2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Lean Verification Techniques: Executable SystemVerilog UVM Defect Table For Simulations

Steve Burchfiel[1], Kevin Schott[2], Kamel Belhous[3], Paul Ulrich[4]

[1]CorrectDesigns, steve.burchfiel@correctdesigns.com, Austin TX, USA

[2]CorrectDesigns, kevin.schott@correctdesigns.com, Austin TX, USA

[3]Teradyne, kamel.belhous@teradyne.com, Boston MA, USA

[4]Teradyne, paul.ulrich@teradyne.com, Boston MA, USA

***Abstract*—This paper presents a novel approach for an 'executable' defect table to track and work around open issues in the design and verification environments. The defect table is source code, using Systemverilog UVM that is built into the verification environment which tracks the state of an issue, affecting the behavior of the testbench based on the defect state.**

I. Introduction

The task of verifying a complex chip design is a gradual and tedious process. During the verification effort, many bugs will be found in the design as well as the verification environment. But how do the verification engineers keep the simulation model stable, prevent regression failures on features that once worked, and maintain passing regressions to assess the quality of the design? All this while new RTL, checkers, and tests are being added and debugged? How can verification engineers make progress, without being slowed down by the existing bugs which have not been fixed yet by the design teams. One way is to use the defect table. The defect table is a good example of a lean verification technique that enhances verification productivity and quality by enabling engineering the exact capabilities of every verification build and reducing the need for difficult to track "side models". Eliminating (or reducing/minimizing) waste/waiting, also called delays in agile methodologies, is the most fundamental lean principle, the one from which all the other principles follow [1]. The defect table provides the ability to mitigate productivity losses, due to the impact of the delays introduced by design, specification, and verification changes for bug fixes. A defect table also enables the verification environment to strategically work around specific bugs in the design and the verification environment such that existing regression suites can continue to be used to qualify changes and find new, unknown flaws in the code base. The ability to maintain and enhance the verification regression environment while bugs are continually found and fixed is a key benefit of this lean process.

The defect table provides a disciplined approach to using a designated class to keep track of workarounds in the testbench when bugs exist in the design, and/or new unsupported features have been added that might otherwise cause regressions to fail. This gives the designers and verification engineers a much more flexible verification environment that can adapt to bugs and new features. The defect table acts as the de-facto location where "TODO"s are placed. Furthermore, it can affect the testbench's behavior to provide run time decisions on whether a work around should be applied or not. For example, if verification engineers add new tests, checkers and coverage for a bus interface only to discover every transactions fails because one field of every transaction fails because of an RTL defect which has been filed in the official bug tracking system, the verification engineer can create a defect table entry and disable checking of the one field. This allows the entire suite of tests, checkers, and coverage to be released while the designer debugs and fixes the problem. In order to reproduce and verify the RTL issue, the design only needs to remove the defect table entry (or mark it resolved). This provides designers and verification engineers with a stable model, and easy way to verify fixes. The defect table also provides a means to review the complete list of work arounds and TODOs to assure the proper work arounds have been applied or removed prior to tape-out. This is superior to trying to figure out which code in the environment is commented out, which testcases have been disabled or which code was never written or released.

Some of information from the executable SystemVerilog UVM defect table comes from the official bug tracking systems. The defect table is not a replacement for the existing official bug tracking systems. It is a powerful technique for projects involving high RTL churn, frequent features/design changes, and/or long turnaround times from the time an RTL issue is found to when it is fixed.

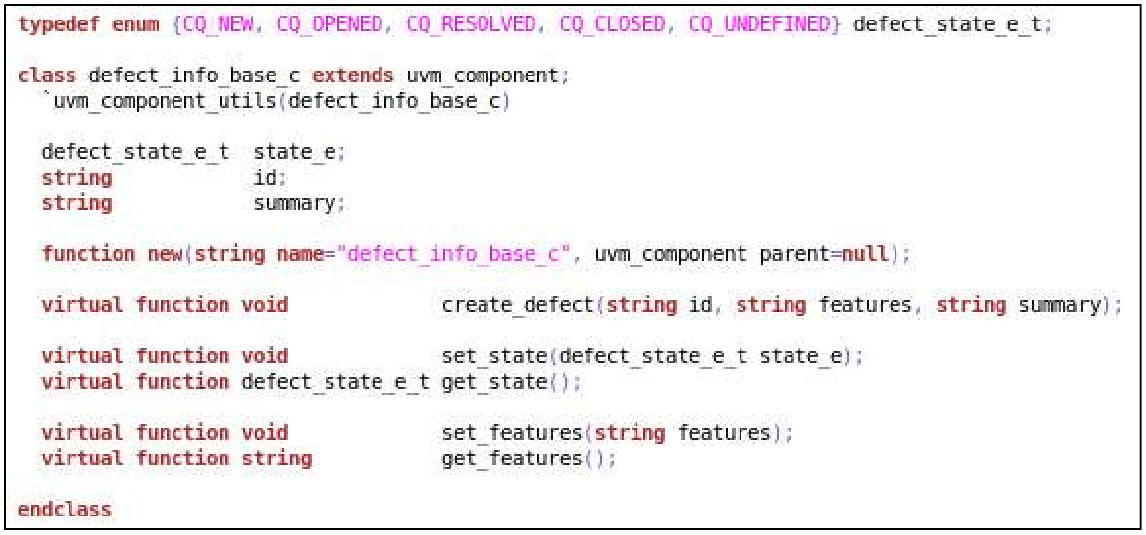
II. BUILDING THE DEFECT TABLE

A. Building the defect table

The source code for a defect table should be project and block independent such that a defect table can be reused and integrated into multiple environments if desired. Creating a defect table from a common, object oriented base class in SystemVerilog is one way to create a reusable defect table implementation. The base classes provide the APIs for defining and accessing a list of defects. The project implements a "defect_settings" class that contains the exact set of defects for a given block or project. The verification environment code can query the state of a defect to take specific actions. Users can also put their defect dependent actions directly in the defects_settings file is some cases (e.g. demote an error to a warning for a specific checker), which is the recommended way for creating an easy, common location for auditing the work around behavior.

B. Defect info base class

Figure 1 below shows the Defect info base class. It contains field values for each defect, along with accessor methods. The Defect info base class is project independent. The main purpose of this class is to capture information about the defect (subset of content in the real bug tracking system) and provide an API for other components in the system to get information about the defect. Small shell scripts (or a DPI layer to the official bug tracking system) may be added ensure the defect table state is up-to-date with official tracking system (e.g. a bug has been resolved but the work around is still be applied because the defect table is still marked as "open").

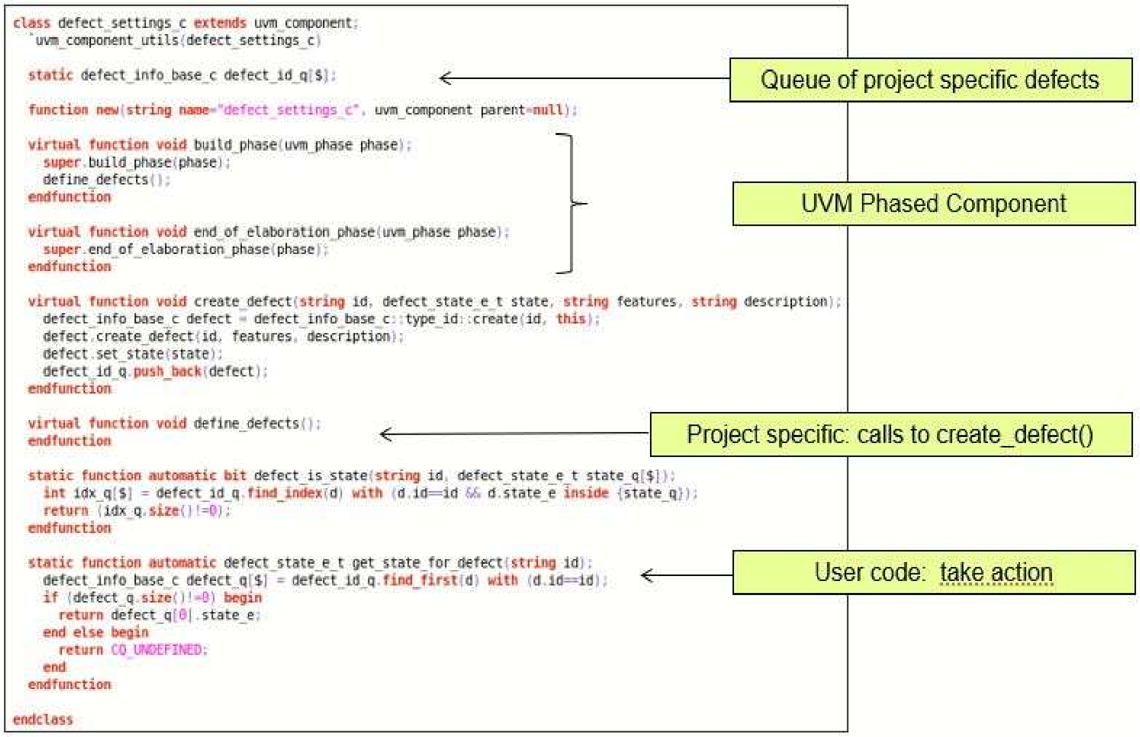


```
typedef enum {CQ_NEW, CQ_OPENED, CQ_RESOLVED, CQ_CLOSED, CQ_UNDEFINED} defect_state_e_t;

class defect_info_base_c extends uvm_component;
  `uvm_component_utils(defect_info_base_c)

  defect_state_e_t  state_e;
  string            id;
  string            summary;

  function new(string name="defect_info_base_c", uvm_component parent=null);

  virtual function void             create_defect(string id, string features, string summary);

  virtual function void             set_state(defect_state_e_t state_e);
  virtual function defect_state_e_t get_state();

  virtual function void             set_features(string features);
  virtual function string           get_features();

endclass
```

Figure 1. Defect info base class

C. *Defect settings base class*

Figure 2 below shows the defect settings base class, used to define project specifics defects, and actions. This class and methods will be used to construct the actual defect table for the block/project.

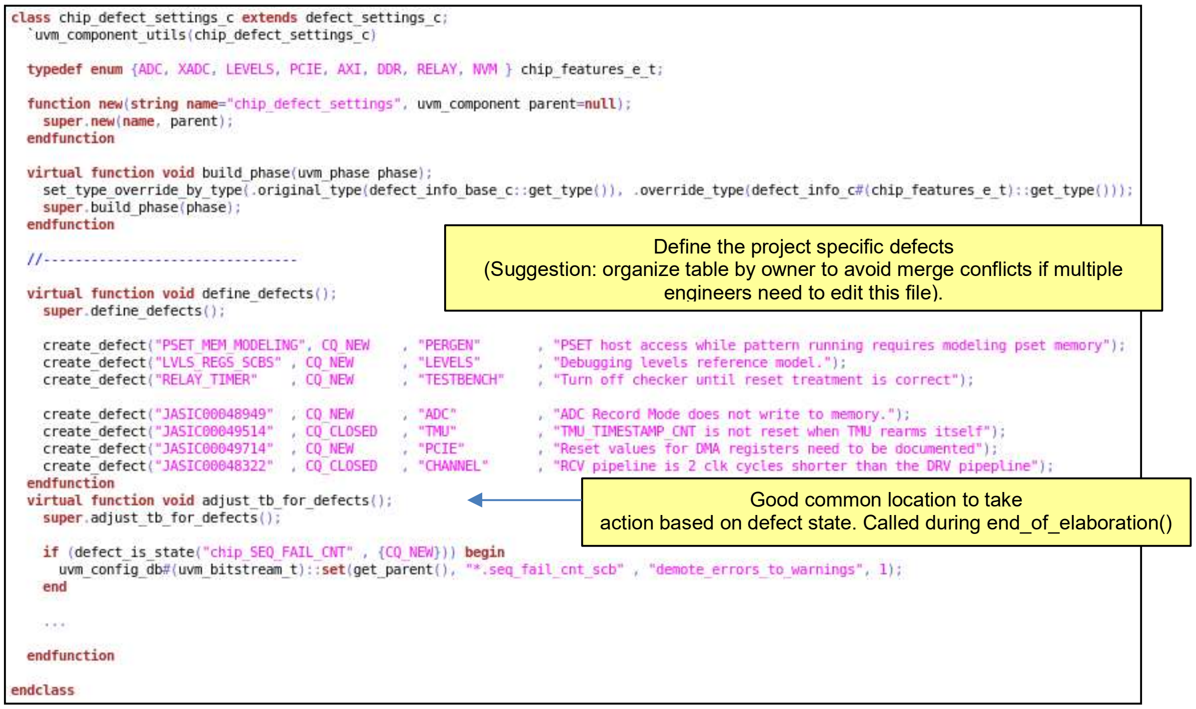

Queue of project specific defects
UVM Phased Component
Project specific: calls to create_defect()
User code: take action

```
class defect_settings_c extends uvm_component;
  `uvm_component_utils(defect_settings_c)

  static defect_info_base_c defect_id_q[$];

  function new(string name="defect_settings_c", uvm_component parent=null);

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    define_defects();
  endfunction

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
  endfunction

  virtual function void create_defect(string id, defect_state_e_t state, string features, string description);
    defect_info_base_c defect = defect_info_base_c::type_id::create(id, this);
    defect.create_defect(id, features, description);
    defect.set_state(state);
    defect_id_q.push_back(defect);
  endfunction

  virtual function void define_defects();
  endfunction

  static function automatic bit defect_is_state(string id, defect_state_e_t state_q[$]);
    int idx_q[$] = defect_id_q.find_index(d) with (d.id==id && d.state_e inside {state_q});
    return (idx_q.size()!=0);
  endfunction

  static function automatic defect_state_e_t get_state_for_defect(string id);
    defect_info_base_c defect_q[$] = defect_id_q.find_first(d) with (d.id==id);
    if (defect_q.size()!=0) begin
      return defect_q[0].state_e;
    end else begin
      return CQ_UNDEFINED;
    end
  endfunction

endclass
```

Figure 2. Defect settings base class

D. *Projects specific defects list of action*

Figure 3 below shows an example of an actual defect table for a project. The example shows four defects that are also tracked in the official bug tracking system. They are denoted with the name "JASIC000xxxx". The state (CQ_NEW/CQ_CLOSED) indicates whether the issue is considered resolved in this specific build of the environment and will impact how the verification environment behaves. If an issue does not affect how the verification environment behaves, it should only be tracked in the official bug tracking system. This defect table also shows three text labelled entries (PSET_MEM_MODELING, LVLS_REGS_SCBS, RELAY_TIMER).



```
class chip_defect_settings_c extends defect_settings_c;
  `uvm_component_utils(chip_defect_settings_c)

  typedef enum {ADC, XADC, LEVELS, PCIE, AXI, DDR, RELAY, NVM } chip_features_e_t;

  function new(string name="chip_defect_settings", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    set_type_override_by_type(.original_type(defect_info_base_c::get_type()), .override_type(defect_info_c#(chip_features_e_t)::get_type()));
    super.build_phase(phase);
  endfunction

  //-------------------------------

  virtual function void define_defects();
    super.define_defects();

    create_defect("PSET_MEM_MODELING", CQ_NEW    , "PERGEN"      , "PSET host access while pattern running requires modeling pset memory");
    create_defect("LVLS_REGS_SCBS" , CQ_NEW      , "LEVELS"      , "Debugging levels reference model.");
    create_defect("RELAY_TIMER"    , CQ_NEW      , "TESTBENCH"   , "Turn off checker until reset treatment is correct");

    create_defect("JASIC00048949"  , CQ_NEW      , "ADC"         , "ADC Record Mode does not write to memory.");
    create_defect("JASIC00049514"  , CQ_CLOSED   , "TMU"         , "TMU_TIMESTAMP_CNT is not reset when TMU rearms itself");
    create_defect("JASIC00049714"  , CQ_NEW      , "PCIE"        , "Reset values for DMA registers need to be documented");
    create_defect("JASIC00048322"  , CQ_CLOSED   , "CHANNEL"     , "RCV pipeline is 2 clk cycles shorter than the DRV pipepline");
  endfunction
  virtual function void adjust_tb_for_defects();
    super.adjust_tb_for_defects();

    if (defect_is_state("chip_SEQ_FAIL_CNT" , {CQ_NEW})) begin
      uvm_config_db#(uvm_bitstream_t)::set(get_parent(), "*.seq_fail_cnt_scb" , "demote_errors_to_warnings", 1);
    end

    ...

  endfunction

endclass
```

Define the project specific defects
(Suggestion: organize table by owner to avoid merge conflicts if multiple engineers need to edit this file).

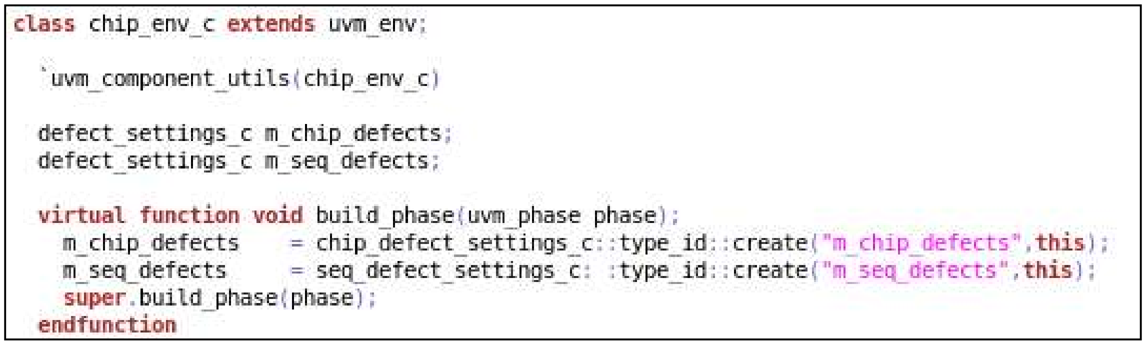
Good common location to take
action based on defect state. Called during end_of_elaboration()

Figure 3. Project specific lists of actions

The text labelled entries are issues that are not officially tracked in the bug tracking system. This allows verification engineers with an easy way to alter the behavior of the verification environment without having to track every change with an official ticket providing an easy, low-cost way, but trackable way to release new verification code that is not ready for 100% production/regression use.

E. *Constructing the defect tables*

Figure 4 below shows the how to create defect tables. Defect tables are UVM components. They can be created during the uvm_env::build() phase. The defect tables from IP blocks can also be built enabling work arounds for multiple blocks under development.



```
class chip_env_c extends uvm_env;

  `uvm_component_utils(chip_env_c)

  defect_settings_c m_chip_defects;
  defect_settings_c m_seq_defects;

  virtual function void build_phase(uvm_phase phase);
    m_chip_defects     = chip_defect_settings_c::type_id::create("m_chip_defects",this);
    m_seq_defects      = seq_defect_settings_c::type_id::create("m_seq_defects",this);
    super.build_phase(phase);
  endfunction
```

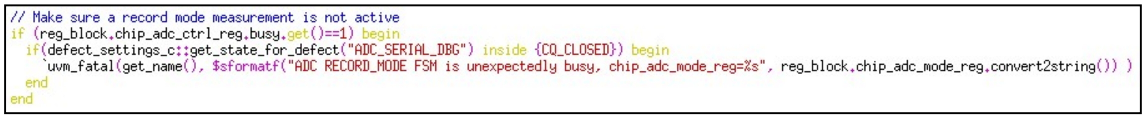
Figure 4. Creating defect tables

III. USE MODELS: CONTROLLING THE DEFECT TABLE

This section provides a few use cases for the defect table. One example is demoting a scoreboard error to a warning. Other examples including disability specific checkers. In addition, constraint randomization can also be changed based on the state of a defect to avoid certain type of stimulus

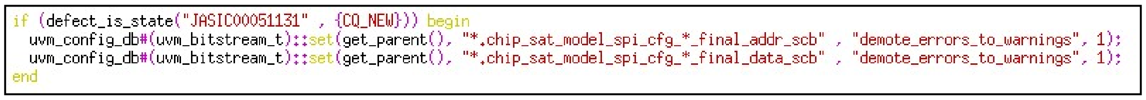
A. *Controlling error messages*

This section provides an example for controlling error messages. It assumes the environment provides hooks into the UVM messaging classes to be able to demote UVM error messages to UVM warnings based on the message ID, contents, etc. via the uvm_config_db. Figure 5 shows an example when an ADC is not allowed to be BUSY unless the defect is in the Open state. Figure 6 below shows an example where address and data errors will be demoted to warnings until a specific bug is fixed.



```
// Make sure a record mode measurement is not active
if (reg_block.chip_adc_ctrl_reg.busy.get()==1) begin
  if(defect_settings_c::get_state_for_defect("ADC_SERIAL_DBG") inside {CQ_CLOSED}) begin
    `uvm_fatal(get_name(), $sformatf("ADC RECORD_MODE FSM is unexpectedly busy, chip_adc_mode_reg=%s", reg_block.chip_adc_mode_reg.convert2string()) )
  end
end
```

Figure 5. Reg should not be set if ticket is closed

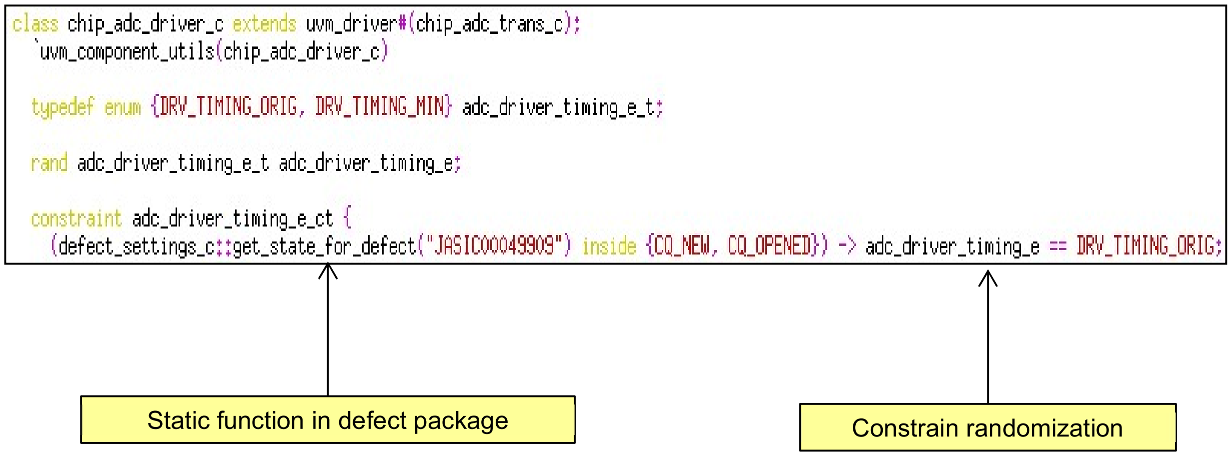


```
if (defect_is_state("JASIC00051131" , {CQ_NEW})) begin
  uvm_config_db#(uvm_bitstream_t)::set(get_parent(), "*.chip_sat_model_spi_cfg_*_final_addr_scb" , "demote_errors_to_warnings", 1);
  uvm_config_db#(uvm_bitstream_t)::set(get_parent(), "*.chip_sat_model_spi_cfg_*_final_data_scb" , "demote_errors_to_warnings", 1);
end
```

Figure 6. Disable scoreboards via UVM config_db

B. *Controlling constraints*

Figure 7 below show an example where the verification environment will not create traffic using "DRV_TIMING_MIN" until defect 49909 is closed/fixed.



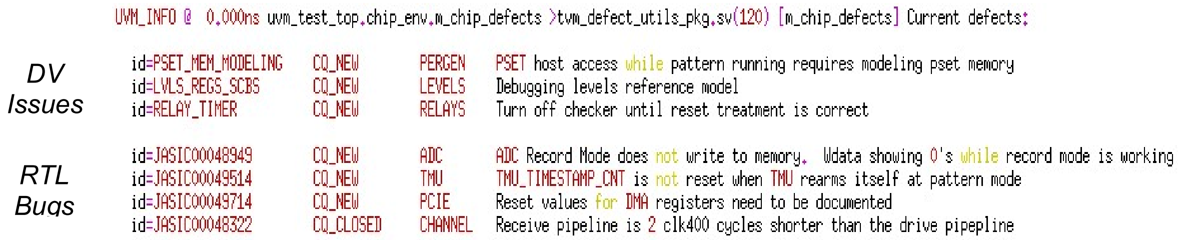
```
class chip_adc_driver_c extends uvm_driver#(chip_adc_trans_c);
  `uvm_component_utils(chip_adc_driver_c)

  typedef enum {DRV_TIMING_ORIG, DRV_TIMING_MIN} adc_driver_timing_e_t;

  rand adc_driver_timing_e_t adc_driver_timing_e;

  constraint adc_driver_timing_e_ct {
    (defect_settings_c::get_state_for_defect("JASIC00049909") inside {CQ_NEW, CQ_OPENED}) -> adc_driver_timing_e == DRV_TIMING_ORIG;
```

IV. OUTPUTS

Figure 7. Controlling constraints

Another nice feature of the defect table is the ability to dump the contents of the table to the simulation logfile. This provides engineers with the ability to know which work arounds are applied when reviewing any logfile regardless of which model it came from.

A. List of actively tracked defects

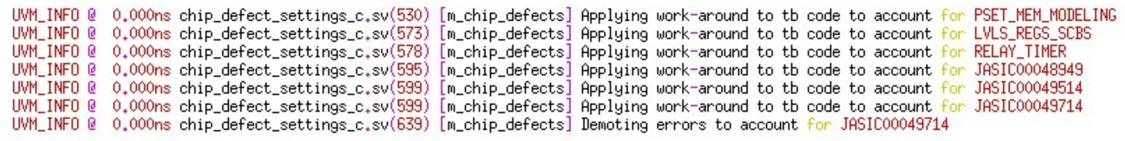


```
UVM_INFO @  0.000ns uvm_test_top.chip_env.m_chip_defects >tvm_defect_utils_pkg.sv(120) [m_chip_defects] Current defects:

  id=PSET_MEM_MODELING   CQ_NEW      PERGEN    PSET host access while pattern running requires modeling pset memory
  id=LVLS_REGS_SCBS      CQ_NEW      LEVELS    Debugging levels reference model
  id=RELAY_TIMER         CQ_NEW      RELAYS    Turn off checker until reset treatment is correct

  id=JASIC00048949       CQ_NEW      ADC       ADC Record Mode does not write to memory.  Wdata showing 0's while record mode is working
  id=JASIC00049514       CQ_NEW      TMU       TMU_TIMESTAMP_CNT is not reset when TMU rearms itself at pattern mode
  id=JASIC00049714       CQ_NEW      PCIE      Reset values for DMA registers need to be documented
  id=JASIC00048322       CQ_CLOSED   CHANNEL   Receive pipeline is 2 clk400 cycles shorter than the drive pipepline
```

Figure 8. List of actively tracked defects

B. Notice of errors demoted to warnings

In addition, the users can provide additional information about the behavior being applied based on the state of a defect. Note in the example below, the user chose to print specific information that a defect was being applied after they queried the state of the defect table and altered the behavior of the simulation.

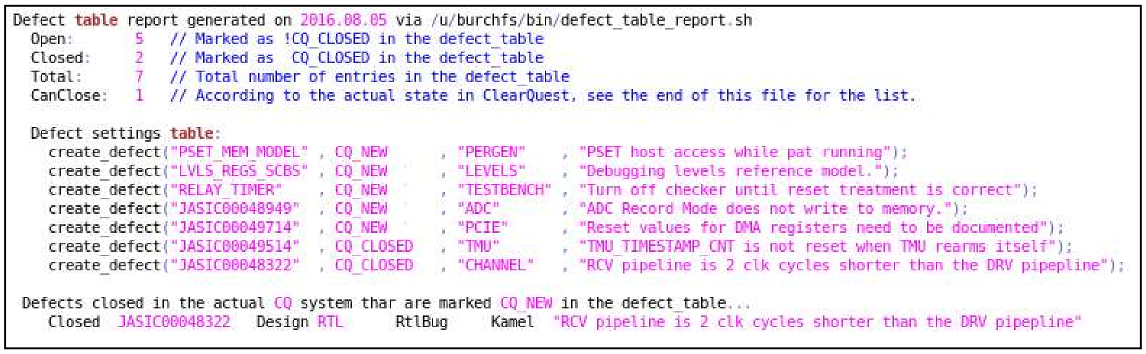


```
UVM_INFO @  0.000ns chip_defect_settings_c.sv(530) [m_chip_defects] Applying work-around to tb code to account for PSET_MEM_MODELING
UVM_INFO @  0.000ns chip_defect_settings_c.sv(573) [m_chip_defects] Applying work-around to tb code to account for LVLS_REGS_SCBS
UVM_INFO @  0.000ns chip_defect_settings_c.sv(578) [m_chip_defects] Applying work-around to tb code to account for RELAY_TIMER
UVM_INFO @  0.000ns chip_defect_settings_c.sv(595) [m_chip_defects] Applying work-around to tb code to account for JASIC00048949
UVM_INFO @  0.000ns chip_defect_settings_c.sv(599) [m_chip_defects] Applying work-around to tb code to account for JASIC00049514
UVM_INFO @  0.000ns chip_defect_settings_c.sv(599) [m_chip_defects] Applying work-around to tb code to account for JASIC00049714
UVM_INFO @  0.000ns chip_defect_settings_c.sv(639) [m_chip_defects] Demoting errors to account for JASIC00049714
```

Figure 9. List of errors demoted to warnings

C. *Automated reports*

Automated reports can be generated to compare the defects_settings.sv with the contents of the real bug tracking system. It is much more powerful, and less error prone compared to the traditional review and closure of the FIXME, TODOs, and comments inserted in the code. The contents of the external (or DPI) auditing system are beyond the scope of this paper as many options are available to enhance the tracking and reporting of a regression with respect to the defect table and official bug tracking system. For example, verification engineers could be notified when a model build contains a defect in the "OPEN" state but the officially bug tracking system moves to "RESOLVED" indicating a new model needs to be built to verify the fix (although the design engineer should have changed the defect state to verify the fix themselves!).



```
Defect table report generated on 2016.08.05 via /u/burchfs/bin/defect_table_report.sh
  Open:       5   // Marked as !CQ_CLOSED in the defect_table
  Closed:     2   // Marked as  CQ_CLOSED in the defect_table
  Total:      7   // Total number of entries in the defect_table
  CanClose:   1   // According to the actual state in ClearQuest, see the end of this file for the list.

  Defect settings table:
    create_defect("PSET_MEM_MODEL" , CQ_NEW    , "PERGEN"    , "PSET host access while pat running");
    create_defect("LVLS_REGS_SCBS" , CQ_NEW    , "LEVELS"    , "Debugging levels reference model.");
    create_defect("RELAY_TIMER"    , CQ_NEW    , "TESTBENCH" , "Turn off checker until reset treatment is correct");
    create_defect("JASIC00048949"  , CQ_NEW    , "ADC"       , "ADC Record Mode does not write to memory.");
    create_defect("JASIC00049714"  , CQ_NEW    , "PCIE"      , "Reset values for DMA registers need to be documented");
    create_defect("JASIC00049514"  , CQ_CLOSED , "TMU"       , "TMU_TIMESTAMP_CNT is not reset when TMU rearms itself");
    create_defect("JASIC00048322"  , CQ_CLOSED , "CHANNEL"   , "RCV pipeline is 2 clk cycles shorter than the DRV pipepline");

 Defects closed in the actual CQ system thar are marked CQ_NEW in the defect_table...
    Closed  JASIC00048322   Design RTL      RtlBug     Kamel  "RCV pipeline is 2 clk cycles shorter than the DRV pipepline"
```

Figure 10. Defects table report

V. RESULTS AND BENEFITS

The improvements for the execution time for projects where the defects table was used were estimated to be around 3%. The defect table provided many benefits, as listed below:

- Regressions only fail because of new bugs. Tests are kept active but specific checks/randomizations are limited.
- New features and workarounds are supported in a common manner.
- Simulation logfiles indicate which defects affect the environment.
- Automation easily indicates when an open defect entry should be closed.
- Enable the design engineers to test changes and update the defect table prior to releasing the code.
- A single location to audit workarounds is provided. There is no need to worry about commented out code or forgotten changes that limit verification.

VI. CONCLUSION AND FUTURE ENHANCEMENTS

The defect table was used to track and work around open issues to provide the design engineers and the verification engineers with a stable model and easy way to verify fixes. The defect table provided the ability to thoroughly review the complete list of workarounds and TODOs, to ensure the proper workarounds have been applied and removed prior to tape-out. Future enhancements could include:

A. Future enhancements: defect table

- Command line defects overrides
  - To verify RTL fixes without modifying the defect_settings.sv file.
  - To enable running tests for reproducing failures or testing new RTL features without code modifications.
- System call to a live defect query
  - To automatically compare defect table entries to the actual ticketing system.

B. Future enhancements: apply other lean techniques/principles to functional verification

The defect table is a particularly good example about how lean techniques and principles can be applied to improve functional verification methodologies. Below two additional examples for some other lean principles/techniques which could be easily applied to the work of the functional verification teams, and to the other cross-functional teams collaborating with them, in order to minimize waste, and to optimize the speed and value delivery.

- Value Stream Map (VSM):

  Develop a Value Stream Map, or a flowchart to manage the dependencies, the different functional verification steps, and the handoffs: the internal hand-offs within the functional verification teams, and the various external hand-offs to and from the other teams. Below, a non-exhaustive list of the benefits of VSM [2]:

  - helps identify bottlenecks and pain points
  - manages errors and defects
  - creates greater visibility and traceability throughout the entire cycle
  - eliminates redundant and wasteful processes
  - fosters cross-functional collaboration
  - reveals opportunities for automation
  - fuels faster, integrated feedback
  - provides context and process clarity with data and visuals
  - emphasizes results and KPIs
- Andons:

  The Andon cord concept in lean manufacturing [3], involves using signals and alerts to indicate that a station on the assembly line has a problem. Alarms are activated manually by workers pulling a physical cord or automatically by the equipment itself. Developed as one element of the “Jidoka” quality method and pioneered by Toyota and the Toyota production System [4], the Andon cord empower workers to notify co-workers and management of quality and process problems across the assembly line, and even stop production.

In the context of functional verification, Andons could take the form of automated email notifications for certain actions or delays once they start to negatively impact the project execution to a certain extent. To illustrate the concept with a quite simple example, one could set some threshold for the maximum acceptable delays/turnaround times (number of days for example) to fix bugs according to their priorities. Different settings could be applied to different projects. For one project for example, notifications would be sent as soon as the bugs would have been filed for showstopper/urgent bugs. Other notifications would be sent a certain number of (predefined) days after the bugs have been filed, for bugs which were not showstopper/urgent, but have been filed as high priority bugs.

REFERENCES

[1] Mary Poppendieck, Tom Poppendieck, “Lean Software Development: An Agile Toolkit”, Addison Wesley, May 08, 2003.

[2] https://www.lucidchart.com/blog/value-stream-mapping-for-devops

[3] https://devops.com/youre-not-devops-cant-pull-cord/

[4] Jeffrey K. Liker, “The Toyota Way”, Mc GrawHill, 2004