2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Automating the formal verification sign-off flow of configurable digital IP's

Giovanni Auditore (giovanni.auditore@st.com), Giuseppe Falconeri (giuseppe.falconeri@st.com)
STMicroelectronics
Stradale Primosole 50,
95121 Catania, Italy

***Abstract* – In this paper we propose a sign-off criteria for configurable digital IP verified using formal methods, which is based on a configuration metrics definition and collection. Once the target metrics is declared in a verification document, this paper propose a method and flow to automate the verification closure towards the defined sign-off target.**
The proposed solution overcomes the limitation of exhaustive configuration verification whenever this will lead to unacceptable run time. It also adds verification robustness by mean of randomization. A case of study shows how to automate and optimize the proposed flow making use of one of the verification automation platforms currently available on the market.

I. INTRODUCTION

The increasing complexity of HW systems requires nowadays maximizing reuse and automation. To make this feasible the configuration complexity of Digital IP's is also increasing. The verification of such digital IP's needs to cope with this trend providing configurable testbenches that are able to handle any of the possible HW implementations. The sign-off of such configurable IP's requires that a minimum subset of configurations have been tested. While this process is quite well defined in a dynamic verification sign-off flow, it is not the same in a formal static sign-off flow.

Configurability in formal testbench may apply to both property instantiation and property body implementation. A testbench has therefore different properties sets for each selected configuration and may easily lead to successful runs for some of the verified configurations and failures or complexity issues for some others. Any issue found may be due to both RTL and testbench coding errors. It is key to early identify and address all these issues in order to optimize verification re-use and avoid delays when going to verify any new configuration of the IP.

When the size of the total space of possible configurations is small, an exhaustive verification of all of them is often applied, by executing the formal proof of each configuration. In such cases automation is applied to formal runs in order to reduce the manual effort of re-running. EDA verification automation platforms not only allow to handle multi configurations execution, but also provides a way to collect and combine, whenever is needed, coverage metrics from the various runs. Collected results may be linked to a verification plan, where the achieved results can be compared with what has been planned and agreed as valid sign-off criteria by the verification project stakeholders.

Unfortunately, very few IP's nowadays allow to run exhaustive formal verification. The consequence is that the verification is often limited to what strictly required by the integrated version in the current SoC or at most extended to a few configurations agreed between design and verification and which may result far from what will be required for next products embedding the same IP.

The intent of this paper is to overcome this limitation by mean of automation of HW configurations generation, collection and verification for all those digital IP's which are highly configurable. A parametric verification plan defines not only the usual coverage metrics to be achieved, but also the configuration coverage requirements. Therefore, the IP can be signed-off if and only if a subset of all possible configurations achieving the agreed configuration metrics target has been successfully executed.

II. CONFIGURATION SPACE

In the context of highly parametric designs, having many parameters is a great advantage because the design can be reused in different contexts, but it will require a very high verification effort, since the design space can become easily huge. A high design configurability has also an impact in terms of execution run time and disk space consumption, because the same verification flow must be iterated many times.

The main questions related to the verification of a highly parametric design are the following:

- Is it reasonable to verify millions of combinations or is it possible to choose a good subset of this?
- Which should be the 'best' subset of combinations?
- When is it possible to consider a configurable design fully verified?

Ideally, the verification flow for a parametric design should be applied for all the possible configurations of the design as described in the following loop:

```
compute total_space;
foreach configuration C in total_space
{
  do complete verification flow for configuration C;
}
analyze verification results;
```

Figure 1. Ideal verification flow for a parametric design

The ideal flow is very often not feasible because of a limited amount of time and/or resources available. Therefore, it is necessary to find a reduced space on which we can apply the verification flow. Moreover, an accurate analysis of the quality of the reduced list selected becomes fundamental to meet the final quality metrics requirements of the whole verification activity. The real flow now becomes like this:

```
repeat
{
  select reduced_space;
  analyze reduced_space;
} until reduced_space meets analysis target;

foreach configuration C in reduced_space
{
  do complete verification flow for configuration C;
}
analyze verification results;
```

Figure 2. Real verification flow for a parametric design

The difference between the ideal flow and the real flow is in the selection and analysis of the reduced space. The first step can be iterated as many times as necessary until an acceptable reduced space is identified.

Covering multiple combinations of configurable parameters is somehow very similar to cover multiple combinations of input stimuli (functional coverage items definition). Nevertheless, the selection process of the valid combination of parameters has been so far based on an arbitrary direct choice approach, without any measure on the quality of this choice. The outcome was to have a very different practical handling of the two very similar processes.

We decided to fully extend the coverage approach also to the definition of the configurations of the reduced space. A pseudo-random generation has been used to generate valid configurations sets. Coverage items on individual parameters and on their correlation are implemented. The collected metrics can be used to make sure that the analysis target are met by the reduced configuration space.

The definition of the coverage metrics and coverage targets used to identify the configurations subset is a matter of individual judgment based on the knowledge of the design. Depending on the nature of the design, the cover items can be a simple cross between the parameters or a function of the parameters values.

For example, assuming the parameters control the bus width of two interfaces, the requested configuration metrics can be:

- full cross coverage of the two parameter values
- only a subset of the whole space of combinations
- all possible results values of a defined function (ex. equality: equal, greater than, less than)

To better clarify the concept, let us make a generic example using a verification tool like *Specman*, which has both the capability of constrained random generation and coverage collection.
The small example of the Figure 3 describes an IP with three parameters having 128 different possible combinations. This example of code generates a given number (CFG_NB) of random configurations and it is able to produce a functional coverage model where all the parameters (and relevant cross) are covered.

Analyzing the functional coverage, it's easy to verify that 4 configurations are enough to consider it exhaustive (because of the cross coverage ParamX, ParamY), while modifying the cross coverage item in cross ParamX, ParamY, ParamZ, the minimum set of random configuration satisfying the functional coverage is 12. The choice of the cover item, very specific to the design under test, is then fundamental for the selection of the configurations of the reduced space.

```
struct cfg_s is {
  ParamX: bool;
  ParamY: bool;
  ParamZ: uint;
    keep ParamZ in [0..31];

  event cfg_param_cover_e;

  cover cfg_param_cover_e {
     item ParamX;
     item ParamY;
     item ParamZ using ranges ={
       range([0] ,       "Min Value");
       range([1..30] ,   "Mid values");
       range([31] ,      "Max Value");
     };

     cross ParamX, ParamY using name="Basic_configs"; //4 combinations
     cross ParamX, ParamY, ParamZ  using name="Extended_configs"; //12 combinations
  };
};

extend sys {
  cfg_list: list of cfg_s;
    keep cfg_list==CFG_NB;
  post_generate() is also {
    for each (cfg) in cfg_list {
      emit cfg.cfg_param_cover_e;
    };
  };
};
```

Figure 3. Small IP with 3 parameters

The output of this *Specman* run is also a set of cfg_<x>.yml files containing the selected configuration parameter values:

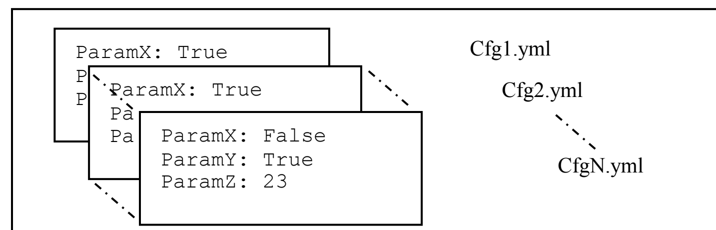

Figure 4. Generated configuration files

These files can be used as inputs to major scripting languages which provide built-in libraries for parsing the YAML format. The implemented script can then be used to execute the verification formal proof.

III. CONFIGURABILITY IN FORMAL VERIFICATION

Formal testbenches to address the verification of configurable Digital IPs are often inheriting the same parametrized structure of the HW which has to be verified.

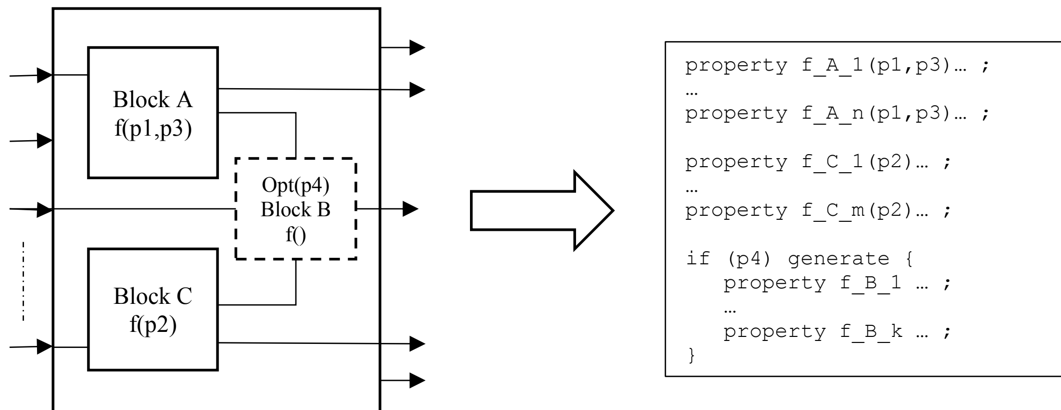

Figure 5. Configurable RTL vs formal testbench configuration

Somehow similar to the configurability in the RTL coding, the parameters are used to control:

- the property set structure, by creating a direct dependence between the parameter value and the existence and/or the kind of one or more properties
- the property functionality, by the use of one or more parameters in the body of the property itself

Since properties represent the allowed stimuli when configured as assumptions, the implemented checkers when configured as assertions and the functional scenarios when configured as cover, the achieved verification target becomes highly dependent on the selected parameters configuration set.

It is recommended to track the sign-off target intent as a function of the parameter values combination, but this becomes often too heavy to be handled in a reasonable time when the total number of combinations is high and the dependency is widely spread among the properties code.

Ideally, the verification intent should declare for each set of the parameters value which property applies, how it applies (assume, assert, cover) and the coverage target for all those parameters which control the body of the property itself. This intent should then be mapped to the testbench actual implementation in order to be sure that no unwanted assumptions are present or wanted checkers are missed.

As the intent of this paper is to reach an overall sign-off target and not a per-configuration sign-off target the proposed mapping is approaching this problem from a global point of view.

IV. AUTOMATED FLOW

Verification automation platforms are nowadays widely adopted to manage multiple job execution and results collection and analysis. With the target of reducing the manual effort on handling multi-configuration verification, it is natural to make use of such tools also to integrate the configurations generation and reduction step. Many different algorithms of reduction of the verification space can be applied. Such algorithms are out of the scope of this document and are not described in here. Whatever will be the selected reduction algorithm it is always required to perform a coverage analysis step to make sure the reduced space is optimized with respect to the planned coverage requirements.

Two approaches can be followed:

- full metrics driven
- configuration metrics driven

A full metric approach requires the generation and full execution of the various configurations runs. Metric analysis will be more reliable since it already includes the whole space of the planned metrics. Unfortunately, such approach

is very expensive from a run time point of view, since the initial size of the pseudo-random configuration may be quite big in order to reach the wanted coverage target.

A configuration metrics approach requires only the generation of the configuration, but does not require the execution step on a first iteration. The metrics of the generated configuration space can be collected and analyzed with respect to the configuration metrics target in relatively smaller time compared to the full verification execution. Ranking techniques can be applied to select the subset of configurations which maximize the coverage target. Execution is performed only in a second iteration step and it is normally faster with respect to the full metrics approach, since the total number of selected configurations has been already optimized. Full coverage metrics of executed runs require to be collected during the execution step and results may be less effective than the ones obtained using a more extensive number of runs. Therefore, the initial speed-up in run-time can be degraded by the overall coverage metrics effectiveness achieved.

The selection among the two flow is often not so trivial, but some rule of thumb can be given. Design where the configuration is mainly controlling code instantiation (parameters mainly used in HDL “if/for generate” blocks) are more likely to converge faster using a configuration driven approach. Design where a configuration is mainly controlling code functionality (parameters value used in code implementation) are more likely to converge faster using a full metrics driven approach.

Both flow can be automated using a verification automation platform although the approach is slightly different. A graphic representation of both implementations is showed in the following figures.

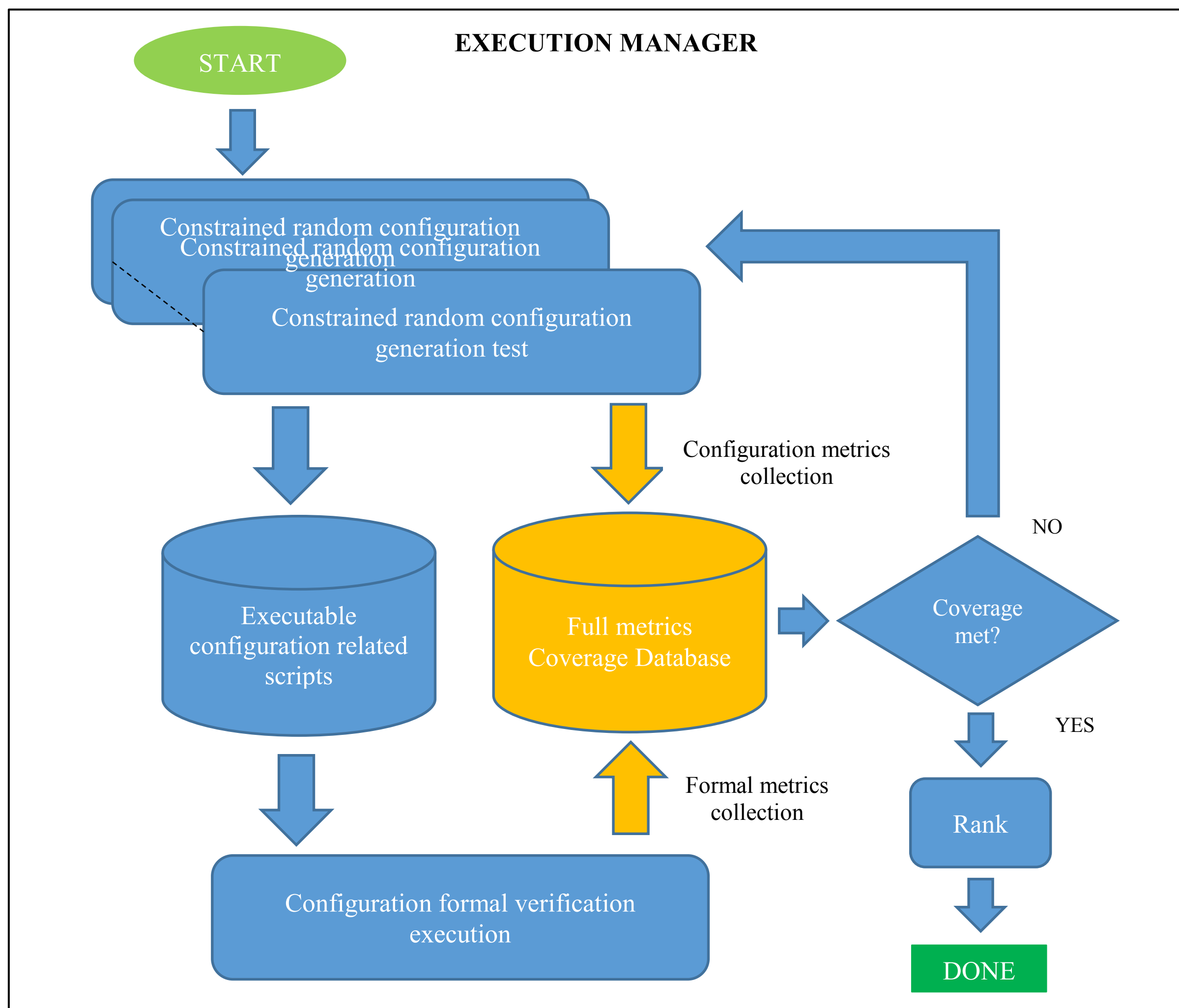

Figure 6 Regression automated flow full metric driven

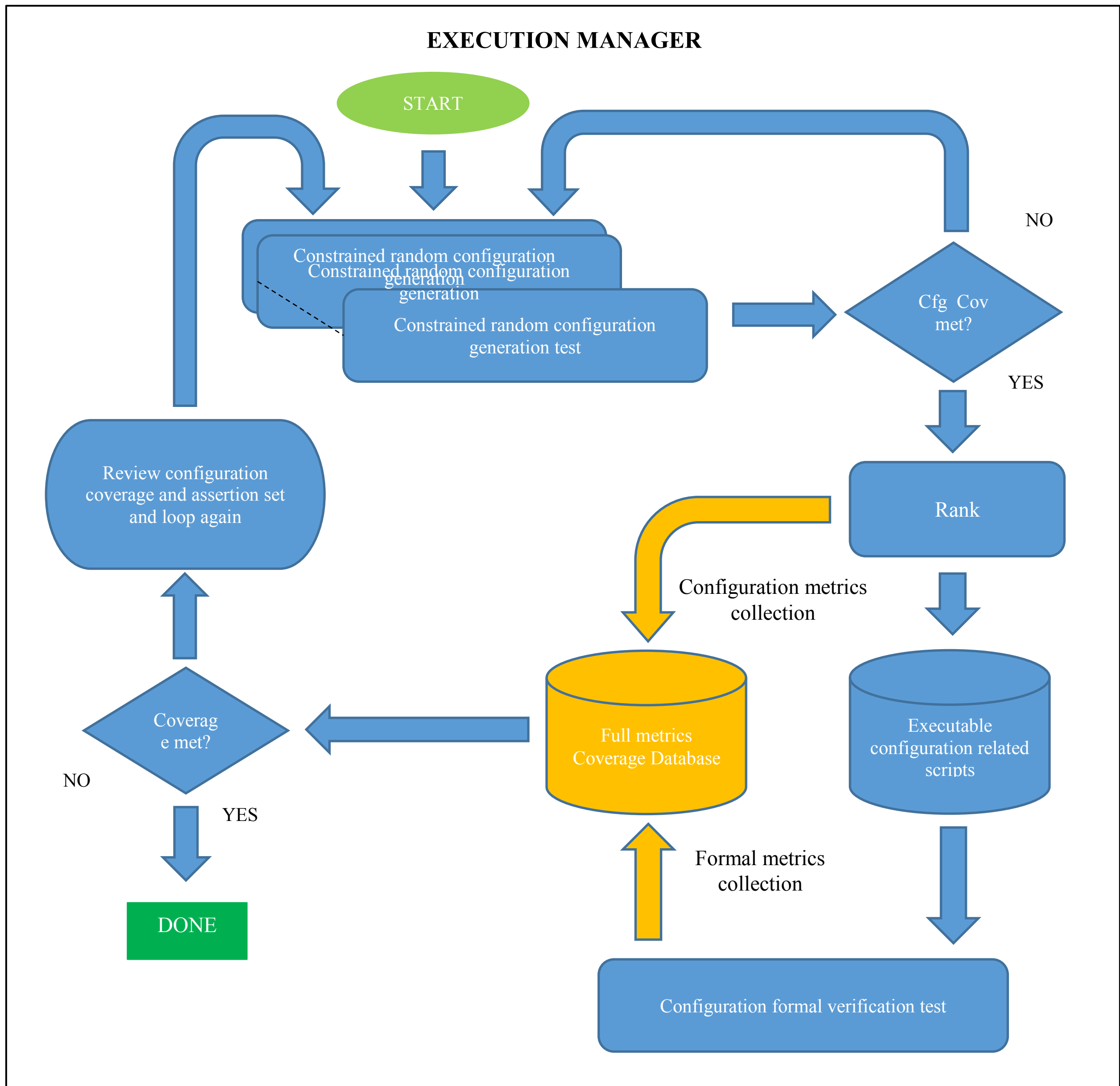

Figure 7. Regression automated flow configuration metric driven

To make the sign-off coverage target easier to be understood and reviewed, also by non-verification experts, it is recommended to gather together the required metrics in a verification document. Such practice can be extended to configurable IP's adding configuration requirements directly into a global verification plan which can be mapped to the overall achieved results.

V. PLANNING

For what concerns the verification plan, a configurable IP design is normally associated to a parametric verification plan. Such plan is valid for a specific HW implementation in terms of functionality to be addressed and related checkers. Some functionality may exist or not depending on the configuration value, while in some other case the function itself depends on the value of one or more parameters.

An alternative approach will be to have a global verification plan covering all possible functionality.

Users may be tempted to reduce the planning to the simple coverage of individual functionality, merging the multi-configuration runs results into a union of functional coverage items. Once a function has been exercised in any of the executed configurations, it is marked as covered. Such simplification can be dangerous because it is neglecting the impact of the changes in the design microarchitecture that may influence the input scenarios to the given function or the parametric dependence of function use modes.

A more robust approach would be to guarantee that all functionalities are exercised in all possible ways in which these can be configured and/or combined together. The coverage implementation for such kind of plan should expect not only the given function to be exercised in one configuration, but to be exercised in all possible HW context (all possible microarchitectures where the function is implemented) and in all function configuration dependent modes.

Achieved results can be mapped to the planned items accordingly to the selected approach and assure that each functionality is exercised as required.

VI. REAL CASE STUDY

To prove the effectiveness of the proposed flows we apply an implementation of both to an actual configurable HW design, whose parameters enable, control and instantiate several different functionalities.

The implementation relies on Cadence Verification Automation Platform.

A. Design Under Test (DUT)

The DUT is an event controller able to handle up to 96 events and to generate Interrupts and Events to one or more CPUs. It has 6 parameters allowing a great flexibility in the number and kind of events supported. The TABLE I shows the DUT configurability with a brief explanation of the various parameters.

TABLE I
DUT PARAMETERS DESCRIPTION

Parameter Name	Parameter Description	Parameter Range
nb_of_events	Number of events	16 to 96
nb_of_cpu	Number of CPU interrupt controller	1 to 4
trig_cfg	Type of event, a selection among two different kind of events	96 bits mask (only the lower Number of Events bits are valid)
cpu_rxev_en	Optional propagation of enabled events to dedicated CPU outputs	4 bits mask (only the lower Number of CPU bits are valid)
rxev_cfg	Mask bit vector for events enabled to CPU propagation	96 bits mask (only the lower Number of Events bits are valid)
tz_cfg	Mask bit vector for events which implement AHB5 TrustZone security protection	96 bits mask (only the lower Number of Events bits are valid)

The total configuration space overcomes billions of configurations; therefore, a proper selection of meaningful set is mandatory.

B. Formal Testbench

The formal testbench makes use of the same parameters to control the properties instantiation and execution and it is able to verify any given configuration value. An example of property code is shown in the Figure 8 below.

```
generate
   begin: event_index_and_cpu_index
      ///////////////////
      // FOR EACH EVENT
      ///////////////////
      for (evt_idx=0; evt_idx<=nb_of_events;evt_idx=evt_idx+1) begin : gen_event_idx
         ////////////////
         // FOR EACH CPU
         ////////////////
         for (cpu_idx=0; cpu_idx<=nb_of_cpu;cpu_idx=cpu_idx+1) begin : gen_cpu_conf_idx
            ///////////////////////////////////////
            //CONFIGURABLE EVENT at index evt_idx
            ///////////////////////////////////////
            if (trig_cfg[evt_idx] == 1) begin: gen_configurable

               ////////////////////////////////////////////////
               //CONFIGURABLE EVENT with event logic DISABLED
               ////////////////////////////////////////////////
               if (rxev_cfg[evt_idx] == 0 || cpu_rxev_en[cpu_idx] == 0) begin: gen_evt_dis
                  gen_sys_wakeup_idx_wkup_cfg_rxev_dis_chk: assert property (…);
               end
               ////////////////////////////////////////////////
               //CONFIGURABLE EVENT with event logic ENABLED
               ////////////////////////////////////////////////
               if (rxev_cfg[evt_idx] == 1 && cpu_rxev_en[cpu_idx] == 1) begin: gen_evt_enb
                  gen_sys_wakeup_idx_wkup_cfg_rxev_enb_chk: assert property (…);
               end
               …
            end
         end
      end
   end
endgenerate
```

Figure 8.Code snippet of configurable properties

Each property is generated as many times as needed according to the parameter settings. The functionality addressed by each property is also controlled by the selected configuration.

C. *Configuration Generation*

Since the various configurations are randomly generated, their generation has been done through *Specman*, exploiting its capability of constrained random generation. Parameters can have some constraints and it is easy to code them using a verification language like '*e'*. A snippet of the actual code used for the configuration generation can be found in the *Appendix* (Figure 12).

In case of the full metrics driven flow a *Specman 'e'* file is loaded and a large set of configurations are generated by a top visf file under vManager.

Since a unique test generating several configurations cannot be ranked, in the configuration metric driven flow each configuration comes from a separate "test" command under Specview: each one will have its own coverage collection to feed to a ranking algorithm (Cadence IMC). A bash script is used to manage this step of the flow, as shown in the *Appendix* (Figure 13).

The 'e' file generates all the configuration files Conf_<ID>.yml containing the parameters values that will be used in the formal runs. An example of such ".yml" file is shown in Figure 9 .

```
#-------------------------------------------------------#
#    Configuration : rnd_cfg_0
#-------------------------------------------------------#

nb_of_events : 16
nb_of_cpu    : 3
cpu_rxev_en  : 4'h7
trig_cfg     : 96'h013a4
rxev_cfg     : 96'h11b9a
tz_cfg       : 96'h1fbf3
priv_cfg     : 96'h118c5
```

Figure 9. Example of a generate yml file for one configuration

D. Configuration Coverage Collection

Coverage metrics are also be implemented in the *Specman* '*e*' file. Simple cover items collecting parameters values can be implemented as well as very complex parameters cross-coverage dependences. A collection event is emitted after the generation phase to track the achieved coverage of each configuration parameters set. A code snippet for the 'e' file implementation can be found in *Appendix* (Figure 14).

E. Configuration vsif generation

In order to execute the formal verification of all the selected configurations, the most convenient way is to create a unique *vmanager* session where all the configurations vsif (verification session input file) are imported. The various vsif files are also generated by the '*e*' code during the post_generate() execution, just after the generation of the yml files. Example of the generated vsif code can be found in *Appendix* (Figure 15).

F. Formal verification execution

The execution step is common to both flows. It is run under the vManager platform executing the JasperGold proof runs stated int the top vsif file generated by previous steps.

The full metric driven flow has been exercised with 30 random configurations and it satisfies the target of 95% for configuration coverage and 100% for functional metrics. The actual configuration coverage metrics reached was 98.36% with the cumulative runtime being around 180 hours (an average of 6 hours for each run).

Detailed combined coverage results are loaded and visible in the *vmanager* GUI, and the analysis of coverage annotated to the vPlan metrics are displayed in Figure 10.

vPlan Hierarchy — V2_2_VMETRICS

Name	Combined Average Grade	Combined Covered
(no filter)	(no filter)	(no filter)
V2_2_VMETRICS	99.98% *	60188 / 60206 (99.97%)
1 Functional requirements verification plan	99.98% *	60188 / 60206 (99.97%)
1.1 HW Generics	99.78% *	1080 / 1098 (98.36%)
1.1.1 HW Generics cover	99.57% *	1078 / 1096 (98.36%)
1.1.1.1 nb_of_event	100% *	5 / 5 (100%)
1.1.1.2 nb_of_cpu	100%	4 / 4 (100%)
1.1.1.3 cpu_rxev_en	100%	10 / 10 (100%)
1.1.1.4 trg_cfg	100%	1 / 1 (100%)
1.1.1.5 rxev_cfg	100%	2 / 2 (100%)
1.1.1.6 tz_cfg	100%	2 / 2 (100%)
1.1.1.7 priv_cfg	100%	2 / 2 (100%)
1.1.1.8 nb_ioports	100% *	5 / 5 (100%)
1.1.1.9 Cross_dependency_trig_rxev	98.68%	345 / 350 (98.57%)
1.1.1.10 Cross_dependency_contiguous_trig	98.68%	330 / 335 (98.51%)
1.1.1.11 Cross_dependency_contiguous_rxev	97.89%	372 / 380 (97.89%)
1.1.2 HW Generics checks	100%	2 / 2 (100%)
1.2 registers access	100% *	45294 / 45294 (100%)
1.3 Clocking strategy	100%	21 / 21 (100%)
1.4 Input events trigger	100%	528 / 528 (100%)
1.5 IOPORT inputs selection	100%	4128 / 4128 (100%)
1.6 System wakeup on inputs events	100%	1705 / 1705 (100%)
1.7 CPUn wakeup on inputs events	100%	3481 / 3481 (100%)
1.8 Configurable events output interrupt to all CPU's	100%	672 / 672 (100%)
1.9 CPUn output event on inputs events	100%	3279 / 3279 (100%)

Figure 10. Full metric driven flow vPlan coverage results

The coverage metric driven flow has first identified a subset of 22 ranked configurations sufficient to overcome the 95% coverage threshold, in about 30 sec. The actual configuration coverage metrics reached was 99.36% with the cumulative runtime being around 125 hours.

Detailed combined coverage results are loaded and visible in the *vmanager* GUI, and the analysis of coverage annotated to the vPlan metrics are displayed in **Figure 11**.

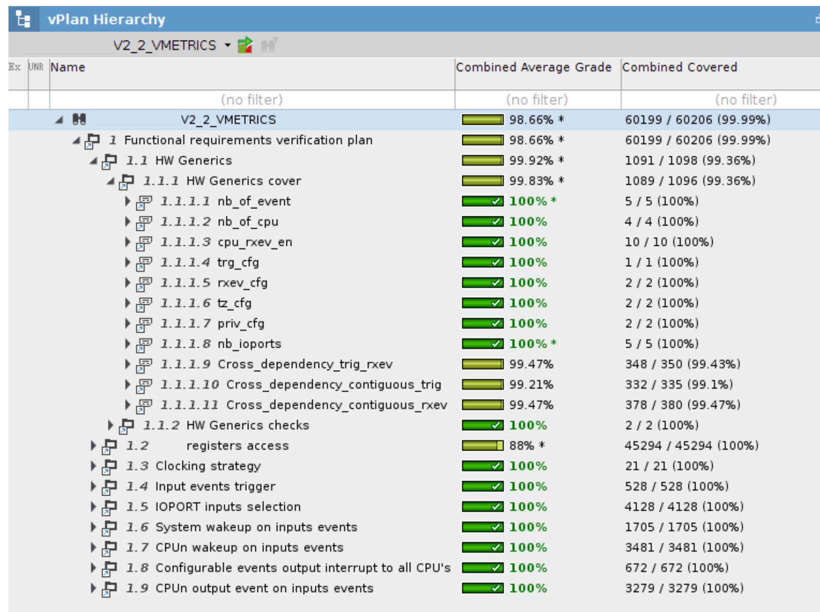

Figure 11. Configuration metrics driven flow vPlan coverage results

We observe no functional coverage degradation and better configuration metrics with an overall saved run time of 30.5%.

VII. CONCLUSION

The main difficulty in the formal verification of highly configurable designs is the management of the configuration space. Exhaustive verification is in most cases impossible and a selection of configurations to be verified is needed. By using correct functional coverage criteria to the parameter space, it is possible to find a good subset of configurations, which will achieve the verification target reducing the total number of formal runs.

This paper describes a simple random generation to generate the parameter values to be collected and eventually exercised. More advanced techniques to optimize configurations selection can be applied in order to further optimize execution runtime.

We have demonstrated a possible implementation of the two proposed flow.

We have observed that sufficient coverage target metrics can be achieved with less execution time effort by using the configuration metric driven approach. A good coverage metric specification is fundamental to achieve a low divergence in functional metric target while reducing the number of runs.

Both flows allow to reach the signoff target of a highly configurable digital IP in a sustainable runtime.

APPENDIX

```
struct config_s {
  nb_of_events : uint(bits: 7);
    keep nb_of_events in [16..95];

  nb_of_cpu    : uint(bits: 2);
    keep nb_of_events in [0..3];

  cpu_rxev_en  : uint(bits: 4);
    keep read_only(nb_of_cpu) < 3 => soft cpu_rxev_en[3:nb_of_cpu] == 0;

  trig_cfg     : uint(bits:96);
    keep nb_of_events < 95 => soft trig_cfg[95:nb_of_events+1] == 0;

  rxev_cfg     : uint(bits:96);
    keep nb_of_events < 95 => soft rxev_cfg[95:nb_of_events+1] == 0;

  tz_cfg       : uint(bits:96);
    keep read_only(nb_of_events) < 95 => soft tz_cfg[95:nb_of_events+1] == 0;

  event config_cover_e;
};

extend sys {

  number_of_configs : uint;
    keep number_of_configs == (get_symbol("RND_CFG_NUMBER")).as_a(uint);

  rnd_configs : list of config_s;
    keep rnd_configs.size() == number_of_configs;

  post_generate() is also {
    for each (config) in rnd_configs {
       var yml_filename : string = append(path,"/configs/yml/",config.hw_cfg,".yml");
       config_f = files.open(yml_filename, "w",  "Text file");
       writef(config_f, "nb_of_events : %d\n", config.nb_of_events);
       writef(config_f, "nb_of_cpu    : %d\n", config.nb_of_cpu   );
       writef(config_f, "cpu_rxev_en  : 4'h%x\n", config.cpu_rxev_en );
       writef(config_f, "trig_cfg     : 96'h%x\n", config.trig_cfg    );
       writef(config_f, "rxev_cfg     : 96'h%x\n", config.rxev_cfg    );
       writef(config_f, "tz_cfg       : 96'h%x\n", config.tz_cfg      );
    };
    emit config_cover_e;
  };
};
```

Figure 12.'e' code snippet for the full metric driven configurations generation

```
for gen in `seq 1 $maxGenNb`; do
  ### generate Specman command line
  specmanCmd="load $specmanScript;"
  for cfg in `seq 1 $expGen`; do
    specmanCmd="${specmanCmd} test -seed = random;"
  done
  ### call Specman to generate the configurations
  $bsubCmd specman -c "$specmanCmd"
  cfgNb=`ls -d ${covwork}/scope/gen_cfg* | wc -l`
  ### call IMC to merge the coverage
  $bsubCmd imc -execcmd "merge ${covwork}/scope/gen* \
                               -out ${covwork}/scope/merged_cfgs \
                               -message 0 -overwrite; \
                         load merged_cfgs; \
                         report_metrics -out metric -overwrite"
  ### evaluate the metric
  metric=`python3 -c "import json; \
                      f = open(\"$jsonData\"); \
                      data = json.load(f); \
                      val = (data[0]['All Cov']); \
                      (dummy, val) = val.split('('); \
                      (val, dummy) = val.split('%'); \
                      print(val)"`
  ### stop if the metric was reached
  targetReached=`echo $metric'>='$targetMetric | bc -l`
  if [ "$targetReached" -eq 1 ]; then
    echo "*** Metric Target $targetMetric Reached ***"
    break
  fi
done

$bsubCmd imc -execcmd "rank gen* -out ${rankResults}"
### extracting ranked configurations
cfgs=($(grep cov_work/scope/gen ${rankResults} | \
        grep -v ".ucm" | \
        awk '$2 != "0.00%"' | \
        awk -F/ '{print $NF}' | \
        perl -pe "s/gen_cfg_sn/rnd_cfg_/g"))
### crete top vsif file importing only ranked configurations vsif files
perl -pe "s/all_cfg/ranked_cfg/g" ${top_tmpl} > ${top_vsif_incr}
for cfg_name in "${cfgs[@]}"; do
  vsif_file=${vsif_dir}/${cfg_name}.vsif
  perl -pe "s/^(\s*)<include_cfg>/\$1\#include \"${vsif_file//\//\\/}\"\n\$1<include_cfg>/"
${top_vsif_incr} > ${top_vsif_tmp}
  mv ${top_vsif_tmp} ${top_vsif_incr}
done

perl -pe "s/^\s*<include_cfg>\s*$//" ${top_vsif_incr} > $out_top_vsif
rm -f ${top_vsif_incr}
```

Figure 13.'bash' code snippet for the configuration metric driven configurations generation

```
struct event_config_s {
  rtl_version     : rtl_version_t;
  event_index     : uint(bits:7);
  trig_cfg        : bit;
  trig_is_port    : bool;
    keep trig_is_port == (read_only(event_index) < 16);
  rxev_cfg        : bit;
  next_trig_cfg   : bit;
    keep soft next_trig_cfg == 0;
  next_rxev_cfg   : bit;
    keep soft next_rxev_cfg == 0;
  tz_cfg          : bit;


  event cover_event_config_e;
};

  cover cover_event__config_e is {
    item event_index using per_instance, ignore=(event_index>95);
    item trig_cfg;
    item rxev_cfg;
    item next_trig_cfg using no_collect;
    item next_rxev_cfg using no_collect;
    item tz_cfg;

    cross trig_cfg, rxev_cfg ;
    cross trig_cfg, next_trig_cfg;
    cross rxev_cfg, next_rxev_cfg;
  };
  cover config_cover_e is {
    item nb_of_events using
      ignore = (nb_of_events>95 or nb_of_events<16),
      illegal = (nb_of_events<16),
      ranges = {
        range([16]    , "Minimum number of events", UNDEF, 1);
        range([17..31], "Low number of events", UNDEF, 2);
        range([32..57], "Medium number of events", UNDEF, 2);
        range([58..94], "High number of events", UNDEF, 2);
        range([95]    , "Maximum number of events", UNDEF, 1);
      };
    item nb_of_cpu    ;
    item cpu_rxev_en  using
      ranges = {
        range([0]    , "ALL OFF", UNDEF, 1);
        range([1]    , "ONLY CPU0 ON", UNDEF, 1);
        range([2]    , "ONLY CPU1 ON", UNDEF, 1);
        range([4]    , "ONLY CPU2 ON", UNDEF, 1);
        range([8]    , "ONLY CPU3 ON", UNDEF, 1);
        range([3]    , "ONLY CPU0 CPU1 ON", UNDEF, 1);
        range([6]    , "ONLY CPU1 CPU2 ON", UNDEF, 1);
        range([0xC]  , "ONLY CPU2 CPU3 ON", UNDEF, 1);
        range([0xF]  , "ALL ON", UNDEF, 1);
        range([5,7,9,0xA,0xB,0xD,0xE]  , "SOME ON", UNDEF, 1);
      };
    item nb_ioports   using
      ranges = {
        range([0]       , "Minimum number of ports", UNDEF, 1);
        range([1..15]   , "Low number of ports", UNDEF, 4);
        range([16..127] , "Medium number of ports", UNDEF, 2);
        range([128..254], "High number of ports", UNDEF, 1);
        range([255]     , "Maximum number of ports", UNDEF, 1);
      };
  };
```

Figure 14. Configuration coverage implementation example

```
#ifndef SESSION;
#define SESSION;
session gen_rnd_cfgs {
  #include "vm_session.vsif"
};
#endif

group rnd_cfg_100133959 {
  #include "vm_group.vsif"
  timeout : "120000";
  hw_cfg  : rnd_cfg_100133959;
  ipver   : v2_2;
  run_script: "jg_vm_single_run.py";
  run_mode    : batch_debug;
  scan_script: "vm_scan.pl -maxpat 2000  $ENV(LDVHOME)/tools/bin/jg.flt";

  test all_props_rnd_cfg_100133959 {
    jg_app       : "fpv csr cov";
    jg_tcl_files : "$ENV(VERIF_BASE)/kit/run.tcl";
    jg_tasks     : "<embedded>:CSR";
  };
};
```

Figure 15. Generated vsif for single configuration execution example

REFERENCES

[1] Laurent Arditi, Anne-Claire Berger, “Sequential Equivalence Checking: The Summer of Love”, Cadence Club Formal France, Feb.2016.

[2] Samuel Dellacherie, “Formal Verification IPs: the corner stone for a broader adoption of Formal Verification”, https://www.design-reuse.com/articles/17065/formal-verification-ips.html

[3] Sundaravaradan Rangarajan, Vishwanath Balasubramanian,” Advanced Techniques for IP Design and Verification”, https://www.eetimes.com/document.asp?doc_id=1276118

[4] Jeroen Vliegen,” Automated Formal Verification of OCP based IP Cores”, https://www.design-reuse.com/articles/18495/ip-design-verification.html