Developer Mode User Guide

Introduction

This document serves as a guide attached to the Regression Failure Triage using ECTB Parameter Clustering - AMIQ Blog Article. The intention behind it is to provide an insight into how to correctly set up and use the sandbox environment used to determine the most suitable clustering methodology. The following examples are for the project case study block, Matrix Determinant, but can be implemented into any testbench set up with the project's requirements:

1. Development of the verification environment using ECTB architecture and principles.
2. Implementing a high-level of abstraction, enabling the verification environment to be governed by a single general sequence. For Matrix Determinant, there are two types of general sequences: 'amiq_md_base_seq' and 'amiq_md_special_seq', both of which completely control all traffic sent to the block.
3. Transparency of values used for all control parameters of a test scenario, by displaying these in the associated run log, regardless of whether they were defined in the plusargs file. This is implemented in the ECTB version attached to the project repo.
4. Dependencies on the following non-standard Python packages: NumPy version 1.21.5, pandas version 1.3.5, Plotly version 5.18.0, Dash version 2.16.1, scikit-learn version 1.3.2. All of these are documented in the README.txt file of the project repo.

Developer Mode Scripts

The scripts used for the sandbox environment are located in the amiq23_md submodule at the path amiq_pclust/amiq23_md/scripts/development, and are:

- **bugs_generator.py** -> This script randomly generates bug to error associations, based on the Excel configuration provided (i. e. template **bugs_generator_config.xlsx)**. The control knobs for the script are:
    - config -> (string) Path to the configuration Excel file.
    - nof_max_bugs_in_error -> (int) the maximum number of bugs associated with the same error. Default 2.
    - seed -> (int) generator seed. Default 0.
    - auto_parse -> (bool) if true, the ECTB regression parameter database is automatically obtained from the generated tests and error model. Running regressions is not necessary anymore using this knob. Default 1.
    - auto_regression -> (bool) if true, all regression related files (synthetic SVAs, test plusargs, and regression VRSF) are generated. Used for replicating the user flow. Default 0.

Example:
$ *python3 bugs_generator.py --config bugs_generator_config.xlsx --nof_max_bugs_in_error 2 --seed 2 --auto_parse 1 --auto_regression 0*

- **clustering_visualize_sandbox.py** -> This script works only if bugs_generator.py was run with auto_parse true. It is based on the same principle as the project script itself (regression_analysis.py), but instead directly sources the ECTB regression parameter database from **bugs_generator.py** (the **sandbox_matrix.xlsx** Excel file). It also adds a way to interactively display the clustering evaluation metric in the GUI.

Setup

The bugs_generator_config.xlsx template Excel file that comes with the project is going to be the configuration for the bugs_generator.py script. This has three sheets:

- **Flags**: will be populated with the parameter names and all of their possible values (each parameter on its own line). Note that if auto_regression is true, the parameter values must be named "*val_Y*" (where Y is a positive integer).

	A	B	C	D	E	F	G
1	flag_name						flag_values
2	param_A	val_0	val_1	val_2			
3	param_B	val_0	val_1	val_2			
4	param_C	val_0	val_1	val_2	val_3		
5	param_D	val_0	val_1	val_2			
6	param_E	val_0	val_1	val_2	val_3	val_4	
7	param_F	val_0	val_1	val_2	val_3		

Figure 1. Example of populating the “Flags” sheet in bugs_generator_config.xlsx.

- **Sequences:** will be populated with the sequence names, the number of runs to be randomly generated for each sequence, the sequence type (just as in SystemVerliog), and the parameters that are used by the sequence (each sequence on a line).

	A	B	C	D	E	F	G	H	I	J
1	seq_names	nof_runs_per_seq	seq_types						seq_flags	
2	seq_0	250	amiq_md_base_seq	param_A	param_B	param_C	param_D			
3	seq_1	250	amiq_md_special_seq	param_A	param_B	param_E	param_F			

Figure 2. Example of populating the “Sequences” sheet in bugs_generator_config.xlsx.

- **Bugs:** will be populated with the desired amount of bugs to be injected into the testbench, by setting a bug name, the number of flags that associate with the bug (chosen randomly), and optionally the sequence type through which it can be traced (if not set, it is chosen randomly). Each bug is defined on a separate line.

	A	B	C
1	bug_ name	nof_flags_in_bug	bug_seq_type
2	bug_0	2	
3	bug_1	3	
4	bug_2	2	
5	bug_3	3	

Figure 3. Example of populating the “Bugs” sheet in bugs_generator_config.xlsx.

```
the assertions matrix:
     bug   error param_A param_B param_C param_D param_E param_F    seq
0  bug_3  error0   val_1     any     N/A     N/A   val_3   val_2  seq_1
1  bug_1  error0   val_2     any     N/A     N/A   val_0   val_1  seq_1
2  bug_0  error1   val_1     any     any   val_0     N/A     N/A  seq_0
3  bug_2  error2     any     any     N/A     N/A   val_4   val_2  seq_1
```

Figure 4. Example of generated bugs and associated errors.

In this case, **N/A** means that the sequence type through which the bug can be detected does not utilize the corresponding parameter, while **any** means that the parameter is used by the sequence type, but is not associated with the bug (meaning it can take any value without influencing).

! Important Note: If the auto_parse knob for the bugs_generator.py script is set to true, the parameter registering and sequence linkage steps below can be skipped. In the general sequences that are aimed to be used with the sandbox environment, the abstract verification parameters have to be registered as strings. The developer has to push all values registered with ECTB for the verification parameters into a queue of strings called 'flags', which can be done by overriding the ECTB 'register_all_vars()' function, as shown in Figure 5:

```
Unset
virtual function void register_all_vars();
       super.register_all_vars();
       reset_pkt_nr_constraints = string_reg("param_A", "val_0");
       flags.push_back(reset_pkt_nr_constraints);
       matrix_pkt_nr_constraints = string_reg("param_B","val_0");
       flags.push_back(matrix_pkt_nr_constraints);
       exact_value_constraints = string_reg("param_E","val_4");
       flags.push_back(exact_value_constraints);
       delay_pattern_constraints = string_reg("param_F","val_3");
       flags.push_back(delay_pattern_constraints);

endfunction : register_all_vars
```

Figure 5. Registering parameters and populating the queue of flags

Also, in the top-level sequence's body, after all variable declarations, the synthetic SVAs have to be included, by adding the macro **`include "<ECTB_seq_name>_svas.svh"**. For Matrix Determinant, the general sequence named seq_0 of type amiq_md_base_seq should be included at the top of the body (after variable declarations) **`include "seq_0_svas.svh"**, as indicated in Figure 6**.**

```
Unset
virtual task body();
       int num_resets, num_input;
       `include "seq_0_svas.svh"
```

Figure 6. Synthetic SVAs to sequence type linkage

Examples of Developer Flow

A. Auto_regression true

1. Run the bugs_generator.py script.

   $ *python3 bugs_generator.py --config bugs_generator_config.xlsx --nof_max_bugs_in_error 2 --seed 2 --auto_parse 0 --auto_regression 1*

2. Synthetic SVAs, testcase plusargs files, and ECTB regression VRSF are generated (as shown in Figures 7, 8, and 9). Run regression as normal.

```
1  bit[4 - 1 : 0] flag_codes [4]= '{0, 0, 0, 0};
2
3  for (int i = 0 ; i < 4; i++) begin
4      int j;
5      $sscanf(flags[i], "val%d", j);
6      flag_codes[i][j] = 1'b1;
7  end
8
9  error2: assert((flag_codes[0] != 3'b100 || flag_codes[1] != 3'b001 || flag_codes[3] != 3'b001)) else
10     $error("error2");
11
12 error3: assert((flag_codes[0] != 3'b001 || flag_codes[2] != 4'b0001 || flag_codes[3] != 3'b100)) else
13     $error("error3");
14
15 error1: assert((flag_codes[1] != 3'b001 || flag_codes[3] != 3'b001)) else
16     $error("error1");
```

Figure 7. Example of generated synthetic SVAs file.

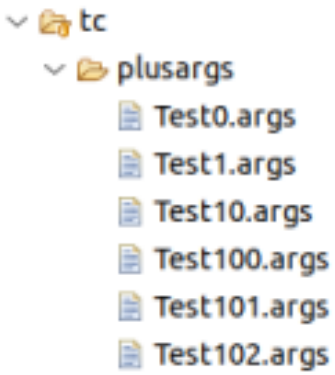

Figure 8. Example of generated testcase plusargs files.

```
session amiq_pclust_reg {
    top_dir: "$ENV(PROJ_HOME)/sim/regression";
    output_mode: terminal;
    pre_session_script: "$ENV(PROJ_HOME)/sim/compile.sh $DIR(session)";
    queuing_policy: round_robin;
};

group amiq23_md_tests {
    scan_script: "vm_scan.pl shell.flt ius.flt ovm_sv_lib.flt vm.flt";
    run_script: "xrun +UVM_TESTNAME=amiq_md_base_test -svseed $ATTR(seed) $ATTR(top_files) -f $ENV(PROJ_HOME)/sim/sim.options -licqueue";
    timeout: 120;

    test amiq_pclust_test0 {
        top_files: "-f $ENV(PROJ_HOME)/tb/tc/plusargs/Test0.args";
        seed: random;
        count: 1;
    };
    test amiq_pclust_test1 {
        top_files: "-f $ENV(PROJ_HOME)/tb/tc/plusargs/Test1.args";
        seed: random;
        count: 1;
    };
    test amiq_pclust_test2 {
        top_files: "-f $ENV(PROJ_HOME)/tb/tc/plusargs/Test2.args";
        seed: random;
        count: 1;
    };
```

Figure 9. Example of generated regression VRSF file

3. Apply **regression_analysis.py** as normal.
   $ python3 regression_analysis.py –path <path/to/reg_dir> [-u] (includes passed runs) [-g] (generates regression database in csv format)

B. Auto_parse true

1. Run the bugs_generator.py script.

   $ *python3 bugs_generator.py --config bugs_generator_config.xlsx --nof_max_bugs_in_error 2 --seed 2 --auto_parse 1 --auto_regression 0*

2. Run the **clustering_visualize_sandbox.py** script.

3. The evaluation metric is displayed interactively in the Dash GUI, indicated by Figure 10.

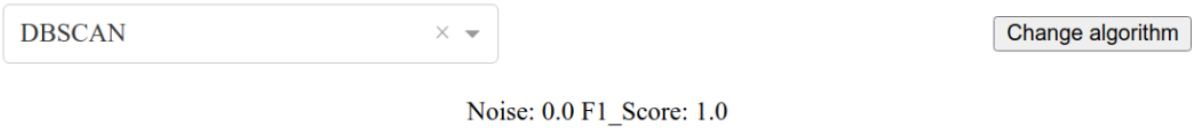

Figure 10. The evaluation metric displayed in the GUI