particular environment variable in the current shell was necessary, namely EMAN_PROJECT. For this purpose, the following instruction was run on the command line, enabling the selection or creation of a project depending on its previous presence/absence in the database, respectively:

```
$ setenv EMAN_PROJECT <project_name>
```

The final task was to set up the web server in which regression results would be analysed through EMan's UI, as well as obtaining a URL so as to access the server. Although already setup by different branches of the company, a personal web server for the verification team was required in order to obtain admin privileges and thus enable the customisation of the UI when required. As with the EMan tool installation, the EMan server setup was also carried out by the IT department, enabling the access to the web server through the company's login credentials upon locating it through its URL. To achieve the latter, the following custom command specific to EMan was used, outputting the server's URL:

```
$ eman -get_web_url

http://hostname:port/eman/
```

5.2 Configuration file creation

Once setup the workspace environment, allowing the EMan tool to function as intended, regression running was consequently enabled. However, as explained in Subsection 4.1.2, in order to run regressions with this tool, *.emc configuration files must firstly be constructed and contain the desired regression information to be processed by the regression engine. Furthermore, as mentioned in Subsection 4.1.3, the repetitive nature of configuration file creation when running different regressions can turn this process arduous and time consuming. Consequently, before running regressions and viewing/analysing their results with the EMan tool, the system for configuration file creation was implemented to turn this process quicker and more effective. This system is illustrated in Figure 5.1 and is based on the working principle of the system overview of Figure 4.4.

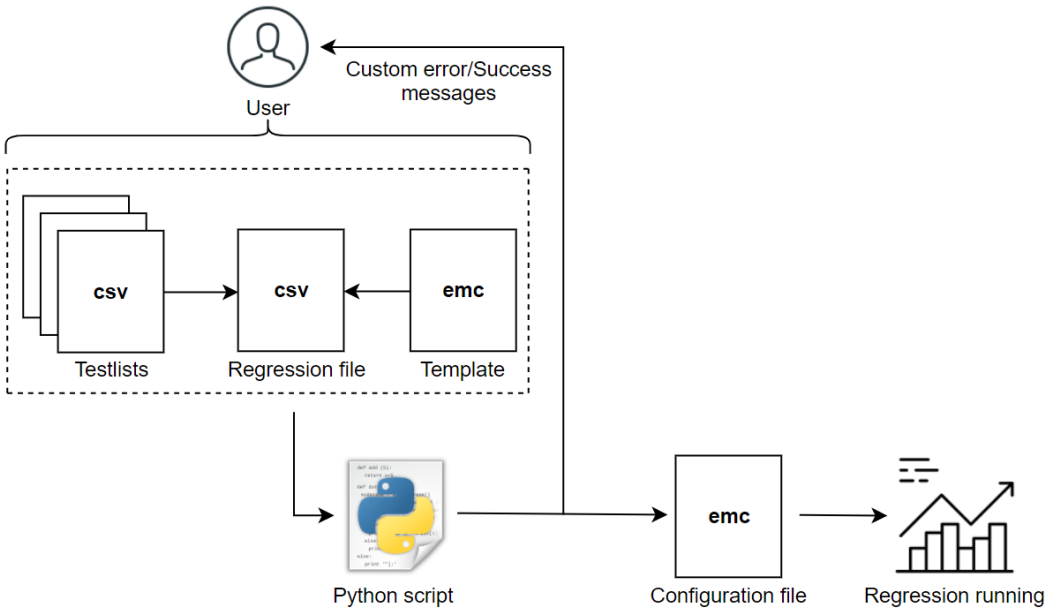


Figure 5.1: Configuration file creation system

As seen in Figure 5.1, the configuration file creation system was implemented through means of a Python script. This method was chosen as a result of the programming language's popularity, readability, versatility and wide range of applications. Furthermore, this method of work in automating tasks through scripting in this language is commonly adopted by the verification team. Therefore, familiarisation regarding the devised algorithms and language attributes as well as possible changes required upon project completion may be carried out in a straightforward manner. The focal point of this system lies in this script, which acts as a black box, handling a series of input information and enabling the creation of the desired configuration file needed for regression running. More specifically, the script accepts

one direct input, which, in itself, outsources further inputs, outputting the configuration file on the other side. The inputs, Python script and output of this system are further detailed in Subsections 5.2.1, 5.2.2 and 5.2.3, respectively.

5.2.1 Inputs

As was previously depicted, a total of three distinct inputs are required in order to achieve the required configuration file creation goal. All three inputs are user editable and require changes in order to form an adequate configuration file for running the desired regression. Thus, in this subsection, the regression file, testlists and *.emc template are further analysed.

5.2.1.1 Regression file

The regression file constitutes the main input provided without which the Python script would have no information to work with in order to create the desired output configuration file. It consists of a Comma Separated Value (CSV) file similar to the example illustrated in Figure 5.2.

	A	B	C	D	E	F	G
1	<BUILD_NAME>	TESTLISTS	<BUILD_OPTIONS>	EMC_TEMPLATE	<BUILD_SCRIPT>	<SIM_SCRIPT>	<SIM_OPTION>
2	build1	testlist_X	-option=an_option	template.emc	build_script.py	sim_script.py	-option=optionA
3		testlist_Y					
4		testlist_Z					
5	build2	testlist_A					
6		testlist_B					-option=optionB

Figure 5.2: Regression file example

The first item of information to note is the file's header, present in the first row of the regression file. This header, as will be seen throughout this section, will be indispensable in the process of creating configuration files in an effortless way, possessing a threefold purpose:

1. turn information in the file intelligible by users;
2. allow information processing and mapping by the Python script;
3. enable the overriding of test options;

Therefore, by viewing the header, users who may want to use a certain regression file to run future regressions should have a clear cut view of the information present in this file, being able to identify each column's contents in a straightforward manner. Secondly, an important matter taken into consideration at the time of development consisted in the approach chosen for the Python script to correctly identify the information in the regression file, seen as this could be done in one of two ways, through:

- column order;
- column names.

One of the previous two elements in the regression file must then be fixed in order to establish a coherent method of identifying, validating, processing and extracting information. Upon consideration, conceiving a regression file with fixed header titles was prioritised over a fixed column order. This allowed for advantages such as a mutable column order, enabling flexible information displaying according to each user's preferences and, above all, standardised header titles, preventing possible confusions with the type of information present in each column. Lastly, the third purpose of the header is to override test case options through matching column titles, which is further detailed in Subsubsection 5.2.1.2.

Now that the the main purposes for the regression file header have been explained, it is now feasible to begin understanding the underlying purpose for the information belonging to each of the columns. For this matter Table 5.1 was conceived.

Table 5.1: Regression file information and rules

Column title	**Nature**	**Information/Purpose**	**Row(s)**
<BUILD_NAME>	Mandatory	Names of builds prepared and compiled during the regression.	-
TESTLISTS	Mandatory	Files containing lists of test cases run on their corresponding build.	All
<BUILD_OPTIONS>	Mandatory	Options parsed to the in-house build script.	New build
EMC_TEMPLATE	Mandatory	File containing template *.emc blocks for use in the output file.	Second
<BUILD_SCRIPT>	Mandatory	In-house script used for preparing and compiling builds.	Second
<SIM_SCRIPT>	Mandatory	In-house script used for generating and executing simulation commands.	Second
Other options	Optional	Override testlist's test case options.	Testlist specific

Alongside with the information provided, each column possesses a mandatory or optional nature depending on whether or not it is imperative for it to be present in the regression file, respectively. Mandatory columns must then be present in the regression file for the Python script to function correctly. The single use regarding optional columns is to override test case options present in a given testlist which,

as alluded to in Table 5.1, consists of a file with a list of test cases to be run on a determined build.

Furthermore, rules regarding column information were also implemented in order to standardise a syntax within regression files. These rules aim to indicate where information from each column can or cannot be inserted with respects to the regression file's rows, taking into account that the first row is reserved for the header. Thus, following the column order presented in Table 5.1, the rules state that build names can be present in any row of the file, since the majority of the other information in the file is dependent on which rows these names are declared. However, at least one build name must be declared in the second row of the file, since the absence of builds in the regression file would mean nothing requires testing. Testlists, on the other hand, must be present in all rows of the regression file, consisting of outsourced files which contain the relevant test cases to be applied to specific builds. The regression file's rules are configured in such a way that the declared testlists' test cases are applied to the last declared build. For example, returning to Figure 5.2, the user configured the regression file in such a way so that test cases from testlist_X, testlist_Y and testlist_Z are applied to build1, while test cases from testlist_A and testlist_B are applied to build2 when running a regression. Therefore, having a row in which a testlist and, consequently, test cases are not declared does not make sense since the main purpose of regressions is to run test cases. Also, a specific build name need only be declared once in the file seen as information like testlists are associated with them and may be appended to these builds in the following rows. This helps organise the file, seen as information relating to the same build is declared in one specific section of the regression file and, additionally, improves the readability of the information, as a user is quickly able to understand to which build information relates to. Next, build options are presented, consisting of options to be parsed to the in-house build script, subsequently addressed. These options must only be declared on new build rows, rows in which a build name has been declared. After this, three additional outsourced files are introduced, namely the *.emc template file, build script and simulation script. The purpose of these files is to provide *.emc template blocks for the output configuration file, preparing and compiling builds and generating and executing simulation commands, respectively. Since these are outsourced files, it is logical that these only need to be declared once, and so must be done in the second row of the regression file. Finally, other columns of an optional and user defined nature can be introduced into the regression file. As previously mentioned, the single use regarding optional columns is to override test case options. Therefore, information inserted in these columns must be testlist specific, meaning that it should be declared in the same row as the testlist containing the test cases whose options the user wishes to override.

5.2.1.2 Testlists

The next input to analyse are the testlists. As previously mentioned, the Python script was developed so that only one input, the regression file, is directly required to be parsed as an argument. Parsing testlist files directly to the Python script is not needed since these are mentioned in the regression file and, as such, are outsourced by it enabling access to their information. Similarly to the regression file, testlists are also CSV files as exemplified in Figure 5.3.

	A	B	C
1	<TEST_NAME>	<N_RUNS>	<SIM_OPTION>
2	test1	8	-option=option1
3	test2	12	
4	test3	2	
5	test4	1	-option=option2

Figure 5.3: Testlist file example

In this case, testlists also make use of a header which follows the same principles and threefold purpose previously described for the regression file. However, as expected, information contained in these files serves a different purpose, as is delineated in Table 5.2.

Table 5.2: Testlist file information and rules

Column title	**Nature**	**Information/Purpose**	**Row(s)**
<TEST_NAME>	Mandatory	Name of a test case	All
<N_RUNS>	Mandatory	Number of test case runs	All
Other options	Optional	Additional test options	Test specific

Column nature follows the same conventions as with the regression file and, thus, mandatory columns must be present in all testlist files. Optional columns in these files carry the sole purpose of allowing for additional test case options to be parsed to the simulation script.

With respect to the rules defining the syntax in the testlist files, and following the column order presented in Table 5.2, test case names as well as the corresponding number of times each test case is run must be present in every row of a testlist file. Contrarily, other optional columns destined to provide additional options to test cases are test specific, meaning that they must only be declared in rows where the specific test cases to which the options want to be applied have been declared.

One last point regarding the testlist files is their susceptibility to having certain declared options overridden by the regression file. This feature, requested by the verification team, works by inserting matching column titles in both the regression

file and testlist files. And, once optional columns in the regression file are testlist specific, their information will override the same column's information in that testlist for all test cases. For example, by viewing the regression file of Figure 5.2 and the testlist of Figure 5.3 it is possible to observe a common column between them, <SIM_OPTION>. Now, considering the testlist in question is testlist_X mentioned in the regression file, then all of the test cases' information in the <SIM_OPTION> column of that testlist will be overridden with the information -option=optionA, including test cases in which this column was blank.

5.2.1.3 Template file (*.emc)

The final input to the Python script is the *.emc template file which, as the name suggests, acts as a backbone to the final configuration file. It contains a foundation for all the YAML blocks which the Python script shall build upon to create the final configuration file. The used *.emc template file is present in Figure 5.4.

```
config_name: <CONFIG_NAME>

variables:

  - current_dir: $PWD

base_build:

  - name: base_build
    error_patterns:
        - ^.*BUILD FAILED:::fail:::build failed:::0
        - ^.*BUILD SUCCEEDED:::pass:::build successful:::0
    abstract: yes

base_test:

  - name: base_test
    error_patterns:
        - ^UVM_ERROR.*@\s\d+\.\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
        - ^UVM_FATAL.*@\s\d+\.\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
        - ^UVM_INFO.*@\s\d+\.\d+ns:\s(?P<ERROR>.*):::pass:::(ERROR):::0
    abstract: yes

builds:

  - name: <BUILD_NAME>
    run_cmd: <BUILD_SCRIPT> <ORIGINAL_BUILD_NAME> <BUILD_OPTIONS>
    run_dir: $current_dir
    extends: base_build

tests:

  - name: <TEST_NAME>
    build: <BUILD_NAME>
    run_cmd: <SIM_SCRIPT> <ORIGINAL_TEST_NAME> <SIM_OPTION>
    run_dir: $current_dir
    count: <N_RUNS>
    extends: abstract/base_test
```

Figure 5.4: *.emc template file

Therefore, the *.emc template file contains six distinct mappings to be used by the Python file in the creation of the sought after configuration file. Each one of them serves a different purpose, layed out in Table 5.3.

Table 5.3: *.emc file mappings and value information

Mapping key	**Value information**
config_name	Configuration name presented in EMan's UI
variables	Environment variables used during the regression
base_build	Properties to be inherited by all builds
builds	Builds to be tested during the regression
base_test	Properties to be inherited by all tests
tests	Tests to be run during the regression

Although the first two mappings of Table 5.3 consist of simple mappings (since they only contain one key value pair) the last four presented contain sequences of further mappings. Therefore, it is also important to analyse these mappings and the keys they contain in order to gain a better perspective of their meaning and purpose. For this reason, Table 5.4 was devised.

Table 5.4: *.emc file keys and value information

Key	**Value information**
name	Name of a build or test case
run_cmd	Command line to run in-house compilation/simulation scripts
run_dir	Directory from where the run_cmd command is executed
build	Name of a defined build on which a test case shall be run
count	Number of times a test case is run
error_patterns	Error patterns to determine the status of a build/test
abstract	Defines an inheritable build/test
extends	Allows a build or test to inherit another build/test

Thus, all *.emc file keys to be used are addressed in the previous two presented tables. However, a closer look at the *.emc template file of Figure 5.4 reveals certain key's values in the form of column titles present in the regression file and testlist files. Here lies the devised method allowing for information mapping between the CSV regression and testlist files to the final configuration file. This mapping process, enabled by the Python script, bases its working principle on the matching of CSV file column titles (solely titles contained between the "< >" characters) and strings which represent values in the *.emc template file. Solely three strings are exception to this rule as their origin is of a different nature, namely <CONFIG_NAME>, <ORIGINAL_BUILD_NAME> and <ORIGINAL_TEST_NAME>. The <CONFIG_NAME> string obtains its value from

the regression filename invoked with the Python script. On the other hand, the latter two strings aim to surpass EMan's limitations regarding duplicate build and test case names. The addressed mapping process is further studied in Subsections 5.2.2 and 5.2.3.

5.2.2 Python script

Following the configuration file creation system previously illustrated in Figure 5.1, and having analysed the three distinct inputs provided, it is necessary to understand the processing of information carried out by the developed Python script. Hence, Figure 5.5 explains the taken steps in this process through means of a flowchart.

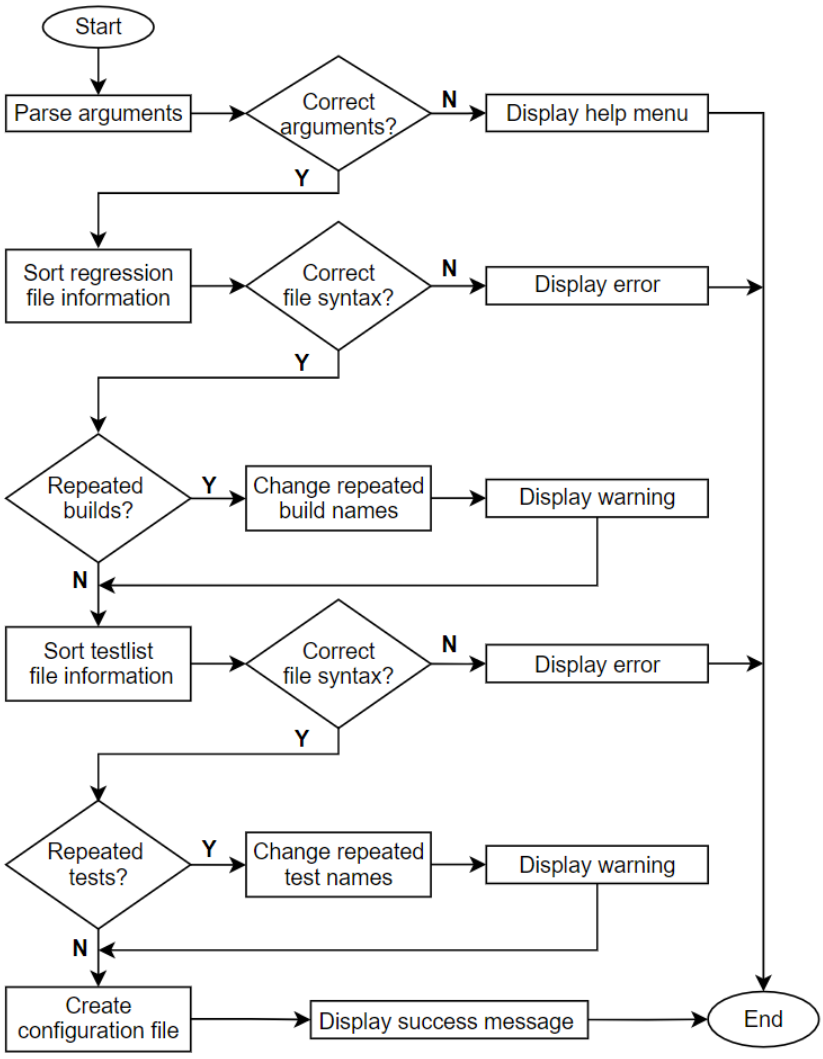


Figure 5.5: Python script flowchart

In order for the Python script to carry out the tasks presented by the flowchart, numerous functions were devised in order to achieve the ultimate intended goal. However, from these, nine major functions stand out. Their names, purpose and calling function are portrayed in Table 5.5.

Table 5.5: Python script functions

Function	**Purpose**	**Calling function**
parse_args	Parse arguments	main
sort_regr_info	Sort regression information	main
check_regr_errors	Check regression file's syntax	sort_regr_info
check_repeated_builds	Check for repeated build names	sort_regr_info
sort_testlist_info	Sort testlist information	main
check_testlist_errors	Check testlist files' syntax	sort_testlist_info
check_repeated_tests	Check for repeated test names	sort_testlist_info
create_emc_file	Create configuration file	main
replace_strings	Replace certain template strings	create_emc_file

Table 5.5 is arranged according to the order of functions called in the Python script and aims to give a general idea regarding their principle of operation. However, a more in depth explanation of the inner workings of these function is detailed below:

- parse_args: Checks if the number of arguments parsed through the command line argument vector (argv) is equal to two, corresponding to the Python script name and the CSV regression file. If not, or if a "-help" switch is parsed through the command line, displays an auxiliary menu to the user in order to explain the Python script's usage;

- sort_regr_info: Opens the regression file and extract its columns' contents into a list of dictionaries, called the build list, where each dictionary in the list contains information belonging to a single build. In these dictionaries, each key corresponds to a column title and the values correspond to the information present in each of the CSV cells. For testlists, a new list is created within each of these dictionaries in order to store the testlists to be applied to each build. Other information in optional columns of the regression file is stored in the same way as testlists so as to keep the information synchronised and guarantee that overriding options are passed to the correct testlists. One last piece of information separately passed to each build dictionary is the <ORIGINAL_BUILD_NAME> key, whose value aims to preserve the original build name in the regression file for use with the in-house build script;

- check_regr_errors: Function that is called by the sort_regr_info function in order to find possible errors associated with each row of the regression file. As a result, as the sort_regr_info function iterates through each row of the

regression file aiming to sort all the information into the build list, it calls the check_regr_errors function in order to assess if each one of these row's syntax is in accordance with the rules asserted in Table 5.1. If any of the row's syntax is incorrect, a custom error message detailing the location of the error is presented to the user;

- check_repeated_builds: Function that is called in the final stages of the sort_regr_info function in order to check if repeated build names exist. If so, repeated build names are edited by adding a version number to these and a warning is emitted by the Python script warning the user of the specific build names which were edited. This insures all build names are unique to Execution Manager according to its configuration file requirements, previously addressed in Subsection 4.1.2 of Chapter 4.

- sort_testlist_info: Progressively opens each testlist file from the build list and extracts their contents into another list of dictionaries, called the test case list, where each dictionary contains information regarding one test case. Keys and values for these dictionaries are selected and stored in a similar fashion as with the regression information. Two separate pieces of information are also included in the test case list including a <BUILD_NAME> key, whose value corresponds to the build to which the test case belongs and will be run on, as well as an <ORIGINAL_TEST_NAME> key, which aims keep the original test case names intact if subsequently changed by the check_repeated_tests function.

- check_testlist_errors: Function called by the sort_testlist_info function to check for possible errors in testlist file rows. The sort_testlist_info sorts all testlist information in the test case list by scanning each testlist row progressively. For each of the test case's row information that is stored, the sort_testlist_info is called once in order to insure that the row information complies with the syntax rules exhibited in Table 5.2. If any testlist rows do not follow the syntax established, a custom error message detailing the location of the error is shown to the user.

- check_repeated_tests: Function called by the sort_testlist_info function in order to verify that repeated build names belonging to the same build do not exist. If they do exist, these are edited in the same fashion as performed by the check_repeated_builds function, adding a version number to these. Warnings for each edited test name are issued informing the user about the event. As with repeated builds, this insures that test cases belonging to the same build are unique as per EMan's requirements.

- create_emc_file: Opens the outsourced *.emc template file specified in the EMC_TEMPLATE column of the regression file and extract all mappings into distinct string variables. Following this, this function creates a new file to be the desired output configuration file with the same filename as the input regression file but with the required .emc extension (for example, a "regression_abc.csv" regression file outputs a "regression_abc.emc" configuration file). Finally, according to the number of builds in the build list and test cases in the test case list, the according number of builds and tests mappings are inserted in the configuration file.

- replace_strings: Function called by the create_emc_file function aiming to replace strings present in the *.emc template with CSV file information. Firstly, this function receives a certain mapping string variable as a parameter as well as a boolean variable identifying the mapping as a build mapping or test mapping. Then, according to the mapping type, the string replacement process occurs, replacing matching column title strings (exemplified in Figure 5.6 with the regression file and the builds mapping) with information from either the build list or test case list.

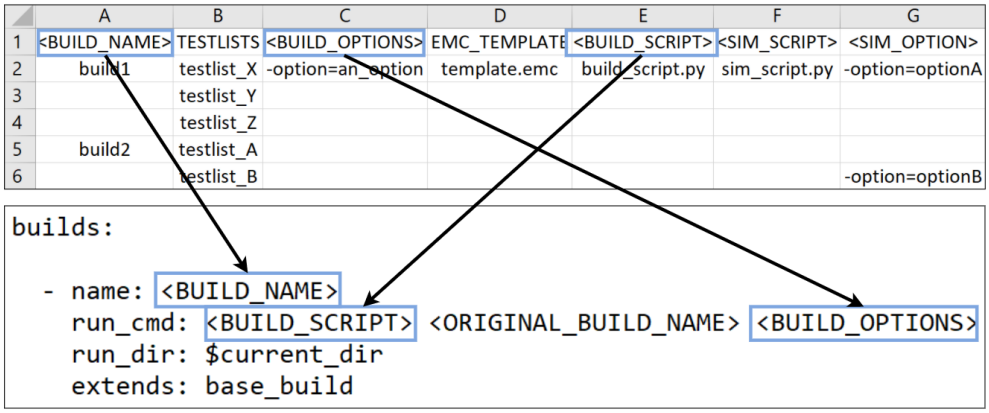


Figure 5.6: Column title and template string correspondence

This takes place in an orderly manner for all mappings present in the newly created configuration file and, once string replacement is finalised, a success message is transmitted to the user informing him of the successful creation of the configuration file.

5.2.3 Output configuration file

Finally, once the developed Python script terminates its intended tasks, an *.emc configuration file is expected to be output by the Python script into the same workspace directory where this script is located. In regards to its contents, the output configuration file is also expected to contain all mapping types presented in the *.emc template file, as well as the correct number of builds and tests mappings

corresponding to the number of builds and test cases passed through the regression file and testlist files (this means that one build or one test case should introduce new builds or tests mappings into the configuration file, respectively). Additionally, the configuration file is also expected to contain the desired information from the CSV regression file and testlist files. This information, as previously mentioned should be introduced into the file via the devised mapping process.

Upon completing these tasks, the configuration file should be ready for use in a regression environment. In the case of the EMan tool, additional options may be enabled such as the merging of coverage data obtained during a regression as well as the back-annotation of these results to a verification plan. EMan allows for regression running through the command line. For the particular scenario regarding the previous options, a regression may be run with the following command:

```
$ eman -config <configuration_file> -merge_dir <merge_directory>
    -plan <verification_plan>
```

5.3 VC Execution Manager UI customisation

Having implemented the configuration file creation system through the developed Python script, a more efficient method for running regressions was enabled. With this being said, and once regression results could be viewed via the EMan tool's UI, the next step was to customise this UI so as to integrate in it all the functionalities possessed by the previous used solution. In order to achieve this goal, the UI customisation system present in Figure 5.7 was followed.

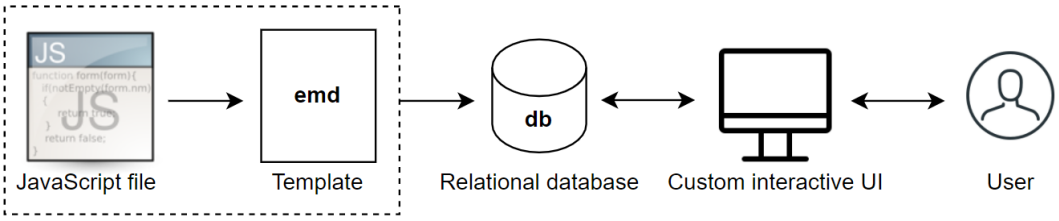


Figure 5.7: UI customisation system

This system is built into EMan's infrastructure and may be used when custom values are required. It bases its working principle on an *.emd template file (not to be confused with the *.emc template file) which calls functions from the JavaScript file. Both these files comprise the inputs to the customisation system, being submitted to the relational database which, in itself, applies these to the browser's UI. This enables interactive features to be inserted in the UI through the JavaScript coded functions present in the JavaScript file. On the other hand, the customised

interactive UI presents users with the newly introduced features devised, allowing them to view, select and input required custom values. Finally, these custom values are inserted into the database so that the user's peers may also view the inserted information and keep in sync with the project developments.

5.3.1 Template file (*.emd)

As previously stated, the UI customisation system bases its working principle on an *.emd template file (which, as expected, must contain an .emd file extension). And, just as the *.emc template file served as a foundation for regression configuration files, the template *.emd file serves as a foundation for an interactive custom UI. The used *.emd template file is illustrated in Figure 5.8.

```
- name: triage_template

- scripts:

  - JS_functions.js

- columns:

  - signature:

    - title: Error Signature
    - tables:
      - tests
    - function: get_signature
    - inputs:
      - status

  - jira:

    - title: JIRA Link
    - tables:
      - tests
    - function: get_jira_link

  - assignee:

    - title: Assigned to
    - tables:
      - tests
    - function: get_assignee
```

Figure 5.8: *.emd template file

By observing Figure 5.8, it can be seen that this type of file's information must be formatted according to a YAML syntax as was also seen with previous files. However, the structure of this file is predefined to be a sequence where each element contains a single primary mapping for a specific purpose. These mapping's keys and value's information are presented in Table 5.6.

Table 5.6: *.emd file mappings and value information

Mapping's key	**Value infomation**
name	Name of template submitted to the relational database
scripts	Outsourced JavaScript scripts for function calling
columns	Custom table columns to be applied to the EMan's UI

Therefore, the mapping corresponding to the name key contains as its value the name of the *.emd template file that will be submitted to the database for identification purposes. The scripts mapping contains a sequence of outsourced JavaScript scripts for subsequent function calling in order to add interactivity to the UI. Lastly, the columns mapping contains a lower level sequence in which each element is a mapping representing a custom column to be applied to EMan's web page tables. The file present in Figure 5.8 contains three of these last mentioned mappings, namely signature, jira and assignee. Each of these aim to insert a custom column in the UI to display error signatures, bug tracker links (in which the bug tracker software used is called Jira) and assigned developers, respectively. Additionally, by delving deeper into the file's contents, further relevant information may be seen within the signature, jira and assignee mappings in the form of yet more sequences. The lowermost mentioned sequence contains common keys shared by each of the columns and which are detailed in Table 5.7.

Table 5.7: *.emd columns mapping's keys and value information

Key	**Value's information**
title	Column's title in the UI
tables	Intended EMan web page tables in which to insert the corresponding custom column
function	Called JavaScript function for column interactivity
inputs	Other columns whose information is useful in obtaining the custom values to be present in the custom column

Thus, as mentioned, these key value pairs are responsible for the custom columns' characteristics. Values associated with the title key represent the custom column titles to be shown in EMan's web page tables. Secondly, the underlying information paired with the tables key determines in which of EMan's web page tables the custom column is inserted. With regards to function values, these aim to call developed JavaScript functions present in the files referenced by the scripts mapping, thus determining the specific interactivity features for that particular custom

column. Finally, inputs aim to specify other columns whose information is vital to the custom column, serving as an argument to the custom column's associated JavaScript function. For example, providing an error signature through the signature custom column may only be done if the test error (present in the default status column of EMan's UI) is provided. Hence, the status column is provided as an input to the signature column.

5.3.2 JavaScript file

This input to the UI customisation system represents the file outsourced by the *.emd template file by means of its scripts mapping illustrated in Figure 5.8. As expected, its contents consist of a series of developed JavaScript functions, some of which are directly called by the *.emd template file through its function key. Functions called directly by the *.emd template, are provided with an argument by EMan in the form of a JavaScript Object Notation (JSON) object. This object possesses information regarding the table row in which the functions operate at any given time and enables them to access other's column information if required. Each of these functions, presented in Table 5.8, are developed with specific custom columns in mind and can grant these with features for user interactivity.

Table 5.8: Developed JavaScript functions

Function	Purpose	Called by
get_signature	Obtain error signature based on a test's status message	*.emd template
get_jira_link	Manage the bug tracker custom column's interactive features	*.emd template
generate_jira_link	Obtain a URL to a specific jira issue and generate the according hyperlink	get_jira_link
get_assignee	Manage the assign to custom column's interactive features	*.emd template
open_email_window	Open EMan's email window and insert the corresponding email address	assign_func
check_email_info	Check email data validity	open_email_window

Table 5.8 aims to give a general explanation regarding each function's purpose as well as the location from which each of these is called in order to perform their intended duty. Nevertheless, a more thorough description of each of the function's inner workings is detailed below:

- get_signature: obtains a specific test's error message via the built-in input status column and replaces all its digits with an "X" character. This enables

the creation of general error signatures, in which errors with similar root causes match, enabling their subsequent binning into groups;

- get_jira_link: uses HTML code in order to create a text box for submitting bug tracker IDs. Additionally, through the provided JSON object argument, it checks if it's the first time an ID is being submitted in association with the corresponding error signature of that particular row. If so, once again through HTML code, it displays a button with the value "Link" and, if not, it displays a button with the value "Change". These buttons, when clicked on by the user, invoke another function responsible for creating the hyperlink to the bug tracker;

- generate_jira_link: receives the inputted bug tracker ID through the action event of pressing the buttons created in the get_jira_link function. And, based on this information forms an URL address to the associated bug tracker issue. Finally, possessing this specific address to the bug tracker, it creates a relative path to the URL in the form of a hyperlink and displays in the jira column.

- get_assignee: inserts a dropdown list and email button in each assignee custom column cell via HTML code. The dropdown list contains all relevant company developer names and aliases for triage purposes. Furthermore, these individuals are grouped according to the company branch they belong to. The email button, when pressed, generates an action event which invokes a function that enables users to use EMan's built-in email functionalities to notify developers about their assignment to issues;

- open_email_window: function invoked by the action event of pressing the email button created by the get_assignee function. Opens EMan's email window made up of three text areas destined for the email address, subject and body information. Buttons for closing the window and sending the email are also included in this window. Additionally, it automatically forms and introduces the email address in the window by merging the individual's name (selected from the dropdown list) with the predefined @synopsys.com sign and domain name. Finally, upon data validation, it uses Asynchronous JavaScript and XML (AJAX) in order to effectively send emails via POST requests, informing the user on the success/failure of this task;

- check_email_info: function called by the open_email_window function in order to validate the recipient and overall introduced email address. Firstly, in order to validate the recipient, the check_email_info function checks if the address' recipient (this is, the username of the address) exists by checking that its length is greater than zero. Secondly, in order to check the overall validity

of the email address including the sign and domain name, regular expressions are used. This ensures that predefined email address syntax is not violated.

Finally, custom values obtained by means of the custom columns introduced into the UI may be saved into the relational database via built-in EMan provided function specific for this task (save_custom_values(save_obj)). However, it must be ensured that prior to this function use the custom values are inserted into an array of JSON objects, called save_obj, comprising the argument to this built-in function.

The purpose of this Chapter was to detail the several tasks performed in the implementation phase of the project. Firstly, the tasks regarding the environment setup were listed in order to enable the Execution Manager tool to work as intended. Secondly, carried out work regarding the configuration file system was addressed, which involved the conception of a Python script so as to act as a bridge between the regression information and the EMan tool where it was to be processed. Customisation of EMan's interface was the last topic discussed, which was implemented through JavaScript functions allied to HTML code in order to allow the interface to act in accordance to its wanted behaviour.

Chapter 6

Obtained Results and Validations

This chapter aims to illustrate the results obtained upon completing the implementation of the several stages of the project. Additionally, parallel to their presentation, validation regarding these results is also performed in order to ascertain if the intended functionalities of each feature are in working order and in accordance with the established goals.

6.1 Python script

Having implemented the Python script functions detailed in Subsection 5.2.2 of Chapter 5, validating their working order and intended purpose was the next step taken in order to achieve the goal of efficient configuration file creation. Checking for correct functionalities provided by the Python script involved validation in four major areas:

- script usage;
- regression file information extraction;
- testlist file information extraction;
- configuration file creation.

Thus, for the Python script to work as planned, functionality regarding these four tasks needed to be assured. Results regarding the performed validation in all four fields is further detailed in Subsections 6.1.1, 6.1.2, 6.1.3 and 6.1.4, respectively.

6.1.1 Usage

The first aspect of the script to be validated before any processing of information could be carried out was its command line usage. In order for the Python script or, for that matter, any script to function correctly, the proper arguments must be parsed through the shell's command line. In this case, and as is conveyed by the configuration file creation system of Figure 5.1, solely one argument is required, the regression file. Consequently, according to the scripts flowchart in Figure 5.5, parsing an incorrect number of arguments should prompt the display of a help menu. Therefore, a usage statement was formulated for this matter. Misusage of the script was deliberately carried out so as to test the developed parse_args function, previously detailed in Subsection 5.2.2. This prompted, as expected, the developed usage statement present in Figure 6.1.

```
*********************************************************************************************
Python_script.py usage statement
*********************************************************************************************
Usage:
        Python_script.py <csv regression file>
*********************************************************************************************
Arguments:
        "csv regression file"   : Mandatory argument depicting a csv regression file
                                  which includes the desired builds, corresponding
                                  testlists, external scripts and other regression options.
*********************************************************************************************
Example:
        Python_script.py regression_file.csv
*********************************************************************************************
```

Figure 6.1: Python script usage statement

The usage statement consists of a printed summary, informing users on how to invoke the Python script, detailing the contents of the single argument to be parsed and exemplifying a usage case. Furthermore, in order to provide further clarification, additional usage messages were devised informing the user of the nature of the error. These additional messages are represented in Figure 6.2 and were implemented so as to be displayed after the presented usage statement.

```
USAGE ERROR! Too many arguments parsed! Please consult the usage statement.
USAGE ERROR! Too few  arguments parsed! Please consult the usage statement.
```

Figure 6.2: Script misusage messages

6.1.2 Regression file information extraction

The next step taken in the validation process was checking the regression file information extraction method. This process consisted in confirming if the Python script could not only attain this information successfully but store it according to the intended Python data types. These data types, mentioned in the sort_regr_info function of Subsection 5.2.2, consist of a list made up of dictionaries which, as already mentioned, is called the build list. Each dictionary within the build list contains information regarding a single build.

Thus, in order to validate this process, test scenarios were devised in which regression files similar to the example of Figure 5.2 were parsed to the script. And, upon modifying the script so as to enable the visualisation of extracted information, the build lists of Figure 6.3 were obtained.

```
[
 {
  '<BUILD_NAME>': 'build1',
  'TESTLISTS': ['testlist_X', 'testlist_Y',
                'testlist_Z'],
  '<BUILD_OPTIONS>': '-option=an_option',
  '<SIM_OPTION>': ['-option=option_A', '', ''],
  '<ORIGINAL_BUILD_NAME>': '-build_name=build1'
 }
 ,
 {
  '<BUILD_NAME>': 'build2',
  'TESTLISTS': ['testlist_A', 'testlist_B'],
  '<BUILD_OPTIONS>': '',
  '<SIM_OPTION>': ['', '-option=option_B'],
  '<ORIGINAL_BUILD_NAME>': '-build_name=build2'
 }
]
```

(a) Unique build names

```
[
 {
  '<BUILD_NAME>': 'build1',
  'TESTLISTS': ['testlist_X', 'testlist_Y',
                'testlist_Z'],
  '<BUILD_OPTIONS>': '-option=an_option',
  '<SIM_OPTION>': ['-option=option_A', '', ''],
  '<ORIGINAL_BUILD_NAME>': '-build_name=build1'
 }
 ,
 {
  '<BUILD_NAME>': 'build1_1',
  'TESTLISTS': ['testlist_A', 'testlist_B'],
  '<BUILD_OPTIONS>': '',
  '<SIM_OPTION>': ['', '-option=option_B'],
  '<ORIGINAL_BUILD_NAME>': '-build_name=build1'
 }
]
```

(b) Duplicate build names

Figure 6.3: Build list with unique and duplicate build names

As seen in Figure 6.3, two separate regression files were tested, one with unique build names and another with repeated build names. Both these experiments were conducted in order to firstly assess if the information was stored according to the developed implementation and, if so, analyse the build name changing feature devised, hereby testing the sort_regr_info and check_repeated_builds functions. This caused the Python script to create the build lists present in Figures 6.3a and 6.3b, respectively. Within these, two dictionaries corresponding to the two builds declared in the regression files were stored, containing the intended key pair values in the form of the regression file column titles and row information for each build. Furthermore, testlist and optional column values were stored as lists, enabling the synchronisation of overriding options with the corresponding test cases. The differences between both build lists can be seen within the <BUILD_NAME> key's value of the second build. While this value remained unchanged for unique build names, in the case of duplicate build names (in which build1 was inserted twice in the regression

file), the Python script successfully altered it with a version number, turning it into build1_1, while simultaneously keeping the initial name by means of the <ORIGINAL_BUILD_NAME> key. In the case of repeated build names occuring, the check_repeated_builds function is also responsible for emitting a warning regarding the altered build name. The warning, as illustrated in Figure 6.4, also specifies the file in which the error was encountered (in this case, the parsed regression file was names Regression1.csv).

```
WARNING! Regression_1.csv: Repeated build name: "build1". New name given: build1_1.
```

Figure 6.4: Regression file repeated build name warning

A primary concern, however, was the consistent manner in which the information extraction process was performed. Further experiments were then carried out in order to test the full proof nature of the Python script. The first scenario of interest in this context regarded input handling, which was tested to check the scripts response to failures in outsourcing required files, such as the regression file, testlists and the *.emc template. This prompted the respective error messages shown in Figure 6.5.

```
FILE ERROR! Regression_1.csv: There was a problem opening this file!
FILE ERROR! Regression_1.csv: There was a problem opening testlist_X.csv!
FILE ERROR! Regression_1.csv: There was a problem opening template.emc!
```

Figure 6.5: Python script input file errors

Next up was the editing/removal of mandatory regression file columns. This triggered the Python script into acknowledging these inaccuracies and performing the required feedback to the user on the encountered problems. Once more, this was devised by means of custom error messages, detailing the missing mandatory column as shown in Figure 6.6.

```
SYNTAX ERROR! Regression_1.csv: Mandatory column "<BUILD_NAME>"    not in file!
SYNTAX ERROR! Regression_1.csv: Mandatory column "TESTLISTS"       not in file!
SYNTAX ERROR! Regression_1.csv: Mandatory column "<BUILD_OPTIONS>" not in file!
SYNTAX ERROR! Regression_1.csv: Mandatory column "EMC_TEMPLATE"    not in file!
SYNTAX ERROR! Regression_1.csv: Mandatory column "<BUILD_SCRIPT>"  not in file!
SYNTAX ERROR! Regression_1.csv: Mandatory column "<SIM_SCRIPT>"    not in file!
```

Figure 6.6: Regression file missing mandatory columns syntax errors

The last test performed in the context of this subsection was the purposeful insertion of incorrect row information within each row of the regression file. This was done with the intention of violating the established rules described in Table 5.1. This enabled the operation of the check_regr_errors function which provided the error messages presented in Figure 6.7.

```
SYNTAX ERROR! Regression_1.csv: "<BUILD_NAME>"    column information missing in the first row.
SYNTAX ERROR! Regression_1.csv: "TESTLISTS"       column information missing for build1.
SYNTAX ERROR! Regression_1.csv: "<BUILD_OPTIONS>" column information entered multiple times in build1.
SYNTAX ERROR! Regression_1.csv: "EMC_TEMPLATE"    column information missing in the first row.
SYNTAX ERROR! Regression_1.csv: "<BUILD_SCRIPT>"  column information missing in the first row.
SYNTAX ERROR! Regression_1.csv: "<SIM_SCRIPT>"    column information missing in the first row.
```

Figure 6.7: Regression file information syntax errors

6.1.3 Testlist file information extraction

The third field requiring testing within the Python script was testlist information extraction. Similarly as to the regression file experiments, the main goal was to certify the attaining and correct storage of information according to the intended data types implemented via the sort_testlist_info function.

Two further test scenarios were then implemented, in which similar testlists to the one presented in Figure 5.3 were referenced in the regression file (via the TESTLISTS column). And, once these testlist's names were present in the builds list, this allowed for their outsourcing by the Python script enablig the creation of the test lists of Figure 6.8.

```
[
 {
  '<TEST_NAME>': 'test1',
  '<N_RUNS>': '8',
  '<SIM_OPTION>': '-option=option1',
  '<BUILD_NAME>': 'build1',
  '<ORIGINAL_TEST_NAME>': '-test=test1'
 }
 ,
 {
  '<TEST_NAME>': 'test2',
  '<N_RUNS>': '12',
  '<SIM_OPTION>': '',
  '<BUILD_NAME>': 'build1',
  '<ORIGINAL_TEST_NAME>': '-test=test1'
 }
]
```

(a) Unique test names

```
[
 {
  '<TEST_NAME>': 'test1',
  '<N_RUNS>': '8',
  '<SIM_OPTION>': '-option=option1',
  '<BUILD_NAME>': 'build1',
  '<ORIGINAL_TEST_NAME>': '-test=test1'
 }
 ,
 {
  '<TEST_NAME>': 'test1_1',
  '<N_RUNS>': '12',
  '<SIM_OPTION>': '',
  '<BUILD_NAME>': 'build1',
  '<ORIGINAL_TEST_NAME>': '-test=test1'
 }
]
```

(b) Duplicate test names

Figure 6.8: Test lists with unique and duplicate test names

It is important to note the difference between a testlist and the test list. Although they possess similar names, the former is a file containing test case information while the latter is the created Python list containing the extracted test case information from one or more of these testlists. By viewing Figure 6.8 it can be seen that the two conducted tests were done through testlists solely composed of two tests. The first, which allowed for the creation of the test list of Figure 6.8a, contained test cases with unique names, while the second contained repeated test names, creating the test list of Figure 6.8b. The notable difference between both test lists is the name of the second test case, which, for the duplicate test name scenario was altered

by the check_repeated_tests function. On the other hand, the initial test name was kept intact through <ORIGINAL_TEST_NAME>, allowing its use with the in-house simulation script. For situations in which repeated test names occur, the check_repeated_tests function also displays the user with a warning regarding the altered test name, as shown in Figure 6.9.

```
WARNING! testlist_X.csv: Build "build1": Repeated test name "test1". New name given: test1_1.
```

Figure 6.9: Testlist file repeated test name warning

Full proof testing of the Python script was also conducted with regards to testlist information, following a similar order as to what was done with the regression file. Firstly, mandatory column titles were altered, testing the script's recognition abilities in detecting these mistakes. This triggered custom messages detailing the missing columns, as is shown through Figure 6.10.

```
SYNTAX ERROR! testlist_X.csv: Mandatory column "<TEST_NAME>" not in file!
SYNTAX ERROR! testlist_X.csv: Mandatory column "<N_RUNS>"    not in file!
```

Figure 6.10: Testlist file missing mandatory columns' syntax errors

Lastly, information to voluntarily infringe the testlist file rules of Table 5.2 was inserted in the testlists. Additionally, for the particular case of the <N_RUNS> column, information not corresponding to integer numbers was also inserted, since the number of runs must be an integer. Once these scenarios were tested, the custom error messages of Figure 6.11 were activated.

```
SYNTAX ERROR! testlist_X.csv: "<TEST_NAME>" column information missing!
SYNTAX ERROR! testlist_X.csv: "<N_RUNS>"    column information missing!

SYNTAX ERROR! testlist_X.csv: "<N_RUNS>" must be an integer number! "one" is not.
SYNTAX ERROR! testlist_X.csv: "<N_RUNS>" must be an integer number! "9.5" is not.
```

Figure 6.11: Testlist file incorrect information syntax errors

6.1.4 Configuration file creation

The final area requiring validation within the Python script domain was the configuration file creation process. The configuration file, as explained in Subsection 5.2.3 of Chapter 5, should be created in order to contain the correct number of builds and tests mappings according to the information passed in the regression file and testlist files, respectively. Furthermore, CSV column title matching strings present in the *.emc template file, exemplified in Figure 5.6, should be replaced. In order to test if these goals were achieved, all required inputs were provided to the

Python script to confirm correct configuration file creation. The obtained output configuration file was analogous to the one presented in Figure 6.12.

```
config_name: Regression_1

variables:

  - current_dir: $PWD

base_build:

  - name: base_build
    error_patterns:
        - ^.*BUILD FAILED:::fail:::build failed:::0
        - ^.*BUILD SUCCEEDED:::pass:::build successful:::0
    abstract: yes

base_test:

  - name: base_test
    error_patterns:
        - ^UVM_ERROR.*@\s\d+\.\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
        - ^UVM_FATAL.*@\s\d+\.\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
        - ^UVM_INFO.*@\s\d+\.\d+ns:\s(?P<ERROR>.*):::pass:::(ERROR):::0
    abstract: yes

builds:

  - name: build1
    run_cmd: build_script.py -build_name=build1 -option=an_option
    run_dir: $current_dir
    extends: base_build

  - name: build2
    run_cmd: build_script.py -build_name=build2
    run_dir: $current_dir
    extends: base_build

tests:

  - name: test1
    build: build1
    run_cmd: sim_script.py -test=test1 -option=optionA
    run_dir: $current_dir
    count: 8
    extends: abstract/base_test

    (...)

  - name: test4
    build: build1
    run_cmd: run_tests.csh sim_script.py -test=test4 -option=option2
    run_dir: $current_dir
    count: 1
    extends: abstract/base_test
```

Figure 6.12: Obtained configuration file (simplified)

The configuration file example of Figure 6.12 was run with the regression file of Figure 5.2 and the testlist of Figure 5.3 (which we will consider as testlist_X). However, only two tests mappings are presented, namely the first and last test cases of the mentioned testlist. Other than for simplicity's sake, this was done

in order to show two examples regarding overriding options. While test1 represents a tests mapping in which the <SIM_OPTION> was overridden by the regression file (with "-option=optionA"), test4 represents a mapping in which overriding <SIM_OPTION> information was not passed to the test cases of testlist_X (maintaining the "-option=option2" specified in the testlist). Furthermore, the <CONFIG_NAME>, is given the same name as the regression file filename which in this case was Regression_1.csv. Furthermore, a configuration file with repeated build and test names is illustrated in Appendix A to show an example of this kind. Once obtained a successful configuration file creation, the Python script outputs a success message to the user, shown in Figure 6.13.

```
EMC CONFIGURATION FILE SUCCESSFULLY CREATED!

 Configuration file name: Regression_1.emc
```

Figure 6.13: Successful configuration file creation output message

As expected, the filename is given based on the regression file name parsed to the Python script. Therefore, using the regression file Regression_1.csv outputs the same filename but with the .emc extension (Regression_1.emc).

The last test scenarios performed with the Python script involved the *.emc template file strings. The previously presented configuration file was obtained considering a correct template file in which all strings matching column titles were present and ready for the information mapping process. However, to test the scripts' robustness, these strings were removed from the template file one by one in order to assess the corresponding behaviour to these events. The obtained messages from this experiment are illustrated in Figure 6.14.

```
SYNTAX ERROR! template.emc: "<BUILD_NAME>"    string not in build mapping!
SYNTAX ERROR! template.emc: "<BUILD_OPTIONS>" string not in build mapping!
SYNTAX ERROR! template.emc: "<BUILD_SCRIPT>"  string not in build mapping!

SYNTAX ERROR! template.emc: "<TEST_NAME>"  string not in test mapping!
SYNTAX ERROR! template.emc: "<N_RUNS>"     string not in test mapping!
SYNTAX ERROR! template.emc: "<SIM_OPTION>" string not in test mapping!
SYNTAX ERROR! template.emc: "<BUILD_NAME>" string not in test mapping!
SYNTAX ERROR! template.emc: "<SIM_SCRIPT>" string not in test mapping!

SYNTAX ERROR! template.emc: "<CONFIG_NAME>"         string not in config mapping!
SYNTAX ERROR! template.emc: "<ORIGINAL_BUILD_NAME>" string not in build mapping!
SYNTAX ERROR! template.emc: "<ORIGINAL_TEST_NAME>"  string not in test mapping!
```

Figure 6.14: Missing *.emc template file strings syntax error messages

Thus, the Python script was able to determine the absence of the mentioned strings in all mapping types, feeding this information back to the user and notifying him about the specific missing string.

6.2 Pattern detection

Another validation process to accomplish was the assurance of correctly working error patterns. As was seen in Subsection 4.1.2 of Chapter 4, error patterns aid EMan in determining the status of a build or test case. Furthermore, it enables EMan to transmit this information to users through its status column present in the UI tables by checking for patterns in the log files created by the in-house scripts through means of regular expressions. Therefore, this process was observed for the used error patterns declared in the configuration file. An example of one of these error patterns used for detecting build statuses is present below:

```
^.*BUILD SUCCEEDED:::pass:::build successful:::0
```

The regular expression forming the pattern field in the error pattern aims to check every build log file string starting at the beginning of every line (^), then skipping any single character (.) zero or more times (*) up until the desired string "BUILD SUCCEEDED" is found. And, once having run a regression in which a successful compiled build was obtained, the log file for that build was analysed, having found the expected detected pattern present in Figure 6.15.

```
4421.   ******************
4422.   SUMMARY OF SUMMARY
4423.   ******************
4424.
4425.   Total verif warnings: 29
4426.   Total verif    lints: 151
4427.
4428.    **** BUILD SUCCEEDED - WITH WARNINGS    ****
4429.
```

Figure 6.15: Detected pattern in build log file

Finally, so as to confirm that the base_status and custom_status were working, the UI's builds table present in the web page hierarchy was consulted, with regards to the status column. There, the results of Figure 6.16 were presented.

Name	Status
build_1	pass

(a) Default status

Name	Status
build_1	build successful

(b) Custom status

Figure 6.16: Status change based on detected error pattern

It was then seen that, an error pattern in which the custom_status field was left blank produced the result viewed in Figure 6.16a where the base_status was presented to the user with the respective colour coded cell. On the other hand, an error_pattern with a declared custom_status presented it to the user with the same colour coded cell associated with the base_status (Figure 6.16b).

However, as seen in the *.emc template file of Figure 5.4, declared error patterns relating to tests are inherently different, as shown with the example below:

```
^UVM_ERROR.*@\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
```

As seen, a named capturing group of the form of (?P<ERROR>) is used. The named group "ERROR" is then declared so as to contain the relevant error message string, which is then applied to the custom_status. From here, similarly as with the build error pattern, EMan displays the error message string in the status column of the web page's UI (this time through the tests table). Thus, as will be subsequently seen in Section 6.3, error signatures may be formed based on these custom statuses.

6.3 UI customisation

Having confirmed the functionality regarding the Python script as well as the error patterns, the last implementation phase left requiring validation was the performed UI customisation. In order to validate the customised UI's functionalities, requirements for the solution's interface, previously delineated in Section 3.2, were taken into account. Therefore, the first of the mentioned requirements was to produce error signatures based on test errors which, as explained in the closing stages of Section 6.2, are obtained through the custom_status field of the test's error patterns. Once possessing the custom test error messages, these were passed by EMan into the database to be displayed in the status column of the EMan UI's test table. By inputting these errors to the signature mapping of the *.emd template file, as previously seen in Figure 5.8, allowed the get_signature function to provide the information present in Figure 6.17.

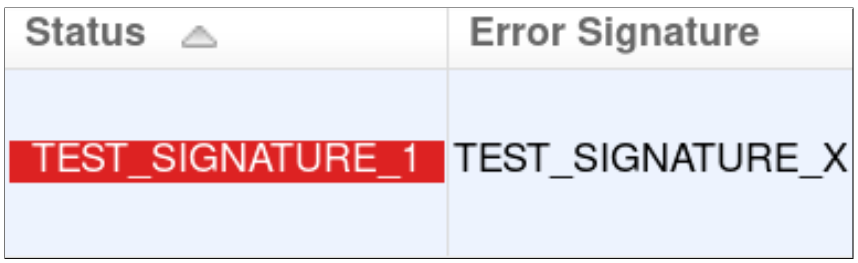
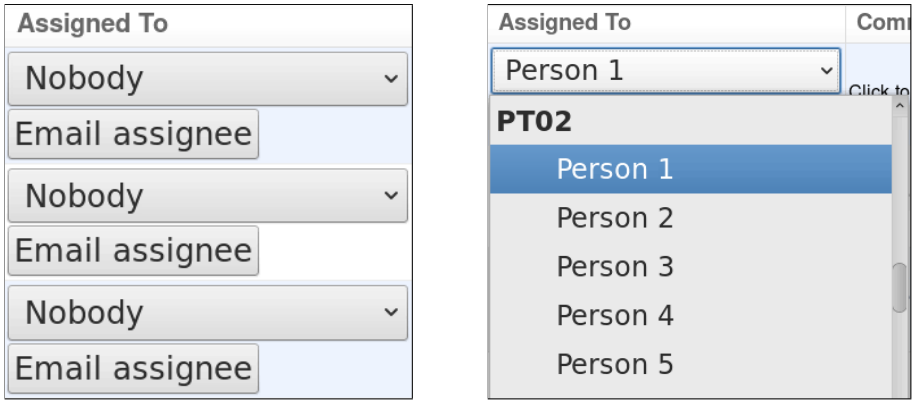


Figure 6.17: Status and error signature columns

By visualising Figure 6.17, two distinct columns can be seen. The first corresponds to the built-in status column, responsible for presenting the mentioned custom test error messages, while the second constitutes the error signature custom column. As can be seen, the error signature differs from the status in that it replaces all its digits, creating a general string that similar errors may also match, thus enabling their future binning.

The next requirements taken into account when customising the UI were the assignment to developers to specific error signatures as well as their notification via email functionalities. For this reason, as illustrated in the *.emd template file of Figure 5.8, the assignee mapping was declared, enabling the creation of the custom column present in Figure 6.18.



(a) No assignee (b) Assignee selection

Figure 6.18: Assign To column

This column, as seen in Figure 6.18a, contains two elements for each error signature cell, enabled through the get_assignee function detailed in Subsection 5.3.2. These elements consisted of a dropdown menu in which to assign developers to the corresponding row's error signature, as well as a button for email sending. With respect to these elements, and as seen in Figure 6.18b, the dropdown presents users with a series of developer names as well as the branch to which they belong (which in this case consists of branch PT02). On the other hand, by clicking the provided "Email assigneee" button, EMan's built-in email window was shown, as illustrated through Figure 6.19, inserting the selected developers' email address automatically and enabling their notification via this method. Furthermore, in order to validate the working order of EMan's email capabilities when integrated with the developed implementation, a test email was sent to a valid company address. This email, as presented by Figure 6.20, was successfully sent to the rightfully assigned individual, containing the correct subject and email body inserted by means of the email window.

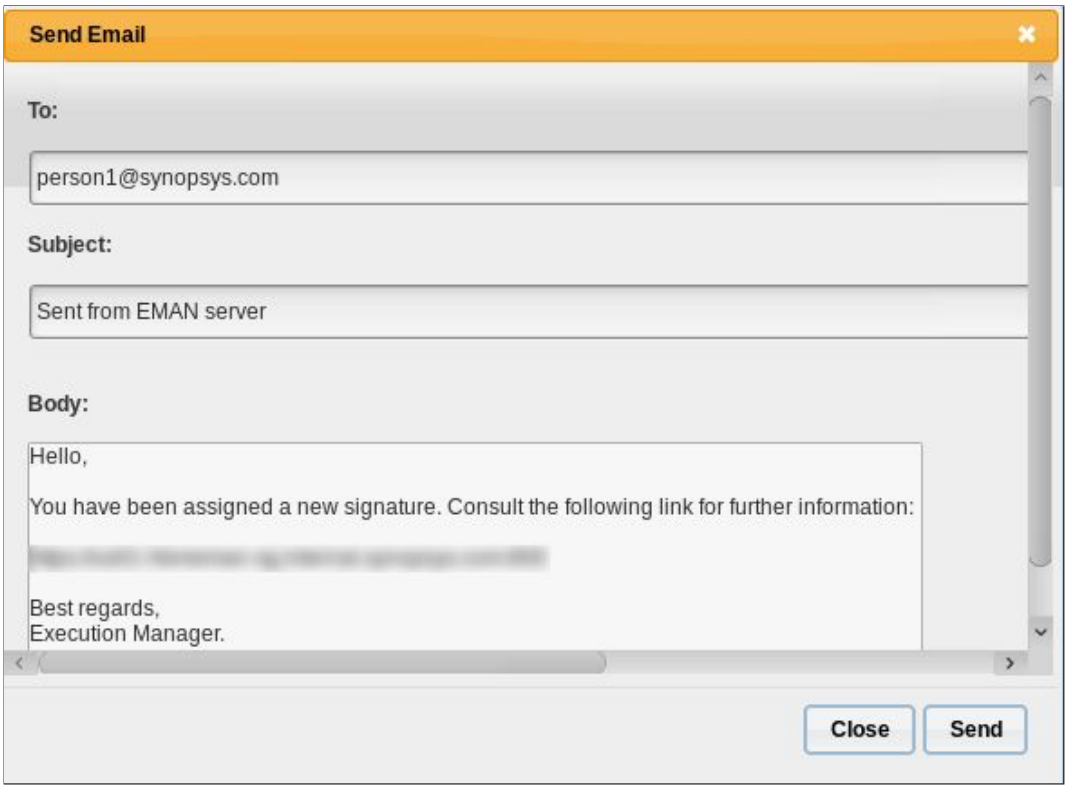


Figure 6.19: EMan's email window

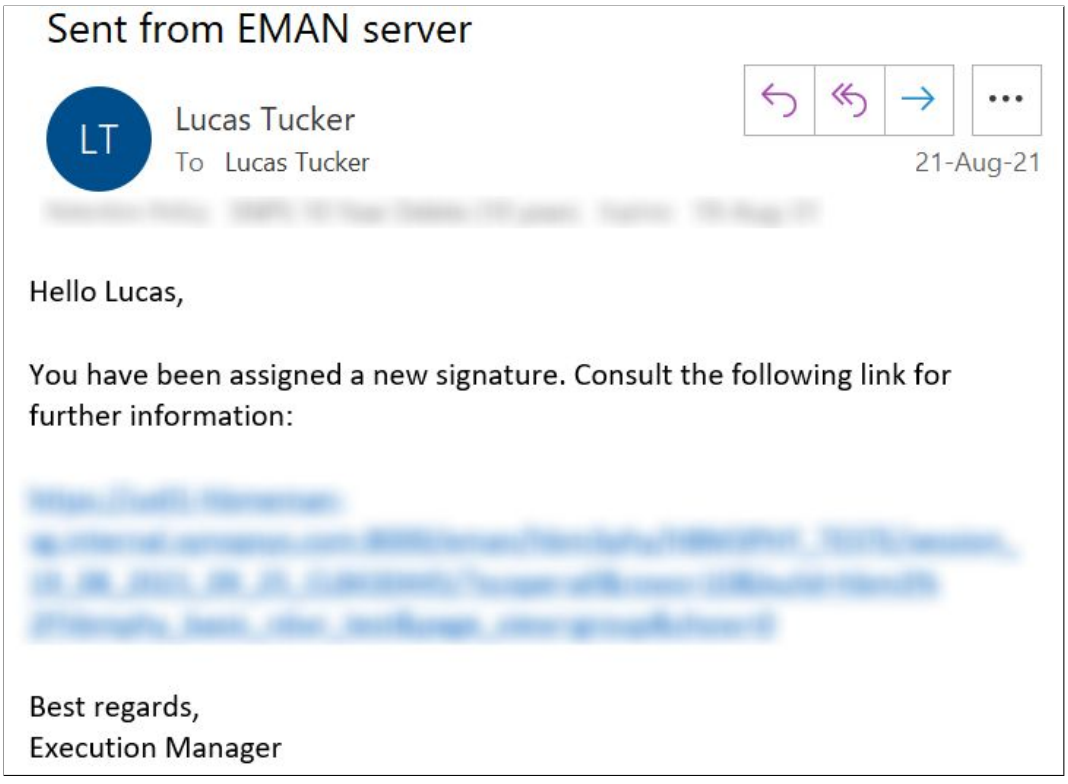
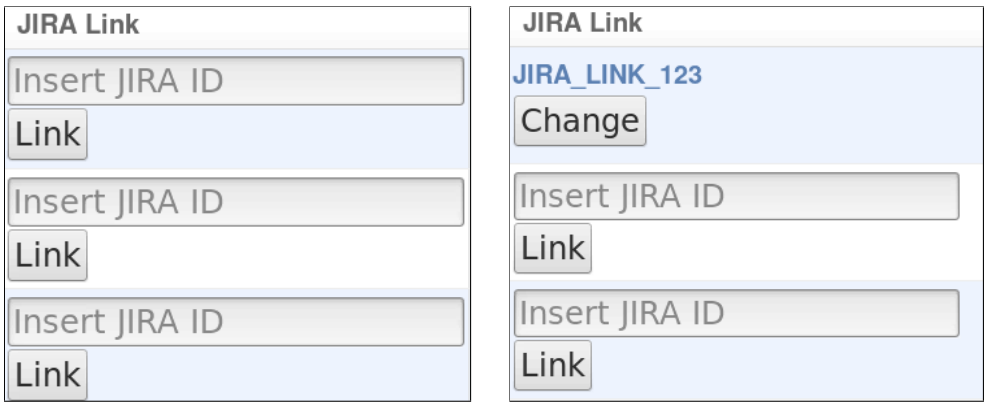


Figure 6.20: Received email

Next on the requirements list was the assignment of error signatures to a corresponding bug tracker link. Once again through the *.emd template file, the declared jira mapping alongside its referenced get_jira_link enabled the creation of a new custom column to be inserted into EMan's test tables, illustrated in Figure 6.21.

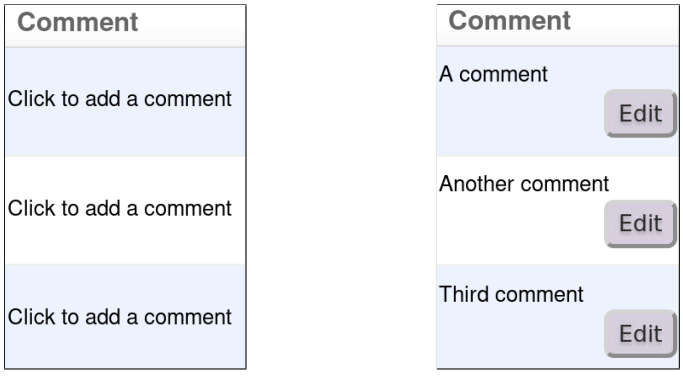


(a) No inserted ID (b) Generated hyperlink

Figure 6.21: Jira Link column

As seen, the get_jira_link, responsible for managing the jira link custom column's interactive features, accommodates for a test box as well as a button within the custom column's cells, as seen through Figure 6.21a. These allow for inserting an ID as well as generating the hyperlink to the bug tracker software, respectively, as seen in Figure 6.21b. Therefore, both this column as well as the Assign To column allowed for on the fly changes through the introduced HTML elements.

Lastly, a final built-in column was also added to the EMan's test tables in order to integrate user comments within the interface. This column, present in Figure 6.22, allows for developers to simply click on its predefined text (Figure 6.22a) to insert the wanted comment and, additionally, allows for comment editing as shown through Figure 6.22b.



(a) No comment inserted (b) Comments inserted

Figure 6.22: Comment column

Thus, once all custom columns were created containing all required information and features, the test table from EMan's page hierarchy was arranged to as to appear similar to the former solution's failure signature table, previously shown in Figure 3.1. A new failure signature table was then obtained for the EMan tool, as can be seen in Figure 6.23.

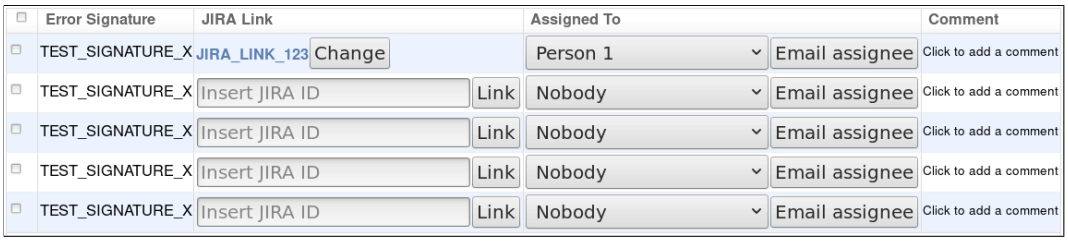


Figure 6.23: Final failure signature table obtained for EMan

With respect to this Chapter, its main objective was to present the obtained results stemming from the performed implementation. Thus, it was organised into three main areas of focus. The first area consisted of the developed Python script, which was thoroughly tested in order to ascertain its full proof nature when faced with invalid or nonexistent information. The results for this script showed that it was able to achieve its intended goal when correctly used alongside inputs and provide users with custom fed back messages detailing the nature of found errors. The second area of focus was the validation of pattern detection, implemented through means of regular expressions. Results obtained in this field showed that these were in working order, transmitting the correctly configured statuses to users regarding builds and tests. The last area of interest in this Chapter were the results associated to EMan's UI customisation. Here, all functionalities regarding newly created columns to be integrated in EMan's tables were validated, allowing for the creation and usage of a final signature table.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Having finished the project and reviewing the established requirements for the solution, it was concluded that these were all successfully achieved. With respects to efficient regression running, this was accomplished through the developed Python script, which acted as a bridge between the regression information to be run and the actual configuration file to be submitted to the Execution Manager's regression engine. The formation and assignment of error signatures to developers as well as bug tracker hyperlinks was also achieved through the customisation of EMan's UI, which was carried out through the addition of custom columns to the web page's test tables. Furthermore, the interactive features incorporated within these columns allowed for on the fly changes to their information, as well as the integration of email functionalities, consisting of another two requirements for the solution. Finally, through the organisation of the EMan UI's information, similarity between the previous and newly devised solution's failure signature table was attained, allowing for an expected straightforward switch between solution on the part of the verification team at Synopsys. Therefore, eliminating the previous solution's in-house nature through the application of a widely used tool in the industry (Execution Manager), allowed for the elimination of maintenance costs and efforts associated to it.

7.2 Future work

Although all established requirements were achieved, it was recognised that additional work could be carried out in order to further enhance the newly developed solution.

Thus, the binning of obtained test error through means of their error signatures was a feature considered invaluable in the continuity of this project, allowing for test errors with the same possible root cause to be grouped together and subsequently debugged by the most adequate individual. This particular feature would then require storing all error signatures in EMan's relational database and implementing a search method in order to ascertain if new errors fall into an already established signature. Additionally, the integration of further external tools, such as Synopsys' Verdi for automated analysis and debug, were also considered to be a major contribution in the enhancement of the project.

References

[1] V. Bertacco, *Scalable hardware verification with symbolic simulation*. Springer, 2006. [Cited on pages vii, 6, 7, and 10]

[2] C. Spear, *SystemVerilog for verification: A Guide to Learning the Testbench Language Features*. 2006. [Cited on pages vii, 9, 10, and 11]

[3] M. Cieplucha, "Metric-Driven Verification Methodology with Regression Management," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 35, no. 1, pp. 101–110, 2019. [Cited on pages vii, 13, and 14]

[4] Z. Poulos and A. Veneris, "Failure triage in RTL regression verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1893 – 1906, 2018. [Cited on pages vii, 5, 15, and 16]

[5] Synopsys, "VC Execution Manager User Guide," 2021. [Cited on pages vii, 24, and 25]

[6] L. M. G. Gomes, "Power Reduction of a CMOS High-Speed Interface Using Power Gating," M.S. thesis, DEEC, FEUP, Porto, Portugal, 2013. [Online]. Available: `https://paginas.fe.up.pt/~ee07306/wp-content/uploads/2013/03/dissertation_lgomes_ee07306.pdf"`. [Cited on page 7]

[7] S. Ikram and J. Ellis, "Dynamic Regression Suite Generation Using Coverage-Based Clustering," *Proceedings of the design and verification conference and exhibition US (DVCon)*, no. February, 2017. [Cited on pages 10, 12, and 13]

[8] Z. Poulos and A. Veneris, "Clustering-based failure triage for RTL regression debugging," *Proceedings - International Test Conference*, vol. 2015-Febru, pp. 1–10, 2015. [Cited on pages 10, 15, and 16]

[9] M. R. Kakoee, M. Riazati, and S. Mohammadi, "Generating RTL Synthesizable Code from Behavioral Testbenches for Hardware-Accelerated Verification," *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pp. 714–720, 2008. [Cited on page 10]

[10] J. Bergeron, *Writing Testbenches using SystemVerilog*. 2006. [Cited on pages 10, 11, 12, and 13]

[11] R. Jindal and K. Jain, "Verification of transaction-level SystemC models using RTL testbenches," *Proceedings - 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE 2003*, pp. 199–203, 2003. [Cited on page 10]

[12] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification Methodology Manual for SystemVerilog*. 2006. [Cited on pages 10 and 11]

[13] P. Kandil, S. Moussa, and N. Badr, "A Study for Regression Testing Techniques and Tools," *International Journal of Soft Computing and Software Engineering (JSCSE)*, vol. 5, no. 4, pp. 64–84, 2015. [Cited on page 11]

[14] F. M. M. Fonseca, "A Parallel Execution Approach for Efficient Regression Testing in the OutSystems Test Infrastructure," pp. 1–10, M.S. thesis, DEI, IST-UTL, Lisboa, Portugal, 2009. [Online]. Available: `https://fenix.tecnico.ulisboa.pt/downloadFile/395139413927/dissertacao.pdf`". [Cited on page 11]

[15] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica (Ljubljana)*, vol. 35, no. 3, pp. 289–321, 2011. [Cited on pages 12 and 13]

[16] R. Kazmi, D. N. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys*, vol. 50, no. 2, 2017. [Cited on page 13]

[17] Z. Poulos, Y. S. Yang, A. Veneris, and B. Le, "Simulation and satisfiability guided counter-example triage for RTL design debugging," *Proceedings - International Symposium on Quality Electronic Design, ISQED*, pp. 618–624, 2014. [Cited on pages 15 and 16]

[18] "RegExr: Learn, Build, & Test RegEx." `https://regexr.com/` (accessed: Oct. 31, 2021). [Cited on page 27]

[19] "The Official YAML Web Site." `https://yaml.org/` (accessed: Oct. 31, 2021). [Cited on page 28]

Appendix A

Configuration file output with repeated build and test names

The output configuration file contents presented below aim to further clarify the importance of the <ORIGINAL_BUILD_NAME> and <ORIGINAL_TEST_NAME> *.emc template strings. These strings allow builds and tests with repeated names to maintain their initial names, even though altered in the context of EMan's UI, for application in the in-house script through means of the run_cmd mapping:

```
config_name: Regression_1

variables:

  - current_dir: $PWD

base_build:

  - name: base_build
    error_patterns:
    - ^.*BUILD FAILED:::fail:::build failed:::0
    - ^.*BUILD SUCCEEDED:::pass:::build successful:::0
    abstract: yes

base_test:
```

```
  - name: base_test
    error_patterns:
    - ^UVM_ERROR.*@\s\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
    - ^UVM_FATAL.*@\s\d+ns:\s(?P<ERROR>.*):::fail:::(ERROR):::0
    - ^UVM_INFO.*@\s\d+ns:\s(?P<ERROR>.*):::pass:::(ERROR):::0
    abstract: yes

builds:

  - name: build1
    run_cmd: build_script.py -build_name=build1 -option=an_option
    run_dir: $current_dir
    extends: base_build

  - name: build1_1
    run_cmd: build_script.py -build_name=build1
    run_dir: $current_dir
    extends: base_build

tests:

  - name: test1
    build: build1
    run_cmd: sim_script.py -test=test1 -option=optionA
    run_dir: $current_dir
    count: 8
    extends: abstract/base_test

  - name: test1_1
    build: build1
    run_cmd: run_tests.csh sim_script.py -test=test1 -option=
        option2
    run_dir: $current_dir
    count: 1
    extends: abstract/base_test
```