Suite File Generation for Test Automation Framework

A project thesis

submitted by

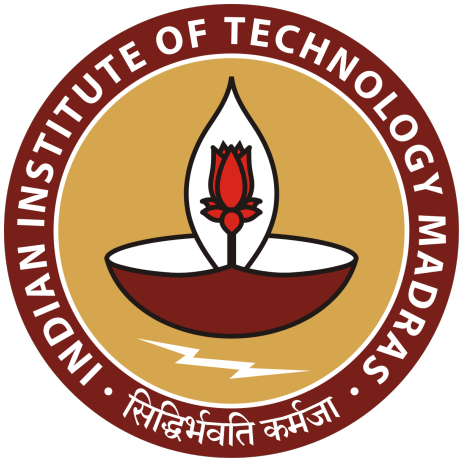
Mittapalli Harshavardhan

ee19m021

in partial fulfillment of the requirements
for the award of the degree of

Master of Technology

June 2021



Dept. of Electrical Engineering
IIT Madras
Chennai 600 036

Thesis Certificate

This is to certify that the thesis (or project report) titled **Suite File Generation for Test Automation Framework**, submitted by **Mittapalli Harshavardhan** bearing roll no.**EE19m021**, to the Indian Institute of Technology Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work done by him (her) under my (our) supervision.The contents of this thesis (or project report), in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Chennai

Date: 28th June 2021

V Kamakoti
Guide
Professor
Dept. of Computer Science Engineering

Avhishek Chatterjee
Co-Guide
Assistant Professor
Dept. of Electrical Engineering

Acknowledgements

I express my deep sense of gratitude to all those who have been instrumental in preparation of this project. I'm very much thankful to **Prof V Kamkoti** ,for accepting to guide me for this Project.

I also extend my gratitude to **Prof Avhishek Chatterjee** who supported me as a co-guide from electrical department.

I acknowledge the kind of support,efforts and timely guidance provided by my mentor **Anand Kumar S**,Pricipal Project Staff, RISE lab,IIT Madras whose guidance, encouragement,suggestions and very constructive ideas have contributed immensely in progress of this project.

Abstract

KEYWORDS: SoC,Features;Test cases;Suite file

.

A System on Chip (SoC) is a single chip that contains the memory, processor, and input/output controllers.In simple terms, a System on Chip, or SoC, is an integrated circuit, or IC, that integrates a whole electronic or computer system onto a single substrate.A central processor unit, input and output ports, internal memory, and analogue input and output blocks are just a few of the components that a SoC tries to contain within itself.Signal processing, wireless communication, artificial intelligence, and other operations can be performed depending on the type of system that has been shrunk to the size of a chip.

Serial communication protocols like UART,I2C,SPI and others are referred as features. Each feature has it's own set of testcases.A suite file is a JSON format file which contains the parameters of test cases of particular feature or single test case.Generation of a suite file has to be done either by entering one or more features manually or after noticing any modifications (or newly added) test cases to features. With this suite file as input, testcases are executed and a html report is generated at the end of the run

Abbreviations

SoC	System on Chip
JSON	Java Script Object Notation
RISC	Reduced Instruction Set Computer
GDB	Gnu Debugger
OCD	On Chip Debugger
UART	Universal Asynchronous Receiver Transmitter
I2C	Inter Integrated Circuits
SPI	Serial Peripheral Interface
GPIO	General Purpose Input Output
PWM	Pulse Width Modulation

Contents

Chapter 1

Introduction

The Reconfigurable Intelligent Systems Engineering (RISE) group at IIT-Madras created SHAKTI as an open-source project.The SHAKTI project is developing a family of six processors based on the RISC-V instruction set architecture.Based on the RISC-V ISA, the project has developed an embedded class (named E-Class) and a controller class (called C-Class) of processors. Also pronounced as "risk-five",RISC-V is a free and open Instruction Set Architecture (ISA) enabling a new era of processor innovation through open standard collaboration.

1.1 RISC-V ISA Overview

RISC-V was created with the intention of assisting computer architecture research and education.However, it has now become the industry's standard free and open architecture.The RISC-V ISA is made up of a base integer ISA that must be present in any implementation, as well as optional extensions to the base ISA.The width of the integer registers and the matching size of the address space, as well as the number of integer registers, define each base integer instruction set.Little-endian RISC-V is available in 32 and 64 bit versions.Int is 32 bits for each.The native register size is pointers and long.In a larger register, signed values are always sign extended.Values in the unsigned 8/16-bit range are zero extended.Sign-extension is applied to unsigned 32-bit data.RISC-V was created to allow for a great deal of customization and specialization. User level Spec and Privilege level Spec are the two volumes of the RISC-V specification.

1.2 Development environment

The tool chain installation required and further steps are followed as per reading on "Learn with Shakti" Development Environment[1]. Also the steps for cloning "Shakti-SDK" repository are followed as per the instructions provided on the page

Setting up Shakti-SDK[2]. Very detailed explanation about the same is mentioned in Shakti-soc-user-manual[3] under section 5.2. After installation a sample program "hello.c" was run to see what are the steps followed for compiling and execution of .c program.

1.3 Why Test Automation?

Manual testing is meticulously carrying out predetermined test cases, comparing the outcomes to the intended behavior, and documenting the findings.Manual testing is time-consuming and prone to errors because it is repeated every time the source code changes.Execution on many platforms is equally tough.
Automation testing is the use of tools and technologies to test software in order to reduce testing efforts and deliver functionality faster and more affordably.It aids in the development of higher-quality software with less effort.Once automated tests are built, they may be readily replicated and extended to perform things that would be impossible to accomplish manually.

1.3.1 Reduces the risks of Failure

The risk of bugs leaking into the field is reduced by a robust set of test suites that are run periodically each time the code is changed. Automated testing aid in the early detection of flaws in software development, lowering the risk of providing incorrect software.

1.3.2 Save Time

While the initial setup of automated testing cases takes a lot of time and work, you may reuse these tests once they've been automated. Automated tests are substantially faster than manual testing, have less errors, and require less effort.Automated tests can be run frequently once the setup is complete, cutting the time it takes to run repetitious manual tests from weeks to hours.

1.3.3 Accuracy and Reliability

After all, manual testers are humans, thus mistakes are to be expected. This could lead to the development team receiving erroneous results. Automated tests repeat the same steps with pinpoint accuracy every time. In most cases, the results are made available to all parties involved in the shortest period possible.

1.4 Problem statement

Given a feature for a SOC, there will be a set of test cases for that particular feature or we can say a feature determines test cases.A suite file is collection of test cases.This project focus is to identify the set of test cases and generate the suite file which contains those test cases,then execute those test cases and generate a html report at the end. This problem statement of generating suite file and execution is the initial step in the high level design of *Test Automation Framework*.

1.5 Approach

The approach is dividing into two parts:

- **Identification:** First we have to generate the suite file which consists the test cases for that particular feature of SOC. A test case will have set of inputs, pass or fail criteria and setup. Based on these suite file is generated.
- **Execution:** Once suite file is generated next task is to execute the test cases in that suite file. Libraries will have test case execution steps. A html report is generated at the end.

  The generated report results are analysed in the final conclusion.

1.6 Introducing JSON

Since the entire focus of this project is to generate a suite file which is ***.json*** format file. It becomes important to understand the structure of ***.json*** file.
JSON (JavaScript Object Notation) is an information trade design that is easy to use.Reading and composing are straightforward undertakings for people. Machines can without much of a stretch parse and produce it.It depends on a subset of the ECMA-262 third Edition - December 1999 JavaScript Programming Language Standard.JSON is a language-free content configuration that fuses norms regular to software engineers of the C-group of dialects, like C, C++, Java, JavaScript, Perl, Python, and numerous others.JSON is an optimal information move language as a result of these attributes.

JSON is made up of two distinct structures:

- Name/value sets are assembled together in this group. This is represented as an object, table, dictionary,struct, keyed list,hash or associative array in many languages.

- A set of values in a specific order.This is implemented as an array, vector, list, or sequence in most programming languages.

These are information structures that are widespread.They are upheld in some structure or another by practically all advanced writing computer programs languages.It's just normal that an information design that can be utilized with various codings depends on these constructions.

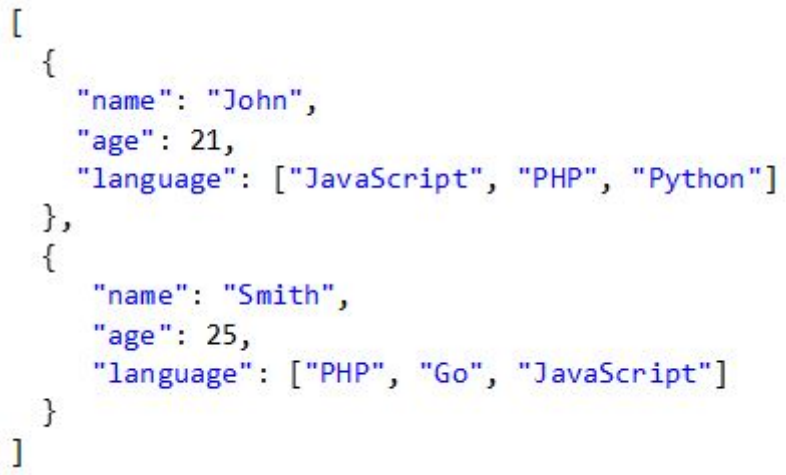


Figure 1.1: Example showing JSON file structure (with arrays) .

For better understanding of a JSON structure one can take the help of Introducing JSON[4]

Chapter 2

Implementation and Algorithm

Test automation frame work is a design to be used for automating testing of SoCs based on the Shakti processor so that time to test is reduced and number of test cases executed can be increased. This chapter deals with steps to generate a suite file and execution of testcases in the suite file.

2.1 Functionalities of the framework

- Select the list of testcases
- Test parameters
- Generate suite file
- Execute tests

2.1.1 Selection of testcases

As stated before, Every feature has its set of testcases.So to identify testcases we need features first. These features are fed to program either manually by user or from the ***git log*** command as shown in Fig 2.1 in the next page. Once the features are known,a list is prepared which comprises of all the test cases of the features.This testcases list serves as one of the parameters in the suite file.Preparation of testcases list has to be done in such a way that it is dynamic and must be able to keep track all the test cases of a particular feature and append the testcases of another feature in the same list. This is done with the help of feature test program mapping.There are several features are to be considered as one can see on Applications[5],but the code that was worked to implement the frame work has taken into consideration only two features i.e., UART and I2C. As it is just replicating a part of code to take into consideration all the other features.These features are also referred as ***applications***.The code worked out was able to successfully detect the testcases of

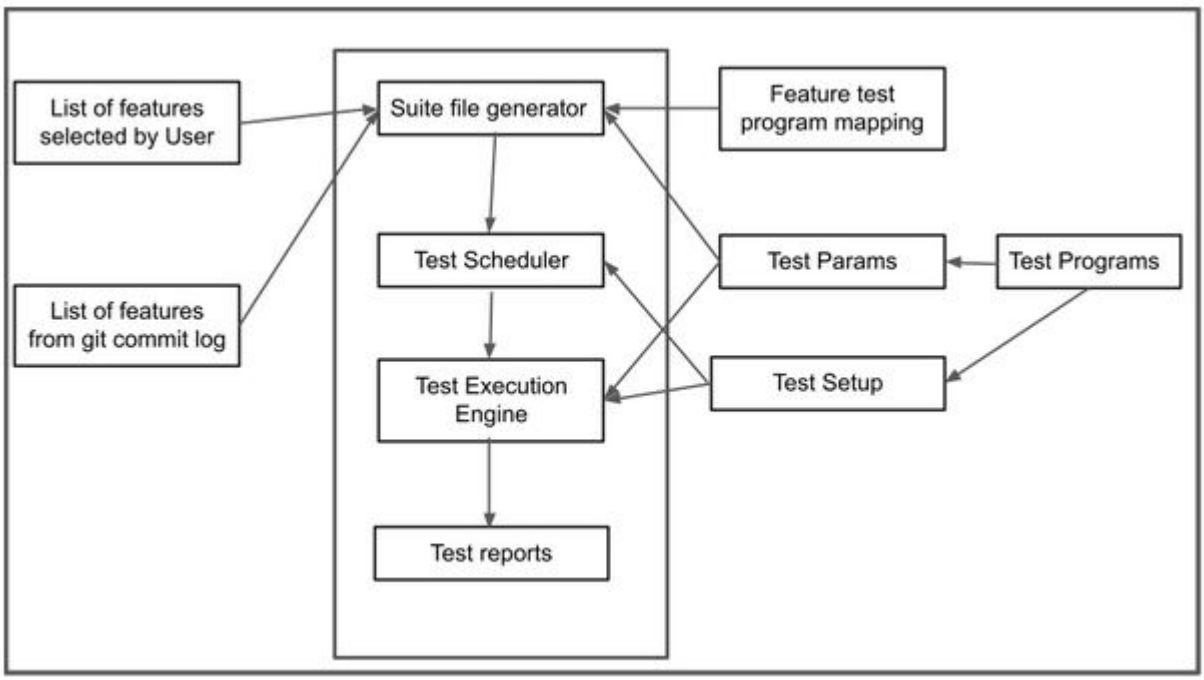


Figure 2.1: High level design for Test Automation frame work.

the entered (one or both) features and was able to extract the list consisting of testcases

2.1.2 Tests Parameters

After the testcase list is prepared, test parameters are required to complete the structure for generating suite file.Test parameters consists of RUNID, Input, Expected output.RUNID is either the time stamp at which the suite file is generated or entered manually by user.

1. **Input:** This list needs two elements. The first element is the path to the gdb debugger. The second element is command to start the gdb debugger in terminal.

2. **Expected output:** This is a list of arrays. Each array will contain the following three fields. The user can run the follows list of commands in their given sequence in the gdb debugger.

   (a) **Command:** The gdb command that need to be run. eg."break main"

   (b) **expOpWordList:** This is the list of key words that will be expected in the terminal if the command runs succesfully. Give a empty string in the list if no output is expected. eg:["Breakpoint","1"]

   (c) **FailWordList:** This is the list of key words that will be expected in the

terminal if the command does not execute. Give a empty string in the list if no output is expected. eg:[”Make”,”breakpoint”,”pending”]

```
"hello": [{
        "Input":["/home/ubuntu/shakti/shakti-sdk/tools/bin","riscv64-unknown-elf-gdb"],
        "ExpectedOutput": [
                {
                        "command": "set remotetimeout unlimited",
                        "expOpWordList": [""],
                        "FailWordList":[""]
                },{
                        "command": "file /home/ubuntu/shakti/
shakti-sdk/software/examples/uart_applns/hello/output/hello.shakti",
                        "expOpWordList": ["Reading","symbols" ],
                        "FailWordList":["No","symbol","table"]
                },{
                        "command": "target remote localhost:3333",
                        "expOpWordList": [ "Remote","debugging","localhost"],
                        "FailWordList":["Remote", "connection", "closed"]
                }, {
                        "command": "load",
                        "expOpWordList": [ "Loading","section","Start","address","Transfer", "rate"],
                        "FailWordList":["Load","failed"]
                }, {
                        "command": "break main",
                        "expOpWordList": ["Breakpoint", "1"],
                        "FailWordList":["No","symbol","table","loaded"]
                },{
                        "command": "continue",
                        "expOpWordList": ["Continuing","Breakpoint"],
                        "FailWordList":["program","not","run"]
                }

        ]
}]
```

Figure 2.2: Figure showing structure of Test parameters.

The input and Expected output are picked up from a separate text(.txt) file for each test case.Before creating this text file which contains this test parameters, executable (ELF) file (with .shakti as extension) for that particular testcase has to be generated.The steps for creating a ELF is explained in detail under section 5.3.3 in Shakti-soc-user-manual[3]

2.1.3 Generate suite file

Once the test case is prepared and test parameters are acquired the next step is to combine them into single in JSON format.The suite files should be in valid JSON format.Note that all the above extractions (test case list and test parameters) are in a dictionary object format. Using ***json.dump***, dictionary format is converting

```
[{
        "RUNID": "SHAKTHIAT1901",
        "testcaseList": ["hello"],
        "Parameters": [{
                "continueonfail": "false"
        }],
        "hello": [{
                "Input":["/home/ubuntu/shakti/shakti-sdk/tools/bin","riscv64-unknown-elf-gdb"],
                "ExpectedOutput": [
                        {
                                "command": "set remotetimeout unlimited",
                                "expOpWordList": [""],
                                "FailWordList":[""]
                        },{
                                "command": "file /home/ubuntu/shakti/
        shakti-sdk/software/examples/uart_applns/hello/output/hello.shakti",
                                "expOpWordList": ["Reading","symbols" ],
                                "FailWordList":["No","symbol","table"]
                        },{
                                "command": "target remote localhost:3333",
                                "expOpWordList": [ "Remote","debugging","localhost"],
                                "FailWordList":["Remote", "connection", "closed"]
                        }, {
                                "command": "load",
                                "expOpWordList": [ "Loading","section","Start","address",
                                                "Transfer", "rate"],
                                "FailWordList":["Load","failed"]
                        }, {
                                "command": "break main",
                                "expOpWordList": ["Breakpoint", "1"],
                                "FailWordList":["No","symbol","table","loaded"]
                        },{
                                "command": "continue",
                                "expOpWordList": ["Continuing","Breakpoint"],
                                "FailWordList":["program","not","run"]
                        }

                ]
        }]

}]
```

Figure 2.3: Structure of a suite file.

into a JSON format and written into a file. This is the ***suite file*** and the name of the file is it's RUNID. So in simple words based on the features selected for testing, suite file generator creates a file that has a list of test cases to be executed. Suite file is generated based on the applicable test cases for a given feature. Input and output values of each test case are provided in test case parameters. The suite file is named after it's RUNID. As shown in fig 2.3, a suite file has RUNID, testcase list, parameters and Input, output test parameters

- **RUNID :** Id of the process
- **TestcaseList :** The list of testcase classes given here will be executed
- **Parameters :**
  *continueonfail :*This takes only two values (true, false). Give true when you want the next subtestcase to be executed when a subtestcase fails. Give false when you dont want the next subtestcase to be executed when a subtestcase fails.
  eg:"false"

NOTE : This is a string "false".This is only to display on the output. It should not be confused with the logical false object type of JSON file i.e., *False* as this string "false" doesn't have any logical significance.

2.1.4 Execute tests

At present, Shakti SoC verification is done by running programs using openocd, gdb and console. Openocd is used to connect to debugger in the SoC, gdb connects to SoC using openocd to load and execute the programs. Output of these programs are directed to UART0 by default. Before executing the test cases, FPGA is programmed with the mcs mentioned in the suite file.
Using pyexpect libraries execution engine opens 3 terminals, one each for openocd, gdb and console for the given setup. Executes the command as provided in the test case and checks for the expected output. Based on the output string, execution engine marks the test case as passed or failed.
Initially the execution engine will run test cases that need one Artix 7100T, capability to work with multiple boards will be added in subsequent updates.

The worked out code for this Execution engine[6] can be described as follows:
This is engine of the Automated Testing framework. This is the class that needs to be run by the user in order to run the testcases stated in the user.suite file. The

Engine requires a suite file to create an instance of it. The Engine class has two functions. They are *Run Testcases* and *Generate Report*.

1. **Run Testcases :** This method is used to run the testcases stated in the testcase list of the user.suite file. It returns a result list which is list of lists containing lists of results of each testcase and the last two elements of this result list are time taken to run all the testcases and the run id.

2. **Generate Report:** This method takes the result list as input parameter. Checks whether each sub test of a testcase is passed or not. If each sub test of a testcase is passed, It implies that the testcase is passed. This method generates a *html* report.

Chapter 3

Work accomplished and Results

Given the background study from the previous two chapters, python code for generating desired suite file was developed. The logic was designed considering only two features i.e., UART and I2C applications.The code can be made applicable to other features and test cases by repeating certain conditions with those features as variables.
Python 3.6 was platform used for working out codes and *Jupyter notebook* was used as editor

3.1 Worked out python codes

Three python codes were developed.They can be accessed from suite file[7]

1. functions.py
2. suitegenerator.py
3. git suitegenerator.py

functions.py : This consists all the relevant logic as definitions which can to be imported.This was done in order to reduce the repeating sections with different variables in other two python codes. And it also becomes easier to debug.This file has to be imported in other two codes for proper execution and getting outputs.

suitegenerator.py : This code was developed keeping in mind the case where user gives the input manually.The input here is the string of features whose test-cases are to be executed.It also takes RUNID as another input which is to be entered manually.If the entered RUNID is of zero length string then it considers time-stamp as RUNID.Time-stamp is the exact point of time (including the calendar date) at which the code was executed. RUNID is specific for each and every execution so that whenever documentation is made, RUNID is taken as reference for that particular

execution.

Before execution of this code, it is important to create two text(.txt) files in a dictionary format.

- File that has relevant directory paths.
- File that has features and it's testcases.

Along with these two files, a folder consisting of text files is created.And this folder is to be created for all the features that are mentioned as Applications[5]. The text files inside this folders will have test parameters like Input and Expected output for that particular testcase of a particular feature.This testparameters are extracted and combined with RUNID,testcases list and other parameters as a list in a dictionary format.Then, finally this dictionary format is converted into a json format and written into new file named after it's RUNID. This is the desired suite file.

git suitegenerator.py : This is for generating suite file from *git log* command. In python, Unix commands are executed with the help of OS and subprocess modules. Using the same *git log* command is executed and the output is captured as a string. Then these string is parsed into a list to narrow down to a feature.Once the feature is identified, the procedure is same as **Suite file generator.py**. The only difference is sometimes we may have generate suite only for a single testcase depending on the output of *git log* command.Before narrowing down to feature or a testcase,the repeated features and overlapped testcases have to be removed in order to reduce the execution time.Otherwise same testcase may be have repeated appearance in final suite file. Code logic was designed accordingly.

3.2 Results

The code *Suite file generator.py* written was able to generate suite file which had testcases of both features UART and I2C.These features are entered manually.It a valid JSON format verified online at JSON lint[8]
Execution report of *hello.c* of UART feature from the generated suite file is shown here.

3.2.1 Output suite file

The output suite file has two sections.

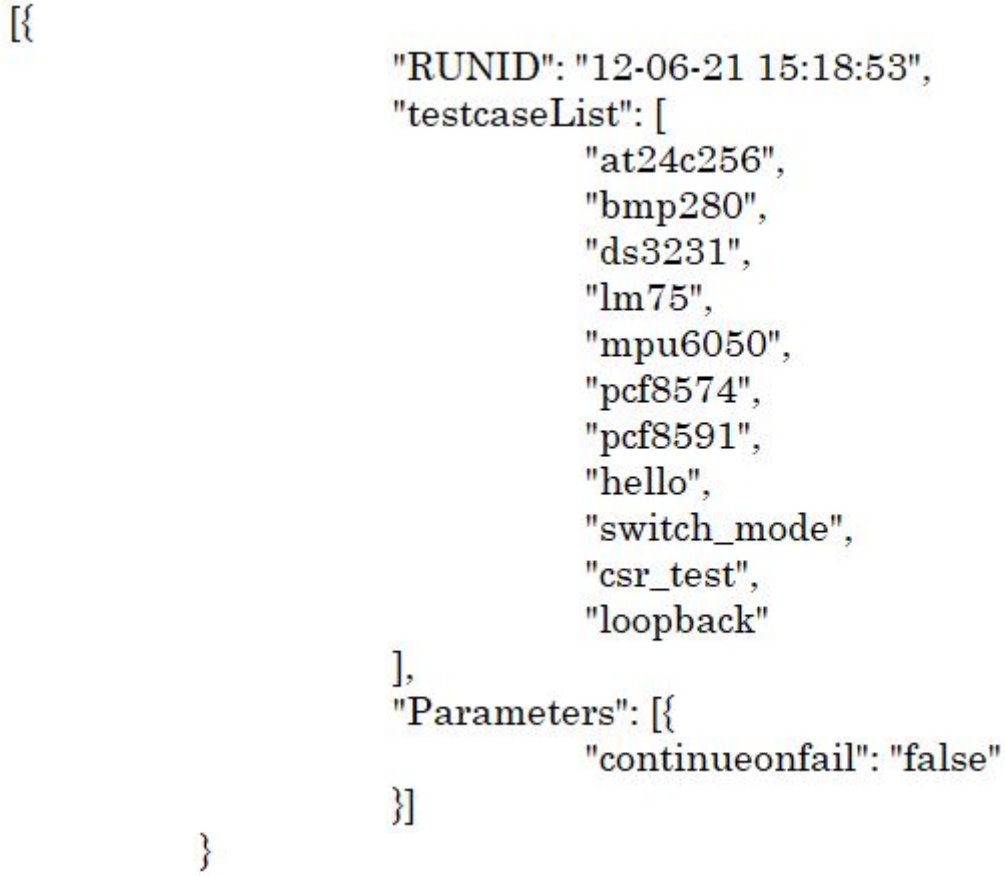


Figure 3.1: Section 1

This is the top section in the generated suite file. It consists of RUNID,testcase list and continueonfail parameters. The testcases list of *i2c* feature is [at24c256,bmp280, ds3231,lm75, mpu6050,pcf8574,pcf8591] whereas the testcase list of *uart* feature is [hello,switch mode,csr test,loopback]. So as expected for the output string of features "i2c applns,uart applns" the testcase list is the combined lists of both which is clearly the case here.

Also one can notice the RUNID is a time-stamp (which is a point of execution of the program) which is also the one of the objective of the suite file.As mentioned in chapter 2, RUNID can be entered manually and the same will be displayed.

```
"at24c256": {
                "Input": ["/home/ubuntu/shakti/shakti-sdk/tools/bin","riscv32-unknown-elf-gdb"],
                "ExpectedOutput": [{
                                "command": "set remotetimeout unlimited",
                                "expOpWordList": [""],
                                "FailWordList": [""]
                        },
                        {
                                "command": "file /home/ubuntu/shakti-sdk/software/examples/
                                        i2c_applns/at24c256/output/at24c256.shakti",
                                "expOpWordList": ["Reading","symbols"],
                                "FailWordList": ["No","symbol","table"]
                        },
                        {
                                "command": "target remote localhost:3333",
                                "expOpWordList": ["Remote","debugging","localhost"],
                                "FailWordList": ["Remote","connection","closed"]
                        },
                        {
                                "command": "break main",
                                "expOpWordList": ["Breakpoint","1"],
                                "FailWordList": ["No","symbol","table","loaded"]
                        },
                        {
                                "command": "load",
                                "expOpWordList": ["Loading","section","Start","address",
                                                        "Transfer","rate],
                                        "FailWordList": ["Load","failed"]
                        },
                        {
                                "command": "continue",
                                "expOpWordList": ["Continuing","Breakpoint","1"],
                                "FailWordList": ["program","not","run"]
                        }
                ]
        }
```

Figure 3.2: Section 2

This is the next section in the generated suite file. It has test parameters of a all the testcases listed in the testcase list in the previous section. For each testcase, the test parameter are in the format which is similar to what is shown above.
Using this suite file, we generate report using execution engine.

3.2.2 Report

Suite file: hello64
RUNID: SHAKTHIAT1901
Start Time: 2021-06-15_19:03:06

Executing test case: Hello World Creating setup
Setup: Done
loaded elf file

Term	Command	Status
gdb	continue	PASS
miniterm1		PASS

Time taken to execute testcase Hello World is: 5.26

number of testcases : 1
number of testcases Passed: 1
Time taken to execute the suite :5.26

End Time : 2021-06-15_19:03:11

This is the report for the *"hello"* testcase. As shown, the report puts out the RUNID,start time,End time,Time taken for executed testcase. minterm1 is related to display on FPGA. As shown in the report the testcase was passed.
The remaining 10 testcases report are similar.

3.2.3 Conclusion

Achieved desired result of generating a suite file for the given features (in the form of a string) and it was observed that all the testcases in the testcase list of the generated suite file were shown status *passed* in the execution report.

Bibliography

[1] "Development Environment and Tool chain installation," https://shakti.org.in/learn_with_shakti/devenv.html, accessed: 12-06-2021.

[2] "Setting up the SHAKTI-SDK," https://shakti.org.in/learn_with_shakti/c-using-shakti-sdk.html, accessed: 12-06-2021.

[3] "Shakti-soc user-manual," https://shakti.org.in/docs/shakti-soc-user-manual.pdf, accessed: 12-06-2021.

[4] "Introducing json," https://www.json.org/json-en.html, accessed: 12-06-2021.

[5] "Applications," https://gitlab.com/shaktiproject/software/shakti-sdk/-/tree/master/software/examples, accessed: 12-06-2021.

[6] "Execution engine," https://gitlab.com/shaktiproject/software/softwaretestautomation/-/blob/master/engine.py, accessed: 12-06-2021.

[7] "suite file generator," https://gitlab.com/shakti-software/testautomation/-/tree/main/suitegenerator, accessed: 15-06-2021.

[8] "Json lint," https://jsonlint.com/, accessed: 16-06-2021.