A Hybrid Approach To Interrupt Verification

Giovanni Auditore, Francesco Rua, STmicroelectronics, General Purpose Microcontroller, Catania, Italy (giovanni.auditore@st.com, francesco.rua@st.com)

Qibo Peng, Politecnico di Torino, Torino, Italy (s288471@studenti.polito.it)

Abstract— **Verification of complex digital systems is a key aspect of the design process. In these systems, status flags and interrupts play a crucial role in the communication between different components, indicating the state of the system and triggering necessary actions. However, verifying these flags and interrupts can be a complex and time-consuming task, especially when relying solely on traditional simulation-based approaches.**

To address these challenges, a hybrid verification approach has been developed, combining simulation based dynamic verification and formal static verification. This approach reduces the complexity of the verification task and increases the reliability of the checks, making it an attractive solution for the status flag and interrupt verification.

The complexity of the verification task is reduced by modeling only the status flags and providing an interrupt connection description in an executable specification format. This enables the automatic generation of connectivity properties, reducing the manual effort required and improving the efficiency of the verification process.

In the case the IP (Intellectual Property) verification environment makes use of Formal Register Verification App such as Cadence Jasper CSR or Synopsys VCF FRV, to further optimize the automation, the solution takes advantage of the App debug information and use these in the automatically generated properties.

The reliability of the checks is increased using status flag set/clear events tracking in a Universal Verification Methodology (UVM) out-of-order scoreboard and applying configurable windowing tolerance to each scored item while matching them. This allows for accurate matching of the expected and actual flags, reducing the likelihood of false negatives and false positives.

Keywords—status-flags; interrupts; verification; hybrid; automation.

I. Introduction

Peripheral Interrupts outputs verification is a common, but sometime tricky task of digital IP verification. Most of the known solutions have limits of applicability, mostly affected by either the frequency of the interrupt condition occurrence (mainly for dynamic solutions) or the complexity of the status logic to be modelled (mainly for static solutions). Moreover, there is no automated solution available on the market and this often leads to IP specific solutions which are poorly reusable.

The proposed solution overcomes these limitations. Reusability is gained through abstraction and applicability is gained thanks to the flexibility of the hybrid static-dynamic approach.

The key factor was to detach the functionality which is IP specific from the interrupt logic architecture which can be made generic.

The specific function has been further decomposed to reduce the risk of hazards scenarios and to be able to verify concurrent events and the associated arbitration scheme.

The paradigm used in this work is: static property verification always goes first; dynamic verification comes after, only if needed.

In most peripherals interrupt logic controls and status are stored into registers. It is a widely adopted standard in the market to verify register files using formal register verification Apps such as Jasper CSR from Cadence or VCF FRV from Synopsys.

The solution reduces the complexity by reusing the formal model of the register verification App, this is because both vendor's solutions allow to access some debugging information which can be used while implementing the interrupt automatically generated properties.

A future application of the method (not part of this paper) could be to natively embed the interrupt connection description similarly to the register model description XML vendor extension format, and to fully use the App model without the workaround of deriving the required register field information from the debug ones.

II. APPLICATION

In most of the embedded systems each CPU present in the system handles an asynchronous event occurrence from one of the system components by means of interrupts.

From a hardware point of view each component implements one or more interrupt lines. Each single bit line may group one or more events, each representing a change of the component status. The software may optionally enable/disable some/all events linked to each line allowing dynamically configurable event propagation.

Once the interrupt line triggers, it is propagated by the interrupt controller to the CPU or CPUs controlling the given component. All simultaneous occurrences are then managed by the SW on a priority basis using the interrupt handler routines. The software upon interruption occurrence may access a more detailed set of status information by reading some internal memory region (normally a status register) of the component; it can also filter, upon application needs, some notification using interrupt enable and/or status flag enable control bits.

The proposed solution addresses the complete verification of the component status control and status event, together with the verification of the propagation of the enabled event occurrence towards the component interrupt output lines for a specified interrupt architecture.

To make the solution reusable the architecture specification of each interrupt line has been defined as generic as possible, leading to the abstract representation showed in Figure 1.

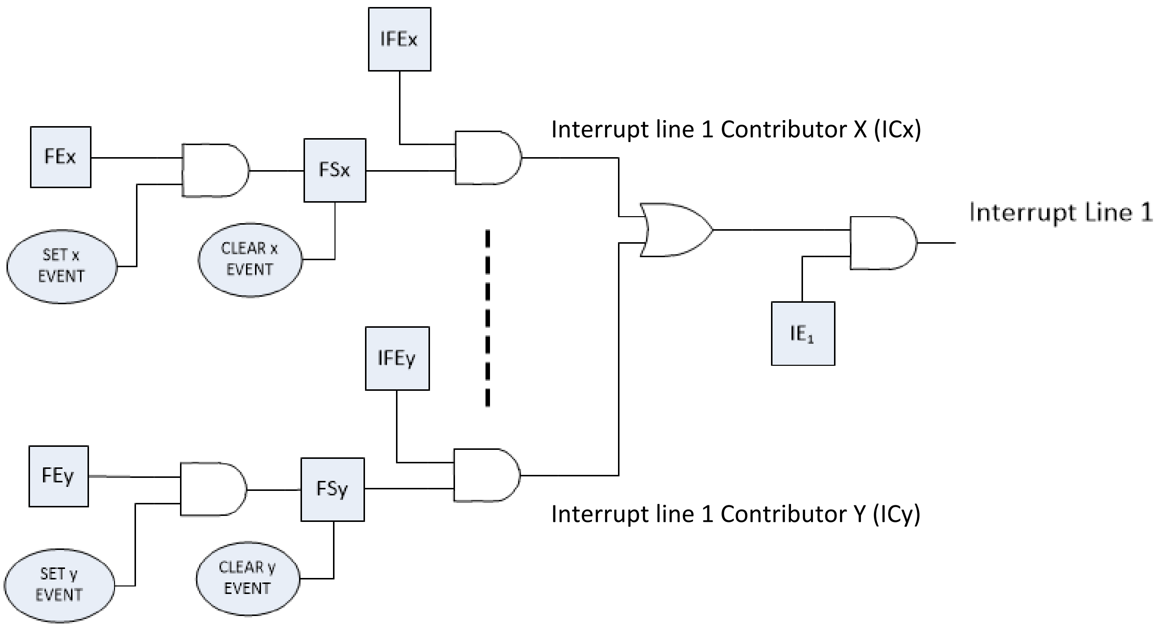


Figure 1: Abstract peripheral interrupt line architecture

In this architecture we consider a potential interrupt status flag X set event (SETx) which can be masked using an optional SW programmable Flag Enable bit (FEx). Upon a set event occurrence, the component status change is recorded into an interrupt Flag Status (FSx), but only if the flag enable bit FEx is set. Once set, a status flag is cleared when a clear event occurs (CLEARx).

One active (set) interrupt status flag can contribute to the final interrupt generation in a programmable way thanks to an Interrupt status Flag Enable bit (IFEx). Status flags are often grouped together to reduce the number of output interrupt lines. We named each of the elements into this group as Interrupt line Contributor (ICx).

Finally, the cumulative result of all active contributors is propagated to the Interrupt output Line 1, but only if enabled by the Interrupt line 1 Enable bit (IE_1)

Such structure can be repeated for all the peripheral interrupt outputs lines.

To automate the definition of a given component status flags and interrupts specification, we propose a format using the YAML meta-language. To further abstract the description in a more human readable format, we show in the proposed solution how such formalized description can be generated by a high-level executable implemented in Python language.

Back to the verification scope, it should be noted that most of the verification complexity resides in the SETx and CLEARx events occurrence prediction. This complexity can further be reduced whenever the CLEARx event is only originated from a SW access to an interrupt clear register, thus simplifying the clear prediction model to the same complexity as FEx, FSx, IFEx, IE_1 and similar register fields which are a subset of the register model and can be either provided by the user or , in a more effective way, extracted by the formal register model if any.

Given the above assumptions we expect that the full interrupt line architecture can be formally verified, with a good convergence expectation, if the SETx event has a simple formal model.

If this is not the case, the proposed solution is to decouple the SETx and CLEARx event complexity and verify it in a standalone dynamic environment for status flag verification.

The complete verification solution therefore includes two components:

- Interrupt architecture executable specification and related automated assertion generation
- Status Flag UVM VIP

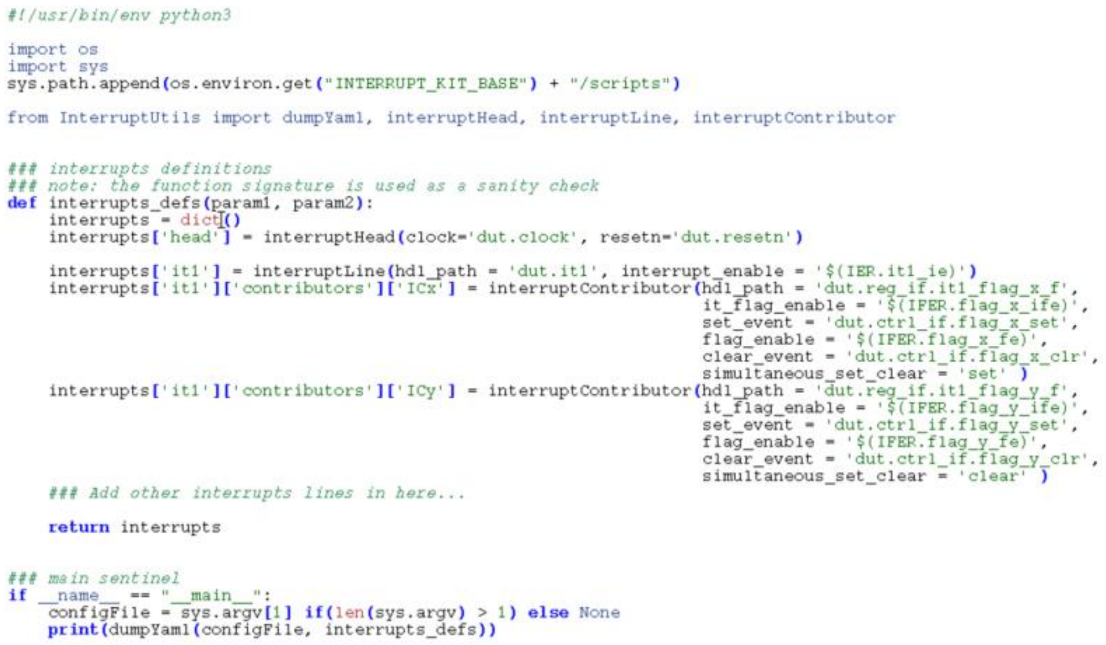
A. *Interrupt architecture executable specification*

We use a Python file to describe the generic architecture using dictionaries.

In this description:

- Each IP component has a dictionary of several interrupt lines, an optional default clock and an optional default resetn (active low reset).
- Each interrupt line has a dictionary of several interrupt contributors, one clock and one resetn which overrides the default ones if specified.
- Each interrupt line contributor has set and clear events, status flag enable, status flag, interrupt status flag enable.

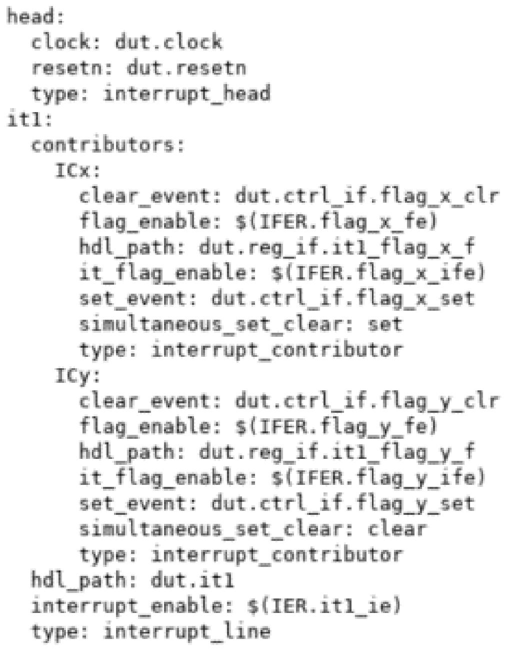
The code snippet shown in Figure 2, describes one example line specification. Note that the information stored into register fields is represented by the syntax “$(<REG_NAME>.<FIELD_NAME>)”.



```
#!/usr/bin/env python3

import os
import sys
sys.path.append(os.environ.get("INTERRUPT_KIT_BASE") + "/scripts")

from InterruptUtils import dumpYaml, interruptHead, interruptLine, interruptContributor


### interrupts definitions
### note: the function signature is used as a sanity check
def interrupts_defs(param1, param2):
    interrupts = dict()
    interrupts['head'] = interruptHead(clock='dut.clock', resetn='dut.resetn')

    interrupts['it1'] = interruptLine(hdl_path = 'dut.it1', interrupt_enable = '$(IER.it1_ie)')
    interrupts['it1']['contributors']['ICx'] = interruptContributor(hdl_path = 'dut.reg_if.it1_flag_x_f',
                                                                    it_flag_enable = '$(IFER.flag_x_ife)',
                                                                    set_event = 'dut.ctrl_if.flag_x_set',
                                                                    flag_enable = '$(IFER.flag_x_fe)',
                                                                    clear_event = 'dut.ctrl_if.flag_x_clr',
                                                                    simultaneous_set_clear = 'set' )
    interrupts['it1']['contributors']['ICy'] = interruptContributor(hdl_path = 'dut.reg_if.it1_flag_y_f',
                                                                    it_flag_enable = '$(IFER.flag_y_ife)',
                                                                    set_event = 'dut.ctrl_if.flag_y_set',
                                                                    flag_enable = '$(IFER.flag_y_fe)',
                                                                    clear_event = 'dut.ctrl_if.flag_y_clr',
                                                                    simultaneous_set_clear = 'clear' )

    ### Add other interrupts lines in here...

    return interrupts


### main sentinel
if __name__ == "__main__":
    configFile = sys.argv[1] if(len(sys.argv) > 1) else None
    print(dumpYaml(configFile, interrupts_defs))
```

Figure 2: Python example code snippet for interrupt lines description.

The Python code uses a utility library of functions to shorten the manually written code and make it more readable. The executable will dump the YAML formalized description, the dump result of the example line is shown in Figure 3.



```
head:
  clock: dut.clock
  resetn: dut.resetn
  type: interrupt_head
it1:
  contributors:
    ICx:
      clear_event: dut.ctrl_if.flag_x_clr
      flag_enable: $(IFER.flag_x_fe)
      hdl_path: dut.reg_if.it1_flag_x_f
      it_flag_enable: $(IFER.flag_x_ife)
      set_event: dut.ctrl_if.flag_x_set
      simultaneous_set_clear: set
      type: interrupt_contributor
    ICy:
      clear_event: dut.ctrl_if.flag_y_clr
      flag_enable: $(IFER.flag_y_fe)
      hdl_path: dut.reg_if.it1_flag_y_f
      it_flag_enable: $(IFER.flag_y_ife)
      set_event: dut.ctrl_if.flag_y_set
      simultaneous_set_clear: clear
      type: interrupt_contributor
  hdl_path: dut.it1
  interrupt_enable: $(IER.it1_ie)
  type: interrupt_line
```

Figure 3: YAML auto generated interrupt line description.

A library of parametric properties has been developed as a support and the script translates the above user executable specification into a set of assertion instances which can then be proven with any proof engine.

Please note that the user can either provide a testbench model of the register flags or the short syntax shown in the above example. The short syntax is automatically translated into a placeholder signal.

The link between the signal and the register app model of the corresponding register field is made by auto-generated assumptions in TCL format. This is needed to be able to have visibility of the register APP debug information (encrypted Register App property model needs to be compiled and elaborated before applying the assumptions).

The entire flow can be represented in Figure 4.

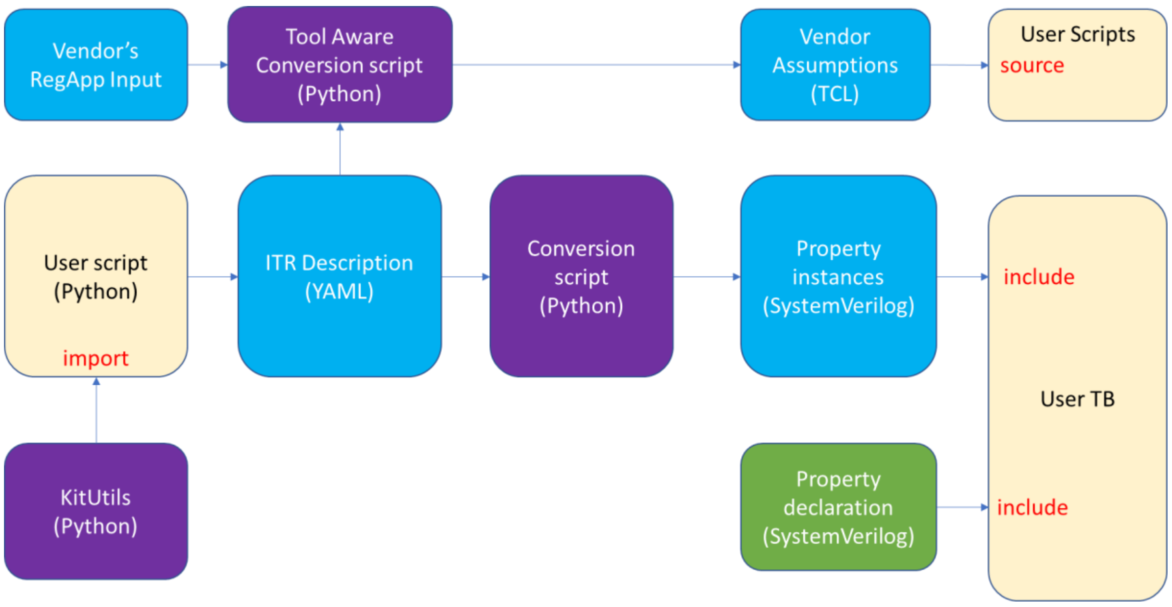
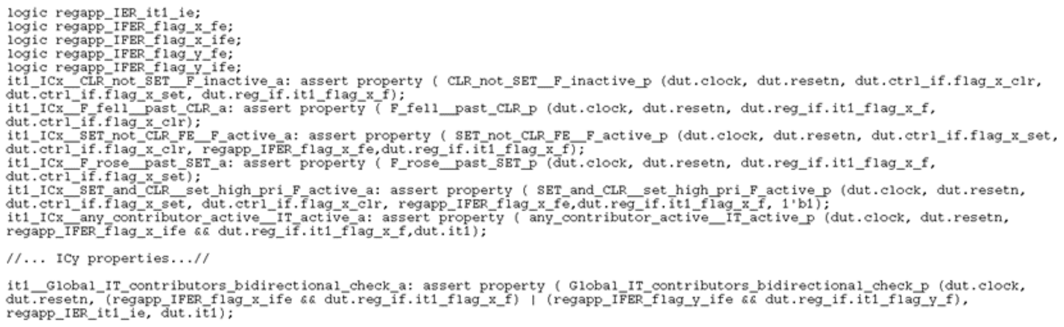


Figure 4: Automated interrupt and status flags properties flow.

The conversion script which reads the YAML interrupt executable description and dumps the generated properties, as well as the properties library file is provided in the Appendix.

The property library file is a set of parametrized properties needed to check that the abstract interrupt architecture behaves as specified. The generated assertion checks are instances of these properties having actual input parameters as specified in the YAML input file.

A partial view of the automatically generated assertions and register fields abstracted symbols to be linked on user responsibility by means of the generated assumption to the Reg App model is shown in **Error! Reference source not found.**.



```
logic regapp_IER_it1_ie;
logic regapp_IFER_flag_x_fe;
logic regapp_IFER_flag_x_ife;
logic regapp_IFER_flag_y_fe;
logic regapp_IFER_flag_y_ife;
it1_ICx__CLR_not_SET__F_inactive_a: assert property ( CLR_not_SET__F_inactive_p (dut.clock, dut.resetn, dut.ctrl_if.flag_x_clr,
dut.ctrl_if.flag_x_set, dut.reg_if.it1_flag_x_f);
it1_ICx__F_fell__past_CLR_a: assert property ( F_fell__past_CLR_p (dut.clock, dut.resetn, dut.reg_if.it1_flag_x_f,
dut.ctrl_if.flag_x_clr);
it1_ICx__SET_not_CLR_FE__F_active_a: assert property ( SET_not_CLR_FE__F_active_p (dut.clock, dut.resetn, dut.ctrl_if.flag_x_set,
dut.ctrl_if.flag_x_clr, regapp_IFER_flag_x_fe,dut.reg_if.it1_flag_x_f);
it1_ICx__F_rose__past_SET_a: assert property ( F_rose__past_SET_p (dut.clock, dut.resetn, dut.reg_if.it1_flag_x_f,
dut.ctrl_if.flag_x_set);
it1_ICx__SET_and_CLR__set_high_pri_F_active_a: assert property ( SET_and_CLR__set_high_pri_F_active_p (dut.clock, dut.resetn,
dut.ctrl_if.flag_x_set, dut.ctrl_if.flag_x_clr, regapp_IFER_flag_x_fe,dut.reg_if.it1_flag_x_f, 1'b1);
it1_ICx__any_contributor_active__IT_active_a: assert property ( any_contributor_active__IT_active_p (dut.clock, dut.resetn,
regapp_IFER_flag_x_ife && dut.reg_if.it1_flag_x_f,dut.it1);

//... ICy properties...//

it1__Global_IT_contributors_bidirectional_check_a: assert property ( Global_IT_contributors_bidirectional_check_p (dut.clock,
dut.resetn, (regapp_IFER_flag_x_ife && dut.reg_if.it1_flag_x_f) | (regapp_IFER_flag_y_ife && dut.reg_if.it1_flag_y_f),
regapp_IER_it1_ie, dut.it1);
```

Figure 5: Automatically generated example properties and register field logic symbols.

As previously described, all assertion inputs referencing a register field in the formal App model are connected by means of automatically generated TCL assume commands. The script which generates such TCL commands, extracts from the YAML interrupt specification the subset of required information and uses the register App input register description to create the path to the register App debug signal containing the relevant information. It then dumps for each reference a TCL assume command in vendor specific format into a TCL file. Each assume command links the property input signal to the internal debug information. The generated TCL file can then be sourced by the user in its run flow after the register app model is loaded.

B. *Status Flag UVM VIP*

Whenever the auxiliary code modelling of the SETx event is too complex for a pure formal static environment, we assume as golden the RTL SETx event in the static environment and provides a VIP component to check the correctness of the related status flag in a dynamic verification environment. We will not describe in detail the architecture of the whole VIP itself since it follows closely the UVM standard.

The novelty of our paper is in the use of an out of order scoreboard to record and match SET/CLEAR event occurrence in the DUT against the ones predicted by the testbench.

With this strategy the VIP is no longer sensitive to status flag timings, and as such the SET/CLEAR models do not need to be cycle accurate.

To understand the advantage of this approach let’s consider a flag status checker with additional windowing tolerance as depicted in Figure 6.

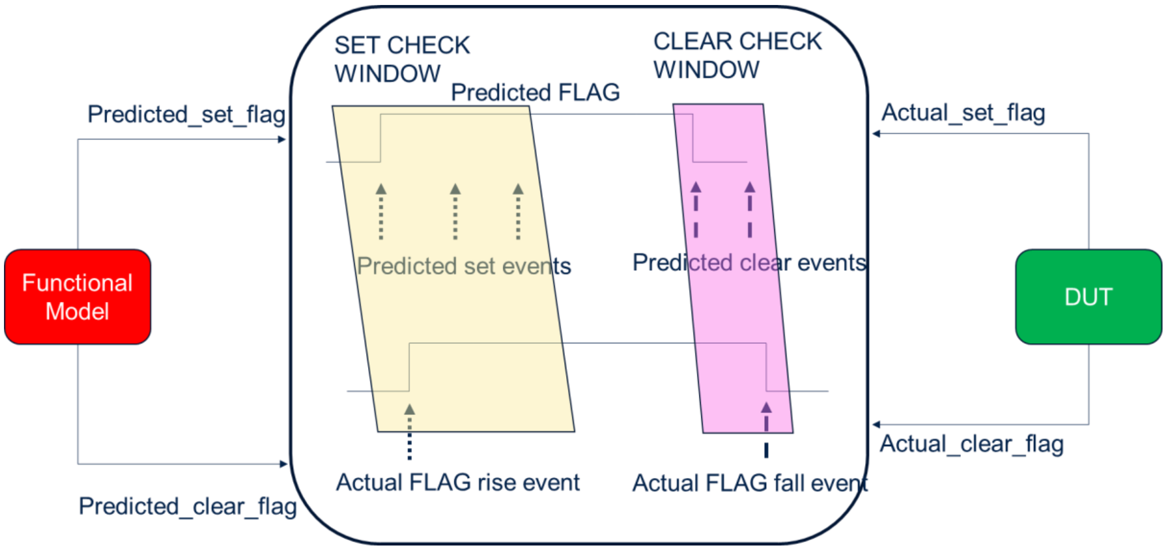


Figure 6: Typical status flag checker with additional windowing tolerance.

If the checker compares the predicted flag wave against the actual one using the window tolerance it works well as long as the set and clear occurrence is not too close. When hitting this kind of corner scenario this strategy may result into false failures as described in Figure 7.

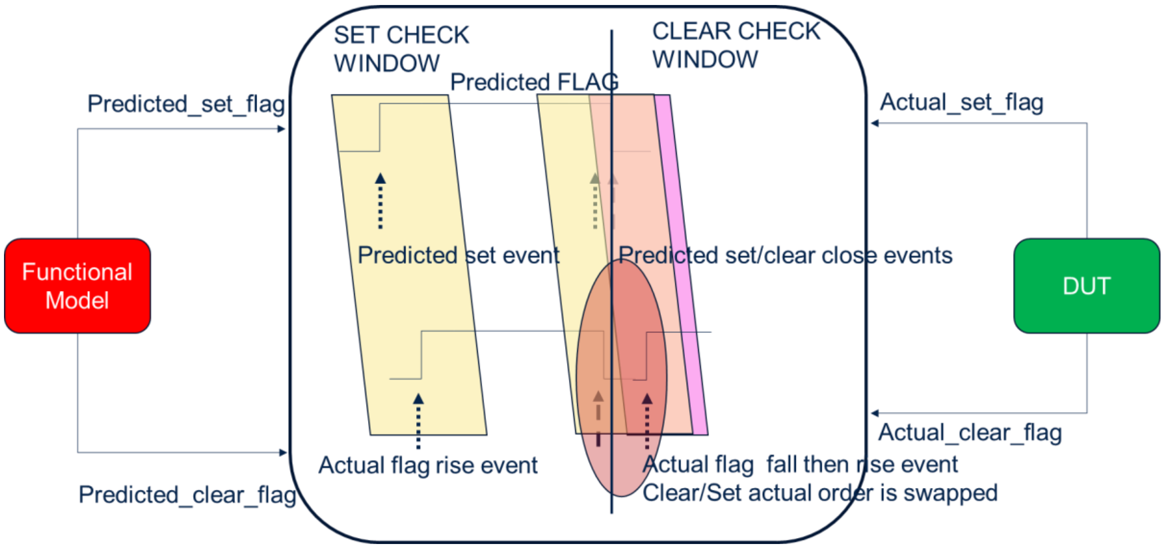


Figure 7: Pitfall scenario of a standard windowing flag checker.

As result of the corner case scenario described in the picture the predicted status flag value is low while the actual flag value will rise back again ending into high state. In this and in other similar scenarios a cycle accurate modelling of the predicted set/clear events would be required by the state-of-the-art methodology in order to avoid false failures.

The proposed VIP strategy instead only scores and matches all the predicted set events against the actual RTL set event (these are the SETx events in the static verification environment). As long as all predicted set events match the DUT actual SETx events, and all the predicted clear events match the DUT actual CLRx events we can be reasonably sure that the DUT behavior is correct. Out of order matches are allowed to successfully check the previously described corner scenario avoiding false failures. On top of that, to be sure that the match is consistent also with respect to time, the scoreboard model adds a timeout mechanism, which requires the new item inserted to be matched within a timeout tolerance window.

The complete VIP checking strategy is summarized in Figure 8.

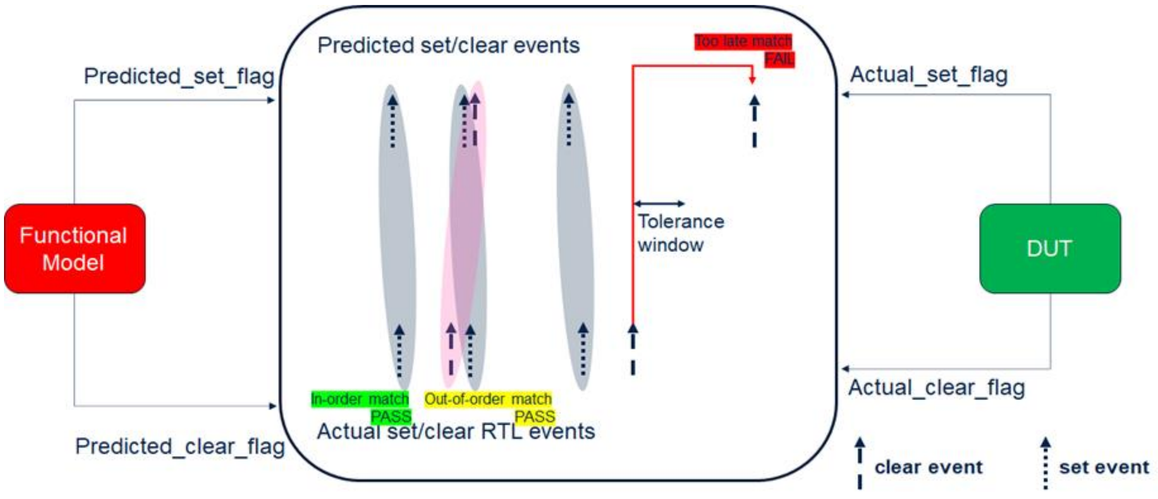


Figure 8: SET/CLEAR out of order scoreboard with timeout checking strategy.

Since the scoreboard only checks SET/CLEAR occurrences, but a bug may reside in the transfer of the DUT SET/CLEAR actual event to the status flag value, the VIP exposes an SV interface which requires to connect predicted SET/CLEAR events, DUT actual SET/CLEAR events and the DUT status flag itself. The interface is also exposing a parameter to configure SET vs CLEAR priority, this to properly handle a simultaneous events occurrence scenario.

This interface shown in Figure 9 also implements simple assertions to check the consistency of DUT SET/CLEAR actual events to the DUT actual status flag value.

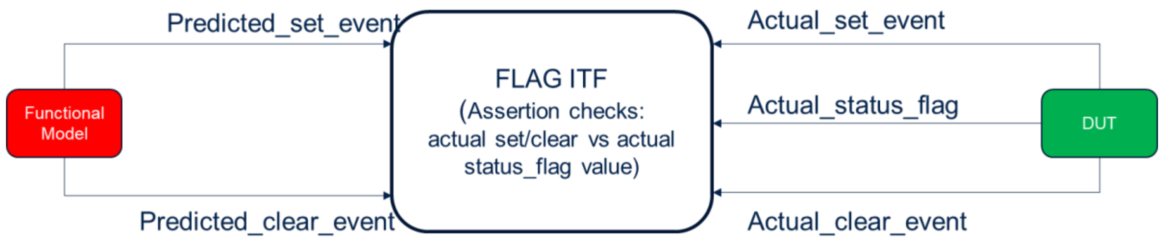


Figure 9: VIF interface embedding actual status flag assertion checks.

The VIP component and its interface are completely abstract and independent of the DUT functionality, so that they can be used for any DUT status flag check. Hence, the user only needs to provide the predicted input events and connect the actual DUT relevant signals.

III. CONCLUSIONS

We propose a very robust, fully reusable, and widely applicable automated verification solution for peripheral interrupt line verification.

The proposed method:

- Formalizes an executable description of the interrupt architecture.
- Reduces the development time by means of automatic assertion generation.
- Extends, if required, the domain of applicability thanks to the proposed hybrid verification approach.
- Reduces the risks of false failures in dynamic verification, thanks to the out-of-order set/clear events collection and matching.
- May optionally exploit commercial App register models, whenever they are used in the verification environment, to reduce register flag modelling effort.

As future development we foresee the possibility that the commercial EDA register Apps ease furthermore the integration process, or even provide a fully integrated commercial interrupt verification App.

ACKNOWLEDGMENT

Special thanks to Prof. Luciano LAVAGNO (Politecnico of Torino University) for his constant support during the solution definition.

REFERENCES

[1] Tejbal Prasad, Shalini Damani, “Plug-n-play UVM Environment for Verification of Interrupts in an IP” - Design and Reuse IP-SOC Conference 2013.

[2] Christoph Rumpler, Alexander W. Rath, Sebastian Simon, Heinz Endres, “A UVM-based Approach for Rapidly Verifying Digital Interrupt Structures” - DVCON 2016.

[3] Eduard Cerny, Surrendra Dudani, John Havlicek, Dmitry Korchemny, “SVA: The Power of Assertions in SystemVerilog”

[4] Chris Spear and Gref Tumbush, “SystemVerilog for Verification. (Third edition)” - Springer, April 2014.

IV. APPENDIX

A. *Python Interrupt Utils (interrupt_utils.py)*

```
#!/usr/bin/env python3

from typing import Callable
import yaml

### collecting parameters from a YAML file
### returning a YAML description from a callback
def dumpYaml(configFile: str, cbs: Callable):
    config = dict()
    if(configFile):
        with open(configFile) as f:
            config = yaml.safe_load(f) or dict()
    return yaml.dump(cbs(**config))


### Interrupt head utility
def interruptHead(**kwargs):
    kwargs['type'] = "interrupt_head"
    return kwargs


### interrupt line utility
def interruptLine(**kwargs):
    kwargs['type'] = "interrupt_line"
    kwargs['contributors'] = dict()
    return kwargs


### interrupt contributor utility
def interruptContributor(**kwargs):
    kwargs['type'] = "interrupt_contributor"
    return kwargs
```

B. Python property generator (itr2sva.py)

```
#!/usr/bin/env python3

import yaml
import sys

configFile = sys.argv[1]

config = dict()
if(configFile):
    with open(configFile) as f:
        config = yaml.safe_load(f) or dict()


logic_defs = dict()

def dollar_syntax(reg_path: str) -> str:
    elaborated_path = reg_path

    if elaborated_path.startswith("$") :
        regname = elaborated_path.split(".")[0].split("(")[1]
        fieldname = elaborated_path.split(".")[1][:-1]
        elaborated_path = f"regapp_{regname}_{fieldname}"
        logic_defs[elaborated_path] = True

    return elaborated_path


output = list()

head = config.pop('head')
for interrupt_name, interrupt_line in config.items() :
    clk = interrupt_line.get('clock', head.get('clock'))
    rstn = interrupt_line.get('resetn', head.get('resetn'))

    assert 'hdl_path' in interrupt_line, "ERROR: interrupt line path must be
specified"
    it_line_path = interrupt_line.get('hdl_path')

    it_enable = interrupt_line.get('interrupt_enable', "1'b1")
    it_enable = dollar_syntax(it_enable)

    contributors = list()

    for contributor_name, contributor in interrupt_line['contributors'].items() :
        clr_evt = contributor.get('clear_event', "1'b0")
        set_evt = contributor.get('set_event', "1'b0")
        flag_en = contributor.get('flag_enable', "1'b1")
        flag_en = dollar_syntax(flag_en)
        it_flag_en = contributor.get('it_flag_enable', "1'b1")
        it_flag_en = dollar_syntax(it_flag_en)
        set_wins = contributor.get('simultaneous_set_clear', 'set') == 'set'
        set_wins_bin = "1'b1" if set_wins else "1'b0"

        assert 'hdl_path' in contributor, "ERROR: contributor status flag path
must be specified"
        it_status_flag = contributor.get('hdl_path')

        contributor_active = f"{it_flag_en} && {it_status_flag}"

        prop_name = 'CLR_not_SET__F_inactive'
        asrt_name = f"{interrupt_name}_{contributor_name}__{prop_name}_a"
        port_map = f"{clk}, {rstn}, {clr_evt}, {set_evt}, {it_status_flag}"
        output.append(f"{asrt_name}: assert property ( {prop_name}_p
({port_map});")

### …continues…
```

C. Python property generator (itr2sva.py), continued.

```
### …continued…
        prop_name = 'F_fell__past_CLR'
        asrt_name = f"{interrupt_name}_{contributor_name}__{prop_name}_a"
        port_map = f"{clk}, {rstn}, {it_status_flag}, {clr_evt}"
        output.append(f"{asrt_name}: assert property ( {prop_name}_p
({port_map});")

        prop_name = 'SET_not_CLR_FE__F_active'
        asrt_name = f"{interrupt_name}_{contributor_name}__{prop_name}_a"
        port_map = f"{clk}, {rstn}, {set_evt}, {clr_evt},
{flag_en},{it_status_flag}"
        output.append(f"{asrt_name}: assert property ( {prop_name}_p
({port_map});")

        prop_name = 'F_rose__past_SET'
        asrt_name = f"{interrupt_name}_{contributor_name}__{prop_name}_a"
        port_map = f"{clk}, {rstn}, {it_status_flag}, {set_evt}"
        output.append(f"{asrt_name}: assert property ( {prop_name}_p
({port_map});")

        prop_name = 'SET_and_CLR__set_high_pri_F_active'
        asrt_name = f"{interrupt_name}_{contributor_name}__{prop_name}_a"
        port_map = f"{clk}, {rstn}, {set_evt}, {clr_evt},
{flag_en},{it_status_flag}, {set_wins_bin}"
        output.append(f"{asrt_name}: assert property ( {prop_name}_p
({port_map});")

        prop_name = 'any_contributor_active__IT_active'
        asrt_name = f"{interrupt_name}_{contributor_name}__{prop_name}_a"
        port_map = f"{clk}, {rstn}, {contributor_active},{it_line_path}"
        output.append(f"{asrt_name}: assert property ( {prop_name}_p
({port_map});")

        contributors.append("(" + contributor_active + ")")


    interrupt_contributors = " | ".join(contributors)
    prop_name = 'Global_IT_contributors_bidirectional_check'
    asrt_name = f"{interrupt_name}__{prop_name}_a"
    port_map = f"{clk}, {rstn}, {interrupt_contributors}, {it_enable},
{it_line_path}"
    output.append(f"{asrt_name}: assert property ( {prop_name}_p ({port_map});")


output = [f"logic {signame};" for signame in logic_defs] + output
print("\n".join(output))
```

D. Properties library.

```
//Pre-defined parameterised property set used for interrupt static formal
verification

//If CLR and not SET  implies in next cycle FLAG is inactive
property CLR_not_SET__F_inactive_p (clk, rstn, clr_evt, set_evt, it_status_flag);
  @(posedge clk)
  disable iff(!rstn)
  clr_evt
  && !set_evt
  |=> !(it_status_flag);
endproperty:CLR_not_SET__F_inactive_p

//If FLAG fell implies in past cycle CLR was triggered
property F_fell__past_CLR_p (clk, rstn, it_status_flag, clr_evt);
  @(posedge clk)
  disable iff(!rstn)
  (1'b1 ##1 $fell(it_status_flag))
  |-> $past(clr_evt);
endproperty:F_fell__past_CLR_p

//If SET and not CLR and FE implies in next cycle FLAG is active
property SET_not_CLR_FE__F_active_p (clk,rstn, set_evt, clr_evt, flag_en,
it_status_flag);
  @(posedge clk)
  disable iff(!rstn)
  set_evt
  && !(clr_evt)
  && (flag_en)
  |=> it_status_flag;
endproperty:SET_not_CLR_FE__F_active_p

//If FLAG rose implies in past cycle SET was triggered
property F_rose__past_SET_p (clk,rstn, it_status_flag, set_evt);
  @(posedge clk)
  disable iff(!rstn)
  (1'b1 ##1 $rose(it_status_flag))
  |-> $past(set_evt);
endproperty:F_rose__past_SET_p

//If SET and CLR implies in next cycle if set_high_pri then FLAG is active else
is inactive
//in this case, CLR is over SET, only when FLAG has already been high
property SET_and_CLR__set_high_pri_F_active_p (clk, rstn, set_evt, clr_evt,
flag_en,  it_status_flag, set_wins);
  @(posedge clk)
  disable iff(!rstn)
  (set_evt && flag_en)
  && (it_status_flag && clr_evt)
  |=> set_wins ?(it_status_flag): (!(it_status_flag));
endproperty:SET_and_CLR__set_high_pri_F_active_p

property any_contributor_active__IT_active_p (clk, rstn, Interrupt_contirbutor_i,
it_name);
  @(posedge clk)
  disable iff(!rstn)
  Interrupt_contirbutor_i |-> it_name;
endproperty:any_contributor_active__IT_active_p

//for a group of contributor(it_status_flag) and contributor mask (it_flag_en)
property Global_IT_contributors_bidirectional_check_p (clk, rstn,
Interrupt_contributors, interrupt_enable, it_name);
  @(posedge clk)
  disable iff(!rstn)
  it_name == (Interrupt_contributors && interrupt_enable);
endproperty:Global_IT_contributors_bidirectional_check_p
```

E. VIP out-of-order scoreboard with timeout.

```
//declare the analysis implementation ports for incoming transactions
`uvm_analysis_imp_decl(_dut)
`uvm_analysis_imp_decl(_exp)

class flag_scoreboard extends uvm_component;

  //implemention ports for dut and monitor(exp)
  uvm_analysis_imp_dut #(flag_transaction_item, flag_scoreboard) dut_in_imp;
  uvm_analysis_imp_exp #(flag_transaction_item, flag_scoreboard) exp_in_imp;

  //queues holding the transactions from difference sources
  flag_transaction_item       dut_q[$];
  flag_transaction_item       exp_q[$];
  //The size of search_q maximumly is 1
  flag_transaction_item       search_q[$];
  //reference queue which would be used to compare with item in search_q
  flag_transaction_item       save_q[$];

  //To get the process_id and kill the process when flush is triggered
  local process               process_q[$];
  local semaphore             sema;

  virtual flag_if   vif;

  //default value of bit is 0
  bit                done;
  // if one more item in save queue out of the tolerance window,
  //without any counterpart, no need to do check_phase again.
  bit                is_need_check_save_queue_empty = 1;

  `uvm_component_utils(flag_scoreboard)

  extern function new(string name, uvm_component parent);
  extern virtual function void build_phase(uvm_phase phase);
  //at the end of the test we need to check that the two queues are empty
  //check phase is execued at the end of simulation
  extern function void check_phase(uvm_phase phase);
  extern function void write_dut(flag_transaction_item dut_trans);
  extern function void write_exp(flag_transaction_item exp_trans);
  extern function void search_and_compare(flag_transaction_item a_trans, bit
is_dut);
  //kill thread, search_and_compare(), before creating the sema again. foreach
process in queue, call process_q[i].kill()
  //Question: where to call the flush function? in the run_phase?
  //Answer:whenver you call flush() in environment, the component implement this
function would execute this function automatically.
  extern virtual function void flush();
  extern function bit is_tolerance_window_fulfilled(flag_transaction_item
ref_item, flag_transaction_item get_item);

endclass:flag_scoreboard

function flag_scoreboard::new(string name, uvm_component parent);
  super.new(name, parent);
endfunction:new

function void flag_scoreboard::build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(! uvm_config_db #(virtual flag_if)::get(this,"","vif", vif)) begin
    `uvm_fatal(get_type_name(), "DUT interface not found")
  end
  dut_in_imp = new("dut_in_imp", this);
  exp_in_imp = new("exp_in_imp", this);
  sema = new(1);
endfunction:build_phase
```

F. VIP out-of-order scoreboard with timeout (continued).

```
function void flag_scoreboard::check_phase(uvm_phase phase);
  super.check_phase(phase);
  if(is_need_check_save_queue_empty) begin
    //we need to disable assertions for such a kind of verification. They cannot
be caught
    `ifndef VIP_UNDER_TEST
    COMPARATOR_SAVE_Q_NOT_EMPTY_ERR : assert(save_q.size()==0) else begin
    `else
    if (save_q.size()!=0) begin
    `endif
      `uvm_error(get_type_name(), "MISMATCHED_SAVE_QUEUE_NOT_EMPTY_QUEUE_ERROR
detected from vip scoreboard")
    end
  end
endfunction:check_phase

function void flag_scoreboard::write_dut(flag_transaction_item dut_trans);
  search_and_compare(dut_trans, 1);
endfunction:write_dut

function void flag_scoreboard::write_exp(flag_transaction_item exp_trans);
  search_and_compare(exp_trans, 0);
endfunction:write_exp

function void flag_scoreboard::search_and_compare(flag_transaction_item a_trans,
bit is_dut);
  fork
    begin
      int indexes[$];
      //get the handle of the current process
      process p = process::self();
      // get the sema once process is created.
      //Because there would be many action try to execute this function,
      //lock the resources.
      sema.get();
      `uvm_info(get_type_name(), "Creating one process in vip scoreboard to do
search and compare. Get semaphore.", UVM_MEDIUM)
      // push process handle ASAP after it get the semaphore, to record which
process is using resources
      process_q.push_back(p);
      search_q = is_dut ? exp_q : dut_q;
      save_q   = is_dut ? dut_q : exp_q;
      done = 0;

      indexes = search_q.find_first_index(it) with  (a_trans.shallow_match(it));

      if(indexes.size()==0) begin //no flag match with a_trans, even shallow
match
        if(save_q.size()>=1) begin
          `ifndef VIP_UNDER_TEST
          COMPARATOR_ONE_MORE_ITEM_SAVE_QUEUE_TIME_OUT_ERR: assert
(is_tolerance_window_fulfilled(save_q[0], a_trans)) else begin
          `else
          if (!is_tolerance_window_fulfilled(save_q[0], a_trans)) begin
          `endif
            //one more item in save queue out of the tolerance window, without
any counterpart, detected from vip scoreboard
            is_need_check_save_queue_empty = 0;
            `uvm_error(get_type_name(), "ONE_MORE_ITEM_SAVE_QUEUE_TIME_OUT_ERROR
detected from vip scoreboard")
            `uvm_info(get_type_name(), $sformatf("the oldest item is observed at
%0t", save_q[0].observe_time), UVM_MEDIUM)
            `uvm_info(get_type_name(), "Aleady find one error in one vip
scoreboard, simulation should stop now!", UVM_MEDIUM)
          end
        end
```

G. VIP out-of-order scoreboard with timeout (continued).

```
        //if no match target, even shallow match, as long as the tolerance window
is not violated,
        //save it to save_q and can be compared later
              save_q.push_back(a_trans);
        //clone save_q to dut_q or exp_q
        if (is_dut) begin
          dut_q = save_q;
        end else begin
          exp_q = save_q;
        end
        `uvm_info(get_type_name(), "Get one item saved in Save Queue",
UVM_MEDIUM)
        // cannot do further compare because there is no matched one on target
flag, even shaollow match.
              done = 1;
      end

      //shallow matching meet: flag type matched, time to check deep match or
not, if not, shallow match
      if(!done) begin
        `uvm_info(get_type_name(), "Now start to compare save queue and search
queue",UVM_MEDIUM)
        //if 1, deep match, everything is perfectly matched, NO_ERROR
        `ifndef VIP_UNDER_TEST
        COMPARATOR_NO_ERR: assert(a_trans.compare(search_q[indexes[0]])) begin;
        `else
        if (a_trans.compare(search_q[indexes[0]])) begin
        `endif
          `uvm_info(get_type_name(), "NO_ERROR detected from vip scoreboard",
UVM_MEDIUM)
        `ifndef VIP_UNDER_TEST
        COMPARATOR_SHALLOW_MATCH_ERR:
assert(a_trans.shallow_match(search_q[indexes[0]])) begin
        `else
        end else if (a_trans.shallow_match(search_q[indexes[0]])) begin
        `endif
        //flag types are matched, but cannot meet the requirement of tolerance
window -> SHALLOW_MATCH_ERROR
          `uvm_error(get_type_name(), "SHALLOW_MATCHED_ERROR detected from vip
scoreboard")
          `uvm_info(get_type_name(), "Aleady find one error in one vip
scoreboard, simulation should stop now!", UVM_MEDIUM)

        end
        //clear search queue
        // there should always one item in search queue
        search_q.delete(indexes[0]);

        //clear mirrored search queue, ready for next transaction.
        if(is_dut) begin
          exp_q.delete(indexes[0]);
        end else begin
          dut_q.delete(indexes[0]);
        end

      end
      //pop process handle before release sema, means this process has released
resources. No need to kill again if flush() is triggered
      void'(process_q.pop_front());
      `uvm_info(get_type_name(), "Current process finished. Put semaphore",
UVM_MEDIUM)
       //release sema when everything is done
       sema.put();
    end
  join_none
endfunction:search_and_compare
```

H. VIP out-of-order scoreboard with timeout (continued).

```
function void flag_scoreboard::flush();
  `uvm_info(get_type_name(), "FLUSH DETECTED IN VIP SCOREBBOARD",  UVM_LOW)
  foreach(process_q[i]) begin
    process_q[i].kill();
  end

  this.search_q.delete();
  this.save_q.delete();
  this.sema.put();
endfunction:flush


function bit flag_scoreboard::is_tolerance_window_fulfilled(flag_transaction_item
ref_item, flag_transaction_item get_item);

  //record the observe time when creating the item in vip monitor.
  real ref_item_observe_time = ref_item.observe_time;
  real get_item_observe_time = get_item.observe_time;
  real ref_item_clk_period = ref_item.clk_period;
  real get_item_clk_period = get_item.clk_period;
  real target_tolerance_window = ref_item.tolerance_window;
  real diff_clk_ref_get = (ref_item_observe_time/ref_item_clk_period) -
(get_item_observe_time/get_item_clk_period);
  //due to difference sequence of ref item and get item,
  //the difference of time might be negative, should turn it to positive
  if (diff_clk_ref_get < 0) begin
    diff_clk_ref_get = get_item_observe_time/get_item_clk_period -
ref_item_observe_time/ref_item_clk_period;
  end

  if(diff_clk_ref_get <= target_tolerance_window) begin
    return 1;//tolerance window fulfilled
  end else begin
    return 0;// tolerance window violated
  end

endfunction:is_tolerance_window_fulfilled
```