Error Injection in a Subsystem Level Constrained Random UVM Testbench

Jeremy Ridgeway[1], Hoe Nguyen[2]

Broadcom, Inc.

[1]4380 Ziegler Rd. Fort Collins, CO 80525
[2]1320 Ridder Park Dr., San Jose, CA 95131

www.broadcom.com

ABSTRACT

Error injection is a complicated procedure for any constrained random verification (CRV) environment. An ideal, if rarely possible, setup for CRV comprises a single "good" test and a single "good with error injection" test to enable maximal functional cross coverage. The verification environment must provide (a) a way to inject errors, (b) a method for detecting error injection response, and (c) a pathway to clean-up errors and recover simulation. However, we have found test benches often avoid clean-up by focusing on directed error-injection tests. These cripple functional cross coverage potential and are not indicative of real world use cases. We present our approach incorporating type-parametrized SystemVerilog classes and interface classes, a general error demotion scheme, and an interrupt service routine. We have employed this methodology in multiple subsystem-level PCI-Express verification environments.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Constrained random verification (CRV) has generally been accepted as a gateway to higher functional cross coverage and, thus, verify scenarios that are very difficult or impossible to anticipate (i.e., not in the test plan). Yet, when planning for error testing, the CRV environment is often ignored (or limited) favoring directed tests to drive test plan completeness. High value bugs can be found in both situations: good/normal operation and error detection and recovery. However, the CRV environment, itself, may be deemed too complicated to handle constrained random error injection, detection and, most importantly, recovery.

An ideal, if rarely possible, setup for CRV comprises a single "good" test and a single "good with error injection" test. While the good with error injection may be an extension of the good test, both can be individually run as a universal verification methodology (UVM) test [1]; in other words, a *UVM leaf test*. A UVM leaf test, itself, must not inject and manage the error scenario as a whole. Doing so requires that specific test to be regressed in order to cover that specific error scenario. This cripples functional cross coverage with error injection and is not indicative of real-world use models. Ideally, the error injection leaf test randomly selects a limited number of error injection scenarios (e.g., one or two simultaneously), injects the errors at random delay, and waits for recovery. Once recovery is achieved, the error injection test may randomly select another error scenario. All the while, good/normal operation is active as constrained random scenarios in the background. It *should* become untenable to test all potential functional cross coverage, good and error (otherwise directed testing is sufficient for that design). As such, error scenario testing should be crossed with as many kinds of good operation scenarios as possible—in other words, constrained random verification.

In this paper, we propose an error injection architecture for a CRV environment that provides:

1) A method for error injection,
2) A method for detecting the error injection response via test bench reported errors and device under test (DUT) interrupt behavior, and
3) A method for error clean-up and recovery.

We employ a collection of UVM extended objects and components with both type-parameterization and interface classes [2, 3]. We define an error injection service (EIS) object, in section 2. , as the overall manager for the error scenario. This object instruments the environment for error injection, monitors the environment for error detection, and coordinates recovery. The EIS object is the manager for a specific error scenario. Furthermore, the EIS object is randomly selected and started by the "good with error injection" UVM leaf test, thereby enabling background "good" operation crossed with error injection. Section 3. describes our error demotion scheme and its coordination with the EIS object. Our common interrupt service routine (ISR), and its coordination with the EIS object, is presented in section 4. We tie all the pieces of our error injection architecture together in section 5. by describing the "good with error injection" UVM leaf test along with a discussion of UVM phasing, section 6. Finally conclusions are discussed in section 7.

2. Error Injection Services (EIS) Object

A single error injection object manages the error injection, error reporting, and recovery. The remainder of the verification environment should be implemented, initially, to focus only on "good" testing scenarios. The bulk of regression should cover cross-testing good scenarios, while only a subset should focus directly on cross-testing good and error scenarios. For example, in our PCI-Express subsystem verification environments we inject an error in the cyclic-redundancy check

(CRC) field in a data packet [4]. Normal test bench operation calculates this value correctly at packet randomization in some UVM verification component (UVC) stimulus generation sequence while a scoreboard verifies that the transmitted packet received by the DUT matches, as in Figure 1. In the event of an error in the CRC field the test bench should expect the packet to be dropped and an error flagged.

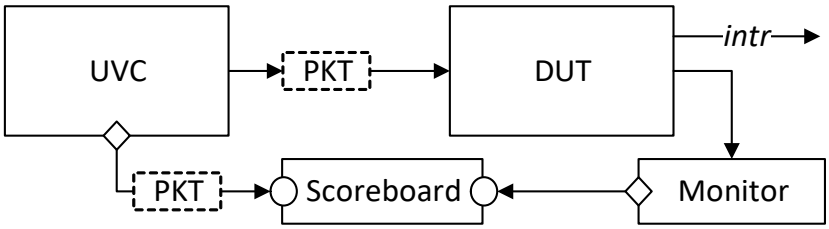

Figure 1: Sample test bench transmitting good packet to DUT (Diamonds = Analysis Ports, Circles = Implementation Ports).

Initial test bench implementation should only focus on the packet generation, the DUT receiving the packet, and the packet being verified in the scoreboard. During error scenario implementation some components should be updated, Figure 2. First, the monitor in the UVC should be updated to check for packet errors.[1] If an error exists in the packet then the checker should report the error and drop the packet instead of publishing to the analysis port, Figure 2-1. Second, a callback mechanism, Figure 2-2, should be added to the driver to allow some external object to modify the contents of the packet, seeding the condition that the monitor will, then, report as an error.

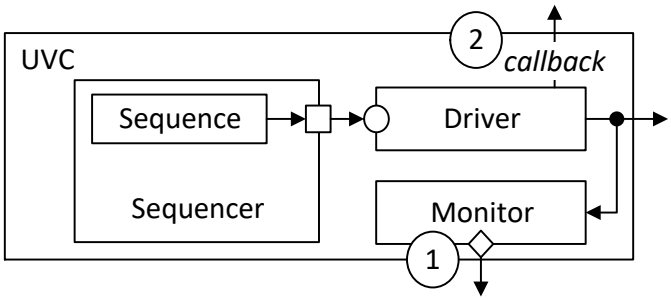

Figure 2: Sample UVC, from Figure 1, with updates required for error injection support.

An external object can both inject the error and check for DUT response. In Figure 1, the response for the DUT is to report the CRC error in an interrupt register and assert the interrupt output. A centralized error injection services (EIS) object can manage all steps for this error injection: from injection via callback to handling the interrupt and clearing the expected interrupt register field. The EIS object begins the error testing scenario by starting with error injection.

2.1 EIS Object Hierarchy

Error injection, depending on the injection point in the verification environment, may require differing kinds of class extension. For example, in Figure 2, the callback is the error injection point. Ideally, a common set of fully formed error injection services (EIS) methods could be encapsulated in a class and extended alongside the callback class, as shown in Figure 3 (A). However,

[1] The UVC should be checking for errors anyway to support uvm_agent passive mode for bus checking.

SystemVerilog does not support this C++-style of multiple class inheritance favoring the Java-style interface class implementation [2, 5], Figure 3 (B). This becomes a maintenance nightmare (how to add features to all EIS methods in all files?) and architecturally does not fit with the "good with error injection" common base test to randomly select, instantiate, and execute error injection.

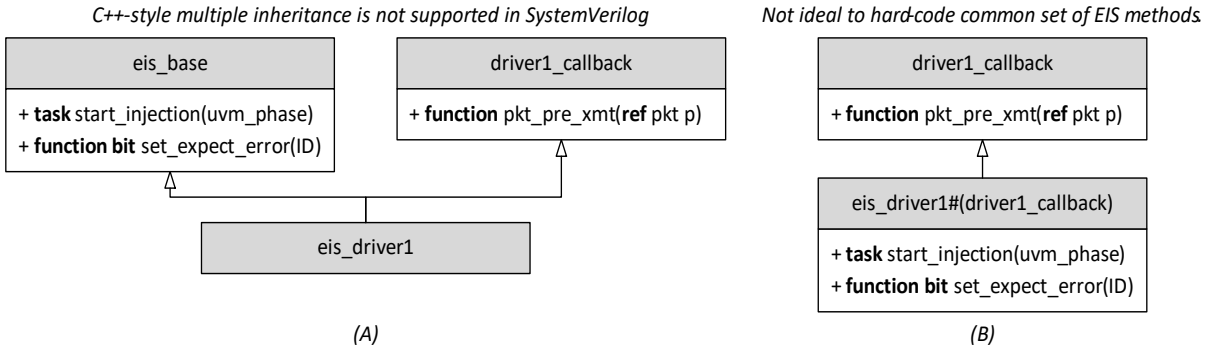

Figure 3: Extension of base set of functions AND injection class, (A), is not supported in SystemVerilog. Harding coding a set of methods, as in (B), is not ideal either: each leaf EIS class would have its own implementation of common functionality.

Our error injection services methods opaquely extend from the error injection class type and *implement* a common programming interface (PIF), Figure 4, with a SystemVerilog interface class. We have split the common error services methods in the eis_base class from Figure 3 (A) into two classes in Figure 4: the opaque type-parameterized eis_base class and the interface eis_pif class. A SystemVerilog interface class may only contain pure virtual methods [2]. These methods may be used by other classes both as a reference (i.e., access an instance of eis_base via an eis_pif reference) and a known set of methods, a programming interface. We capitalize on these features in our error-injection enabled UVM base test to manage an array of active eis_pif references. When an error injection is selected and instantiated, the base test starts injection by executing eis_pif.start_injection(phase).

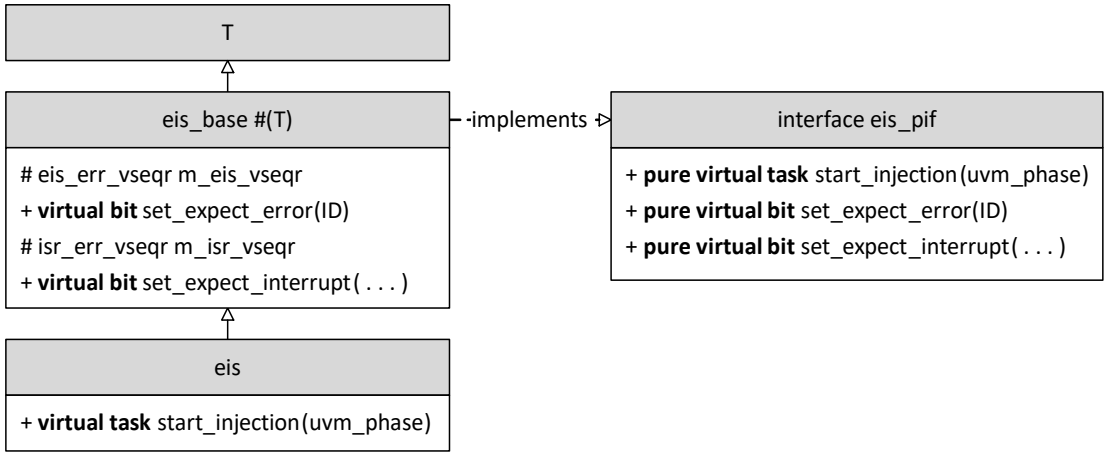

Figure 4: Error injection services (EIS) base object extending from opaque type.

The leaf eis class in Figure 4 implements the start_injection() task and calls the T classes' methods to perform the injection. An example for the UVC driver callback implementation as an EIS object is in Figure 5. When selected by the error injection enabled base test, the eis_driver1 class'

start_injection() task sets up for callback from the UVC driver and, when its pkt_pre_xmt() function is executed, injects the CRC error into the outgoing packet. Additionally, the eis_driver1.start_injection() task sets up to expect errors that the UVC monitor will report when it detects a bad CRC in an outgoing packet. The eis_base class implements the required functions to register expected error reports. We present catching expected UVM error reports in section 3. Finally, the DUT may report an interrupt as a result of a bad packet received. The eis_driver1.start_injection() task also instruments the verification environment to react to the interrupt. An existing common interrupt service routine (ISR) is modified by the eis_driver1 class through eis_base class functions to both expect an interrupt and handle it appropriately. By default, the ISR will clear the interrupt register field corresponding to the bad packet. We present ISR handling in section 4.

Multiple mutually exclusive error injection classes may operate simultaneously in the verification environment to maximize cross-error scenario coverage. Because a common programming interface is available, the error injection enabled base test can start multiple EIS objects. While multiple EIS objects can safely handle the same error report, only one EIS object may handle one interrupt service routine register field at a time: a one-to-one relationship. Therefore, as long as each EIS object handles mutually exclusive interrupt register fields, multiple EIS objects may exist and operate simultaneously. We tie this action together in section 5.

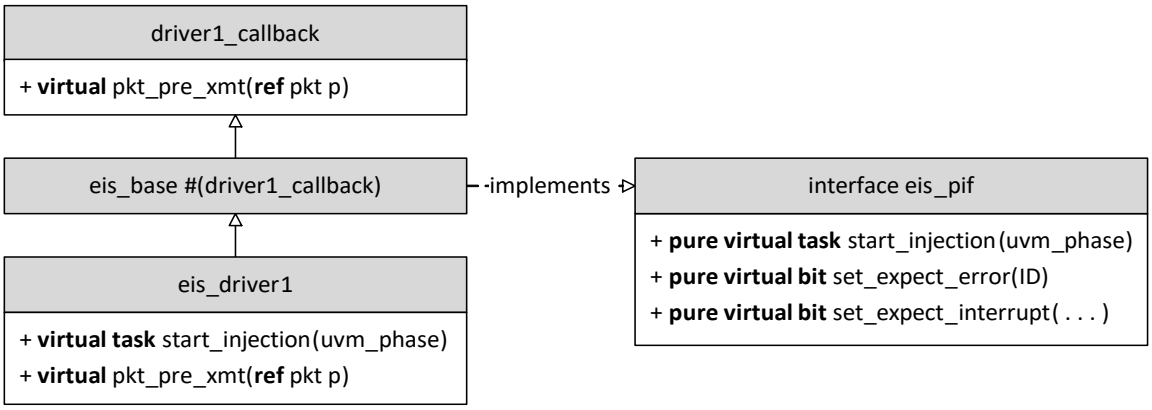

Figure 5: EIS example for implementing CRC error at UVC driver callback error injection point.

2.1.1 EIS Programming Interface Class

The full programming interface class is provided below as a preview to the remainder of this paper. All tasks and functions in the interface class must be abstract (pure virtual). We have noted, referring to Figure 4 for class hierarchy, which methods are implemented in the eis_base#(T) class and which are expected to be implemented in the leaf EIS class.

```
1   interface class eis_pif;
2     // Methods for compliancy with UVM object, forces eis_base#(T) to be UVM
3     pure virtual function string get_name();
4     pure virtual function string get_full_name();
5     pure virtual function string get_type_name();
6
7     // Task always required to be implemented in EIS leaf class
8     pure virtual task start_injection(uvm_phase phase);
9
```

```
10    // API for callbacks
11    //    - basic handling implemented in eis_base
12    //    - optionally implemented in EIS leaf class
13    pure virtual function int error_active(string err_key,
14                                  ref pss_err_eis_err_info_t err_info);
15    pure virtual task interrupt_active(pss_err_isr_vseq_base intr_vseq,
16                                  output int handled_code);
17
18    // API for setup/teardown of expected errors
19    //    - implemented in eis_base
20    //    - used by the EIS leaf class
21    pure virtual function bit set_expect_error(string err_key,
22                                  int err_num = -1, // -1 = always
23                                  int err_verbosity = UVM_MEDIUM,
24                                  int rpt_verbosity = UVM_MEDIUM);
25    pure virtual function bit clear_expect_error(string err_key,
26                                  int rpt_verbosity = UVM_MEDIUM);
27    pure virtual bit clear_expect_errors(int rpt_verbosity = UVM_MEDIUM);
28
29    // API for setup/teardown of expected interrupts
30    //    - implemented in eis_base
31    //    - used by the EIS leaf class
32    pure virtual function bit set_expect_interrupt(
33                                  pss_err_isr_vseq_base intr_vseq,
34                                  int intr_key = (-1));
35                                  // map to a local unique KEY, -1  = no key
36    pure virtual function bit clear_expect_interrupts(
37                                  bit clear_intr_key = 1);
38
39    // API for mapping ISR virtual sequence references to local KEY
40    //    - implemented in eis_base
41    //    - used by the EIS leaf class
42    pure virtual function bit set_expect_interrupt_key(
43                                  pss_err_isr_vseq_base intr_vseq,
44                                  int intr_key);
45    pure virtual function int get_expect_interrupt_key(
46                                  pss_err_isr_vseq_base intr_vseq);
47    pure virtual function bit clear_expect_interrupt_key(
48                                  pss_err_isr_vseq_base intr_vseq);
49
50  endclass
```

Lines 3-5 imply a dependency on the UVM object class (or class extensions). For most cases, this is no problem as the all user classes from the UVM library are extensions to the uvm_object class. Furthermore, all factory-compatible UVM class extensions generally are expected to have the get*_name(), utility functions available. However, get_type_name() is defined via UVM class utility macros (`uvm_object_utils, `uvm_component_utils, etc.). As such, care must be taken when the type parameter class T is *not* UVM factory-compatible. Simply, the verification engineer must also define these methods in class T when used in an EIS object (eis_base#(T) extends T).

In line 8, the start_injection task is indicated. This task is *not* implemented in eis_base#(T). Therefore, the engineer implementing the leaf EIS class must implement it. Here, the task is meant to declare a common programming interface that any object may call to start error injection. In section 6. , we indicate how we use this task in a test_base class that selects erorr injections based on enumerated type. Once selected, the error injection is initiated by calling start_injection().

Lines 13-16 are the main callback methods to the EIS object. While the eis_base#(T) class provide rudimentary functionality here – basically flagging errors and maintaining automatic demotions – the user may opt to extend the functionality for more specific control.

Lines 21-27 are the methods for error expection setup and clearing. As described in section 3. , the string error ID may be indicated as an exact match upon report message callback, or as a light-glob pattern. When an error message report is caught, the error_active() function is called. For example, consider an error injection that may cause a PCIe link down event. The Synopsys PCIe VIP will flag phy-layer errors that, in this test, are expected [6].

```
task myeis_leaf::start_injection(…);
  setup_expect_error("MYERR1", 1); // expect 1 error
  setup_expect_error("MYERR2", 2); // expect 2 errors
  setup_expect_error("register_fail:ACTIVE_PL:*"); // always
endtask
```

Without any additional implementation, the eis_base#(T)::error_active() function will expect and demote to a UVM_INFO message two instances of MYERR2 and one instance of MYERR1, as described in section 3.3 . If further MYERR1 or MYERR2 error reports are made, they are reported as actual errors (barring demotion described in section 3.2). Additionally, if the expected error count is not reached then this EIS object may report an error itself (via the virtual sequencer connecting the error demoter with the EIS objects). However, all "register_fail:ACTIVE_PL*" messages are both always expected and are outside of expect count checking. Here, the erorr stays active until the end of error injection and/or clear_expect_error() or clear_expect_errors() is called.

Lines 32-37 handle EIS object manipulation of the interrupt service routine, as described in section 4. The ISR is a collection of virtual sequence references maintained in a central table. These functions modify the structure of that table during EIS object managed error injection. When an interrupted is seen during error injection, the interrupt_active() function is called.

It is difficult to identify which interrupt virtual sequence calls back the EIS object when more than one has been setup. As such, the eis_base#(T) class handles a simple mapping scheme from a unique int value. Consider two interrupt handling sequences are created at start injection:

```
task myeis_leaf::start_injection(…);
  isr_field_link2eis_vseq intr1 = new(…);
  isr_field_link2eis_vseq intr2 = new(…);
  setup_expect_interrupt(intr1);
  setup_expect_interrupt(intr2);
endtask
```

Now, upon callback when one of these interrupts is asserted the EIS object would have to compare the reference with known references:

```
task myeis_leaf::interrupt_active(isr_field_link2eis_vseq intr_vseq, …);
  if(intr1 == intr_vseq) …
  else if(intr2 == intr_vseq) …
endtask
```

Alternatively, the leaf EIS object may maintain an associative array mapping the virtual sequence reference to a unique key and vice versa. This is exactly the functionality implemented in the eis_base#(T) class referred to in lines 42-48 above.

```
task myeis_leaf::start_injection(…);
  isr_field_link2eis_vseq intr1 = new(…);
  isr_field_link2eis_vseq intr2 = new(…);
```

```
    setup_expect_interrupt(intr1, .key(1));
    setup_expect_interrupt(intr2, .key(2));
  endtask
```

The unique key supplied at setup_expect_interrupt() allows for a more efficient callback.

```
  task myeis_leaf::interrupt_active(isr_field_link2eis_vseq intr_vseq, …);
    case(get_expect_interrupt_key(intr_vseq))
      1: // interrupt 1 was asserted, handle
      2: // interrupt 2 was asserted, handle
      default: `uvm_fatal(ID, "Unknown interrupt active")
    endcase
  endtask
```

2.2 EIS Virtual Sequencer

There may be additional components to access, sequences to start, or other work the EIS object is required to perform to actually inject the error and/or respond appropriately. The EIS virtual sequencer provides two essential functions in this framework. First, it ties the error demotion function, described in section 3.2 with multiple EIS instantiated objects. For each error report generated from the environment the EIS virtual sequencer passes control to each EIS object expecting that error report through the error_active() eis_pif class reference virtual function. In Figure 6, the eis_err_vseqr class maintains a queue of eis_pif references for error report string IDs to those EIS objects that requested notification (via connect_error()). When an error report message is detected from the UVM report server the eis_err_vseqr.error_active() function is executed. In turn, for each EIS object associated with that error ID, the virtual sequencer executes the error_active() virtual function in the EIS object instance. The eis_pif interface class allows for the EIS virtual sequencer to maintain a table of EIS objects without knowing the type-parameters or leaf class type (eis_driver1 in the figure).

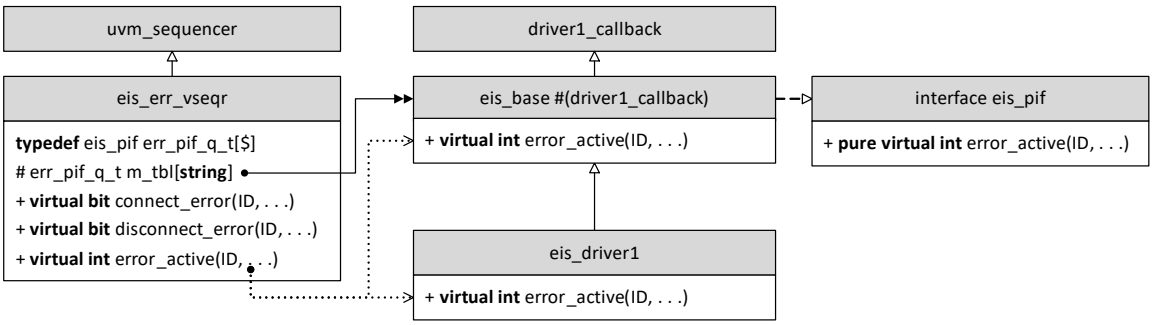

Figure 6: EIS virtual sequencer connects from the UVM report server to pertinent EIS object instances via error_active().

Second, the virtual sequencer provides sequencer functions for virtual sequences—sequences not requiring UVM TLM ports or a connected driver. That is, the EIS object can instantiate and start a virtual sequence on its virtual sequencer at will (we make the assumption that few sequences will be started simultaneously due to the increasing debug complexity).

3. Catching Reported Errors

In this section we describe two approaches to catch and demote reported errors. The first is intended to demote errors in any simulation from any UVM-enabled reporting object. The second directs caught error reports to the EIS object for randomized error injection services.

3.1 UVM Error Report Dissection

Any UVM enabled object or component in the test bench may report an error using the `uvm_error macro or uvm_report_error() reporting functions (with and without component context) [1]. Consider the following report error in UVM test bench code, in a uvm_env class extension instantiated inside the UVM leaf test:

```
`uvm_error("MYERR", "Error seen.")
```

This report message may be broken into two items that the verification engineer can control during coding:

1) Reporting ID: MYERR; and
2) Message to report: Error seen.

In simulation, the following report message is generated:

```
UVM_ERROR test_env.sv(46) @ 0.0ns: uvm_test_top.ENV [MYERR] Error seen.
```
[2]

The general, the full UVM message error report (when all message components are available, as in the example above), is as follows:

```
UVM_ERROR filepath(lineno) @ timestamp: testbenchpath [KEY] message
```

a) Message type: UVM_INFO, UVM_WARN, UVM_ERROR, or UVM_FATAL;
b) Originating file path, when available;
c) Originating file line number, when available;
d) Simulation time stamp;
e) Simulation test bench path, when available;
f) Reporting KEY; and
g) Message to report.

The file path and line number may not be available if the UVM report macro is not used or if the built-in SystemVerilog `__FILE__ and `__LINE__ macros are not used with UVM report functions. The test bench path may not be available when reporting from the global UVM report function or from any instantiated UVM object. In these cases often only "reporter" is provided for test bench path. Some sequences may report using a test bench path of "reporter@@sequence_instance_path_name" which can include a hierarchy of sequences, but generally not the instantiating component.[3] All UVM components, by default, include instantiated test bench path.

3.2 Catching and Demoting UVM Error Reports

The built-in UVM report server employs UVM callbacks in the guise of an uvm_report_catcher class [1]. The UVM report server's report() function processes report messages in the following priority order:

[2] We have filtered out the full file path, only reporting the file name, for brevity in this paper.
[3] Interestingly, the log does actually use the double @ in its reporter signature

1) Is the report enabled: uvm_report_enabled(verbosity_level, severity, id)?
2) Is the report actionable: get_action(severity, id) != UVM_NO_ACTION?
3) Execute uvm_report_handlers; then is it still OK to report?
4) Execute uvm_report_catchers; then is it still OK to report?
5) Compose report message and report.

We have implemented our general purpose error demotion utilizing the uvm_report_catcher in step (4), above. It is possible that step (4) may never be executed for a specific error report due to special handling via UVM class extension. However, the UVM uvm_report_handler reference entry indicates that it is "not intended for direct use" [1]. As such, we have made the assumption that for UVM error reports likely steps (1), (2), and (3) will not block execution of step (4) and, therefore, we *can* catch all test bench reported errors. There is also the likelihood that, depending on ordering of the processed report catchers in step (4), a specific UVM report error may be caught and demoted *prior* to our general purpose demotion utility to catch and demote the same error report. We may mitigate this possibly, but not guarantee, by indicating prepend during uvm_report_catcher registration.

```
uvm_report_cb::add(.obj(null), .cb(this), .ordering(UVM_PREPEND));
```

We indicate that this report catcher (.cb(this)) affects all objects (.obj(null)) and that our report catcher should be processed first in the queue of report catchers (.ordering(UVM_PREPEND)). However, report catchers may be registered and deregistered dynamically and from anywhere in the test bench. We have an internal guideline not to override the general purpose demotion catcher, but there may be specific instances where this is necessary. Therefore, we cannot guarantee that our general purpose catcher will catch all home grown test bench error reports. While error reports originating from third-party verification IP (VIP) will likely be processed by our catcher we cannot guarantee they will preserve report catcher ordering. Thus, we cannot guarantee that our general purpose report catcher will catch all VIP error reports, either. We can only consider our general purpose catcher to be the *primary* report catcher and error demotion utility in the verification environment.[4] In our experience, so far, we have not hit any of these issues and we have caught all reported errors.

[4] There may be more than one UVM report server in the environment through class extension, but this does not affect catching and demoting error reports, only the composition and display of the report.

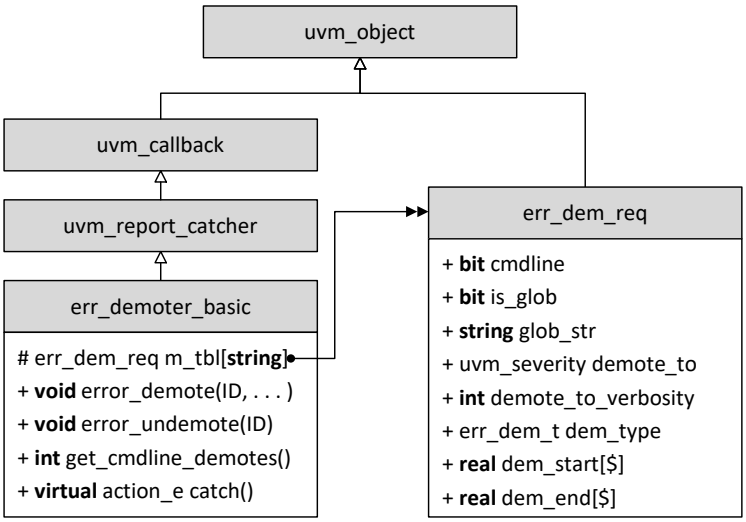

Figure 7: UVM error report catcher class inheritance hierarchy.

Our general purpose UVM error report catcher is shown in Figure 7. There exists a protected associated array, m_tbl, keyed by string that maps to an instance of the err_dem_req class. Considering the complexity of the reported message itself, possibly containing dynamic information, we have opted to key our error demotion by error message ID. The error ID is the string ID in the `uvm_error("ID", "message") report. Error IDs for demotion may be specified as either an exact match or via a light-globbing algorithm. Either way, unique reported IDs, used in the `uvm_error() macro, are necessary for high fidelity demotion.

Errors are registered for demotion in two ways. First, during simulation error IDs may be registered and deregistered via public interface functions error_demote() and error_undemote(), respectively. Additionally, error IDs may be specified on the command-line via plusarg: +DEMOTE=ID. Error IDs may be specified as a simple string, for exact match, or following a light globbing scheme:

KEY	Match KEY exactly
KEY*	Must start with KEY
KEY$	Must end with KEY
K?Y	Wildard character match
K?Y* or K?Y$	May mix ? and * OR ? and $ (mutually exclusive)

We translate this into a match-able pattern (for example, "KEY*" is translated to "^KEY") for use during callback with our match_string() function, see code below. Demoted light-glob strings may be undemoted by providing the same light-glob string. For example, from Figure 7, the error_demote(ID, …) provided is hashed as-is in err_demoter_basic.m_tbl[ID]. The translated ID, suitable for the match_string() function is stored within the request at err_dem_req.glob_str for comparison during callback. Both ID types, exact match and light-glob pattern, may be registered in simulation or on the command-line (though shell-escape for special characters may be required).

We provide three types of demotions: always, demote for a specific count, and demote for a specific simulation time window. The start and end real queues in the err_dem_req class, Figure 7, allow for multiple time windows over successive error_demote() calls. All error demotion types may be registered in simulation or on the command-line.

```
1   // name "abcdefgh" matches pattern "^abc"
2   // name "abcdefgh" matches pattern "fgh$"
3   // name "abcdefgh" matches pattern "*" (and every other string)
4   // name "abcdefgh" matches pattern "a??def?h"
5   function bit match_string(input string name, input string pattern);[5]
6     string sname, spat; int namelen, patlen;
7
8     if(pattern == "*") return 1; // always match
9
10    sname = name; namelen = name.len();
11    spat = pattern; patlen = pattern.len();
12
13    // check for ^ at start - take prefix only
14    if(pattern.substr(0, 0) == "^") begin
15      spat = spat.substr(1, (patlen - 1));
16      if(namelen > (patlen - 2))
17        sname = sname.substr(0, (patlen - 2));
18      patlen = spat.len(); namelen = sname.len(); // update lengths
19    end
20
21    // check for # at end - take suffix only
22    if(spat.substr((patlen - 1),(patlen - 1)) == "$") begin
23      spat = spat.substr(0, patlen - 2);
24      if((namelen + 1 - patlen) > 0)
25        sname = sname.substr((namelen - patlen + 1), (namelen - 1));
26      patlen = spat.len(); namelen = sname.len(); // update lengths
27    end
28
29    if(namelen != patlen) return 0; // match strings are different
30
31    // align the ? chars in both string
32    for(int i = 0; i < patlen; ++i)
33      if(spat.substr(i, i) == "?") sname.putc(i, "?");
34
35    return (sname == spat);
36  endfunction
```

During error report message processing in the UVM report server, our UVM catcher is entered via the catch() function. For any UVM_FATAL, UVM_ERROR, and UVM_WARNING severity we attempt to match the error report string ID with the internal table. Each key in the internal table is compared against the error report's string ID with the match_string function as shown in the code above. Every matching entry in the table (either by exact match or match_string() function result) is processed for possible demotion according to the mapped err_dem_req instance. For example, consider two demotions of type DEMOTE_BY_CNT:

```
demoter.error_demote("MYERR", DEMOTE_BY_CNT, 1); // exact match
demoter.error_demote("MY*", DEMOTE_BY_CNT, 2);   // light-glob
```

[5] Legacy Broadcom code reused here, thanks to whomever coded this first—gave me less to think about.

The resultant structure of the error demotion table is shown in Figure 8. Note that the m_tbl's string key *is* the string provided in the error_demote() function call. However, for the light-glob entry, the key has been translated from "MY*" to "^MY" to conform to the match_string() function's pattern argument.

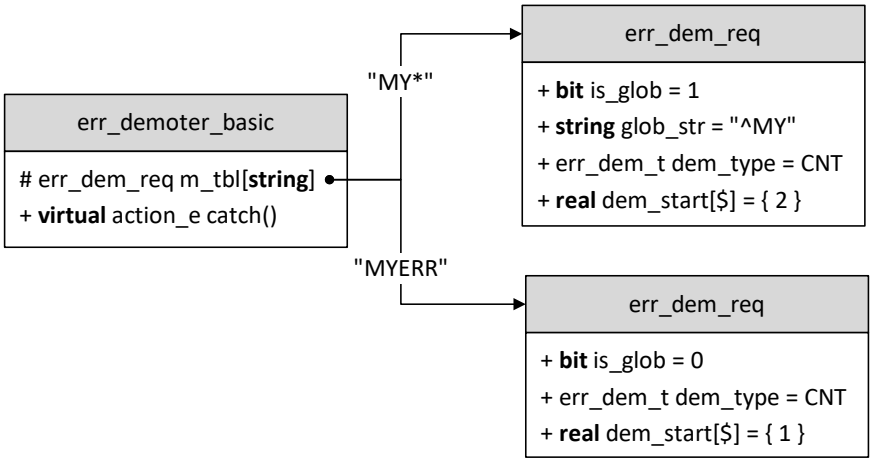

Figure 8: Light-globbing use in error report demotion. Both would match on error report with ID = "MYERR".

Now consider the report `uvm_error("MYERR", "Error seen."). During demotion processing, both entries in Figure 8 match. Each matching entry then has the opportunity to request demoting the report severity to the demote_to severity indicated in the entry, as shown in the class hierarchy in Figure 7. When the report ID matches and the entry is applicable, that is, within the bounds of the request (count > 0 for DEMOTE_BY_CNT, current time is relevant for DEMOTE_BY_TIME, or is DEMOTE_ALWAYS), the demotion takes effect. From Figure 8, both entries are applicable with "MY*" decrementing to 1 and "MYERR" decremented to 0 and removed from the table.

Between all matching entries in the demotion table, m_tbl, the lowest severity is taken (UVM_INFO in worst case), the report severity changed, and the uvm_report_catcher demote counts updated. Then, the UVM report server composes and emits the report according to the possibly modified severity. Referring back to the UVM report error example in section 3.1 , when MYERR is successfully demoted the server will emit:

```
UVM_WARNING test_env.sv(46) @ 0.0ns: uvm_test_top.ENV [MYERR] Error seen.
```

And, at the end of simulation the demotion count will include this report.

```
--- UVM Report catcher Summary ---
…
Number of demoted UVM_ERROR reports  :    1
```

3.3 Integrating the EIS Object to Error Reporting

With the general purpose error demotion UVM report catcher in place we need only to extend for expected error reports, as indicated in Figure 9. There is a difference between *demoted* and *expected* error reports. In the first case, specific errors are treated as not affecting the overall test outcome. For example, if third party VIP flags some errors during or just after reset assertion then we may indicate that these errors are not pertinent to simulation pass or failure and, thus, may be demoted. Demoted errors may occur tangentially to testing scenarios. In the second case, expected errors, logic must be implemented to flag an error if an expected error report does not occur. This is the work of the EIS object and its associated virtual sequencer.

We extend the err_demoter_basic class with a new demoter that connects to the EIS objects' virtual sequencer which, in turn, connects the caught error report to one or more EIS objects. Each EIS object may optionally demote the error report to, by default, a UVM_INFO report. A received expected error report is treated as a normal informational message while unexpected error reports are still errors. If the caught error report matches an expected error report then eis_err_demoter.catch() handles the report. Otherwise, the error report is passed to the super.err_demoter_basic.catch() function to handle.

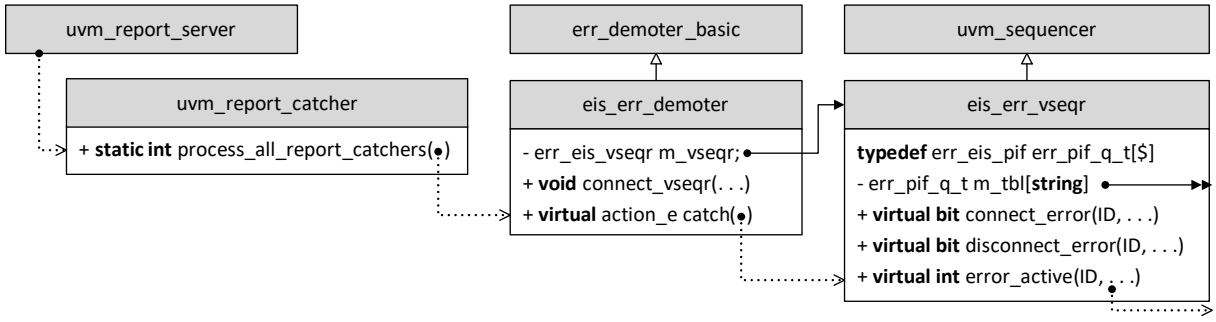

Figure 9: Error Injection Services (EIS) report error demoter extension. The UVM demoter is integrated with the EIS object via the EIS error virtual sequencer. Dotted arrows indicate execution call hierarchy.

The EIS object may register the expected string error ID using its internal set_expect_err() function, Figure 10. Similar to the UVM report error demotion scheme, the string ID provided may result in an exact match (eis_base#(T2)) or light-glob pattern match (eis_base#(T1)). Note that the type-parameter for each eis_base class extension may or may not be the same but references to both are stored in the eis_err_vseqr.m_tbl associative array via the eis_pif interface class type. In fact, the table is composed of an array of interface class references to support a one-to-many relationship between reported error and multiple simultaneous EIS objects monitoring for that error.

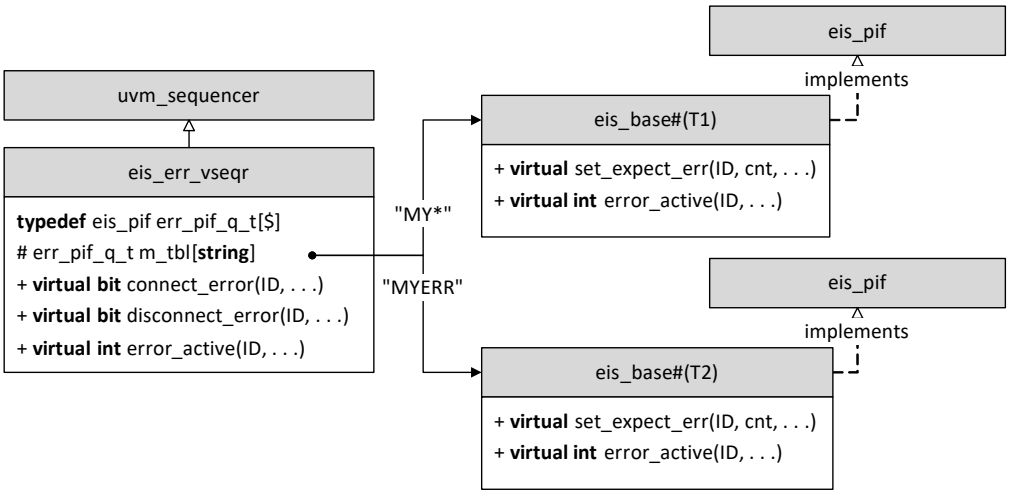

Figure 10: EIS objects connect to the eis_err_vseqr for callback via their set_expect_err() function. The string error ID may be exact match ("MYERR") or light-glob pattern ("MY*").

It may not be sufficient to simply match on the expected string error ID from the reported expected error message. Referring to Figure 2, consider the case where multiple instances of the same UVC exist in the simulation. When multiple instances exist, errors reported by any UVC instance are

likely of the exact same report message format. Referring to Figure 11, the string error ID reported from the UVC instance env.uvc0 is likely the same as the string error ID reported from env.uvc1.

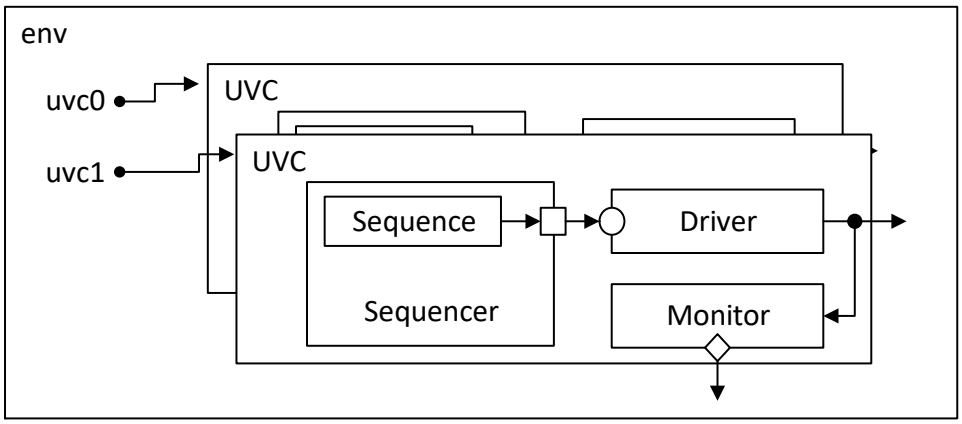

Figure 11: Multiple instances of the same UVC in the test bench.

To accommodate error injection when mulitple instances of the same UVM checker may report errors, the EIS object's error_active() function contains contextual information. For example, consider the general structure to some reported error message:

```
UVM_ERROR filepath(lineno) @ timestamp: testbenchpath [KEY] message
```

The testbenchpath, when reported from a UVM component, is assumed to be unique in the UVM test bench structure (this does not necessarily hold for UVM object reported messages). From Figure 11, the testbenchpath for some error reported from the UVC instance uvc0, should be similar to:

```
uvm_test_top.env.uvc0
```

This information is be collected and passed to each EIS object to check if (a) the error string ID matches *and* (b) the report came from the expected component. The following code illustrates this.

```
1   typedef struct {
2     string error_id;     // same as error_active(err_key,...)
3     string rpt_context; // test bench path
4     string rpt_message; // full reported message
5     int new_severity;   // what to change severity to (default: UVM_INFO)
6     int new_verbosity;  // what to change verbosity to (default: UVM_MEDIUM)
7   } eis_err_info_t;
8
9   class err_eis_base#(type T = uvm_object)
10    extends T implements err_eis_pif;
11    ...
12    virtual function int error_active(string err_key,
13                                      ref eis_err_info_t err_info);
14      ...
15    endfunction
16  endclass
```

Note that the error_active() function takes a *reference* to the eis_err_info_t structure. In this manner we achieve two points:

1) String handling tends to be expensive but passing a reference (memory pointer) is cheap,
2) Each EIS object handling the error report can see what the expected severity will be.

Because the report error execution flow passes a reference to a single eis_err_info_t instance, the full report message may also be provided to the EIS objects for inspection without significant simulation cost. Actually, this structure may even extend to include the filename and line number, if desired, though we have not (yet) needed that in simulation. The new severity and verbosity is seeded to UVM_INFO and UVM_MEDIUM, respectively, by the eis_err_demoter callback function from Figure 9. However, any EIS object may inspect and change these values as required.

With the basic error report demotion and extended EIS demotion facilities in place the EIS object may observe and optionally respond to the reported errors or classes of reported errors. For example, once the error report is observed, the EIS object may deregister callbacks to prevent further error injection. Furthermore, if the observation does *not* take place, the EIS object may report an error of its own (usually at the end of simulation) indicating the error injection failed.

4. Interrupt Service Routine

The interrupt service routine (ISR) must be both dynamic and, itself, understand how to clear flagged interrupts. That is, when an external interrupt signal is asserted, the ISR must traverse the interrupt register tree clearing register fields that indicate the error. Consider the abstract register set description in Table 1. Consider these to be the interrupt registers for the example DUT in Figure 1. When the external *intr* signal is asserted, then at least one field in the top_int register is non-zero. An interrupt service routine starts at this top-level register to determine what caused the interrupt and, as a consequence, how to handle the interrupt.

Table 1: Example ISR Register Set

Register	Field	[MSB:LSB]	Default	Access	Description
top_int	rsvd	[31:2]	0	RO	
	rxpath	[1]	0	W1C	Receive path error was detected
	rxpkt	[0]	0	RO	Received packet had an error
pkterr	rsvd	[31:1]	0	RO	
	CRC	[0]	0	W1C	CRC error in received packet

There are two kinds of register fields in the top_int register. First, top_int.rxpath is a write-1-to-clear register field. When this field is asserted it indicates some receive-path error has been detected. The ISR can "handle" the receive path error indication by just writing a 1 to clear this register field. Second, top_int.rxpkt is a read-only register field. The ISR cannot handle this field directly. Instead, this field is the result of a logical-OR of some other set of register fields. In Table 1, that dependency is only pkterr.CRC register field. Therefore, the ISR must also examine the contents of the pkterr register to handle interrupts. When all interrupts have been handled and cleared the ISR may exit and the external *intr* signal should deassert.

4.1 General Interrupt Service Routine Architecture

We have implemented a general interrupt service routine (ISR) as a set of UVM class extensions consisting of an agent with a monitor (and corresponding virtual interface), a virtual sequencer, and a set of virtual sequences, Figure 12, one each per interrupt output. That is, one agent can

handle multiple interrupt outputs when one output is tied to one monitor and its associated virtual sequencer and virtual sequences (each sequence mapping a register or a register field). We assume the ISR tree, per interrupt output, is composed of a mutually exclusive list of register fields as in Figure 13 (A). However, because the architecture overlays virtual sequence instances onto registers and fields, rather than modifying the register and field instances themselves, an overlapping register field ISR tree, in Figure 13 (B), is supported.[6]

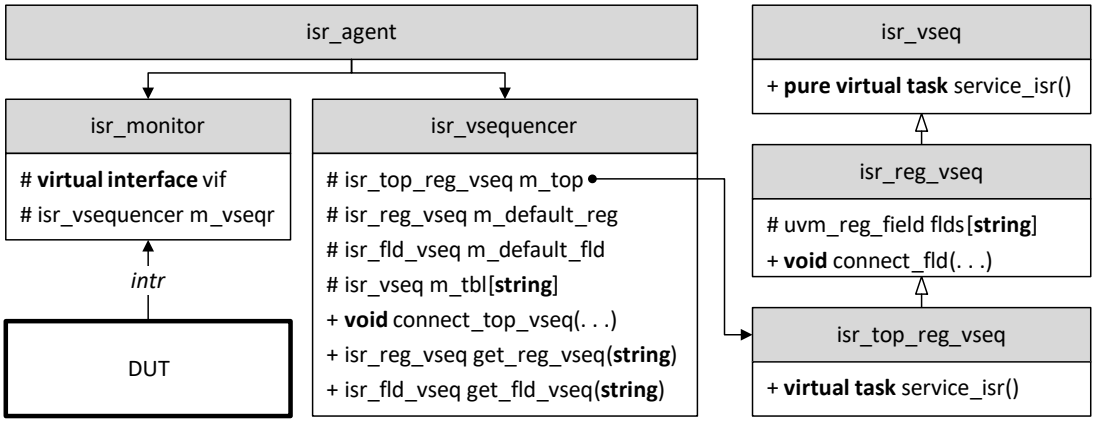

Figure 12: Interrupt Service Routine architecture. The isr_monitor has been attached to the DUT *intr* output from Figure 1.

No sequence in the interrupt service routine architecture is connected to any driver. Instead, the sequences may access registers through a UVM register abstraction layer (RAL) model and access an error injection services (EIS) object. ISR virtual sequence startup and shutdown is handled in the UVM sequence's body() task. Therefore, the main body of each ISR virtual sequence is a tasked named service_isr().

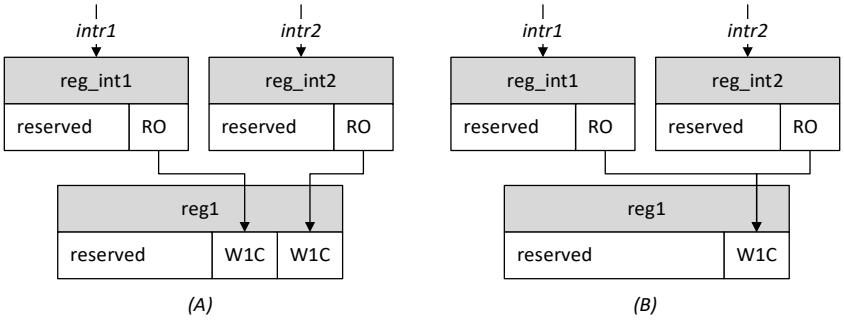

Figure 13: Interrupt service register trees (shaded is whole register, white shows individual fields) when multiple interrupts exist for (A) for mutually exclusive field access or (B) overlapping register field access.

The ISR virtual sequencer, crucially, contains an associative array, m_tbl, keyed by a string that maps register and register full field string paths to its ISR virtual sequence instance (from UVM functions reg.get_full_name() or field.get_full_name()). That is, no register or register field is directly link to another (no linked-list). Instead, the m_tbl array maintains all ISR tree mapping. When no ISR virtual sequence instance is mapped for a register model path then the register or

[6] Note that in the overlapping field case, each ISR may experience blocking during register bus transactions due to both ISR instances accessing the same register

register field default virtual sequence is used in its place. The default register field virtual sequence simply clears the field (for known access types) and reports an error message.

When the DUT asserts the external *intr* signal, in Figure 12, the isr_monitor starts the top register virtual sequence on the isr_vsequencer. When the m_top virtual sequence completes, the isr_monitor verifies the *intr* signal has deasserted. There is a delay between m_top exit and m_top complete due to register access. Therefore the monitor must ensure the *intr* signal checking is after final register write. When the top register virtual sequence starts it creates a sequence item passed between each virtual sequence to mimic the ISR execution stack. In Figure 14, the isr_vseq base class handles the startup and shutdown for all virtual sequence types. This class type is the entry into the interrupt service routine as started by the isr_monitor on the isr_vsequencer.

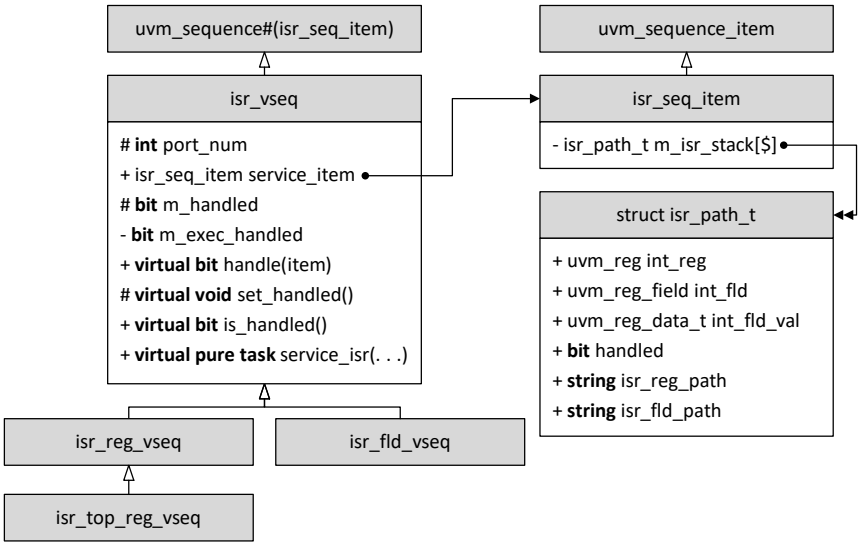

Figure 14: ISR virtual sequence base classes.

4.2 ISR Register Handler

The ISR register virtual sequence base class is presented in Figure 15. This virtual sequence requires a reference to the register it is servicing as is set in the set_service_reg() function. The interrupt register, itself, is only used to determine which register fields should be handled. For each register field that has a non-zero value the isr_reg_vseq will start an isr_fld_vseq to handle that field.[7]

[7] We explicitly indicate comparison to a non-zero value, but in implementation this may be configured, perhaps, to non-reset value.

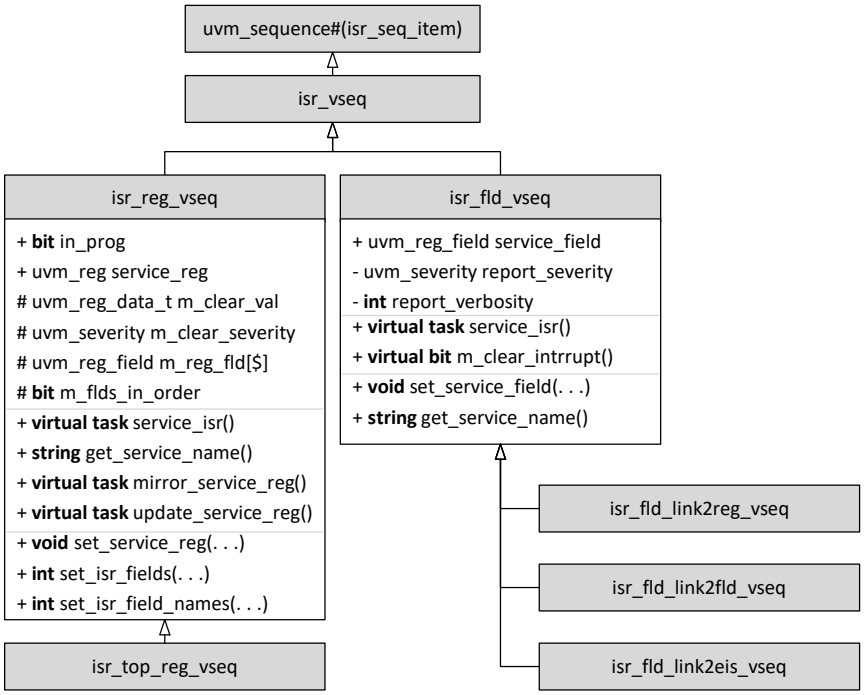

Figure 15: ISR register and register field virtual sequences.

There is some configurability to how the register fields are examined and executed. First, the isr_reg_vseq set_service_reg() function, by default, populates its m_reg_fld queue with all fields in the register via the uvm_reg get_fields() function (i.e., same order as specified at register build() time). However, with the set_isr_fields() or set_isr_field_names() functions, the user may specify both a subset of fields and their examination order. Then, during service_isr() task execution the determine_execution_order() pseudo-code function, in the example code below at line 2, can return a queue of fields either in user specified order or, preferably, random order. Verification generally does not know how firmware will ultimately examine these register fields. Therefore, examination of fields in a random order is preferred so as not to implicitly require specific ordering. Then, for each interrupt register field that is asserted its currently corresponding ISR register field handler is retrieved from the virtual sequencer's table and started via the `uvm_send() macro. During the course of simulation the register field and its corresponding ISR register field virtual sequence handler may change due to error injection. Therefore, the ISR register handler always queries the table for the most current ISR field handler.

```
1  mirror_service_reg();
2  uvm_reg_field ord_fld[$] = determine_execution_order();
3  foreach(ord_fld[i]) begin
4    if(ord_fld[i].get() > 0) begin
5      isr_fld_vseq fvseq = parent.get_reg_fld(ord_fld[i].get_full_path());
6      `uvm_send(fvseq) // handle register field
7    end
8  end
9  update_service_reg();
```

Note that the ISR register virtual sequence handler mirrors its register via the uvm_reg mirror() task before examining register fields. The current value of each field is accessed via the uvm_reg_field get() function. Following register field handling, the ISR register handler updates its register via uvm_reg update() task. Each ISR register field virtual sequence, similarly, only performs uvm_reg_field get() and set() function calls to modify its register field value. In this manner the ISR minimizes the number of register bus cycles and, therefore, simulation time. The ISR architecture assumes regular front-door register access (bus transaction). However nothing *prevents* back-door access (direct RTL hierarchy peek/poke) as long as it is enabled by default prior to ISR processing.

4.3 ISR Register Field Handler

The ISR register field handler base class is presented in Figure 15. The base class and its extension to isr_fld_link2eis_vseq are considered *ISR register field leaf classes*. That is, they do not call the `uvm_send() macro to handle a field or register. The isr_fld_vseq class performs exactly two functions: clear the register field (for known access types) and report an error message. From the isr_vsequencer class, the m_default_reg sequence does only those two functions. Therefore, any time during ISR execution for an interrupt register containing a register field without a corresponding handler, the default field handler is used and will report an error in simulation. During good testing scenarios minimal ISR setup is required to handle interrupt errors. Only the relationships between registers and fields must be defined and only once at the beginning of simulation.

Referring to Table 1, there are two kinds of register fields represented. The top_int.rxpath register field is a leaf field and may be modeled as an isr_fld_vseq instance. If no specific instance is created then the default field handler is sufficient. For the top_int.rxpkt register field, there a relationship between this field and the pkterr register. That is, the rxpkt register field is *not* a leaf field. To handle the rxpkt field the pkterr register must be examined. Therefore, this field's relationship must be instrumented at simulation start as an isr_fld_link2reg_vseq, as shown in Figure 16. This class contains a queue of uvm_reg references rather than just one. We model here one field that is the combination of one or more registers. When there exists a relationship with more than one register to one field, the ordering of `uvm_send() macro calls to ISR virtual sequence register handlers is, preferably, random but may be set at isr_fld_link2reg_vseq construction.

Similarly, it is also possible that one register field is dependent only on a set of register fields rather than a full register. Furthermore, those register fields may not be in one register but spread over multiple registers. We model this scenario with the isr_fld_link2fld_vseq relationship class. When there exists a relationship to more than one register field, the ordering of `uvm_send() macro calls to ISR virtual sequence register field handlers is, preferably, random, but also may set at isr_fld_link2fld_vseq construction. This relationship must be treated with care as it is possible to overlap register mirror/update access task calls during ISR execution.

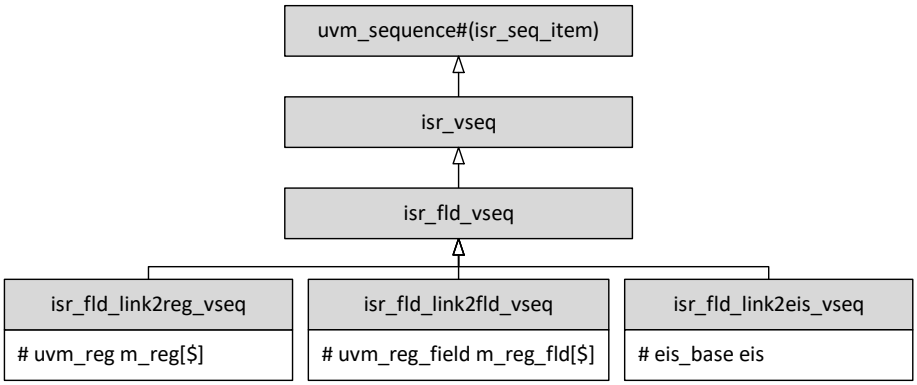

Figure 16: ISR register field relationship classes.

4.4 Integrating the EIS Object into ISR

The ISR virtual sequencer's ISR tree mapping table, m_tbl in Figure 12, allows for easy manipulation of the interrupt service routine itself. As discussed in sections 4.1 to 4.3 , no ISR register or register field virtual sequence handler contains any reference to the next virtual sequence during ISR `uvm_send() traversal (Figure 17 dotted arrows indicate execution not class membership). Instead, the string RAL full path is used to lookup the next sequence in the virtual sequencer's table. Figure 4 presents the EIS object with a function for ISR manipulation, set_expect_interrupt(). During error injection instrumentation in the start_injection() task, the EIS object constructs and sets ISR register field virtual sequence handlers, isr_fld_link2eis_vseq class instances, in the ISR virtual sequencer's table (locally storing any existing virtual sequence for post-error injection restoration). The EIS object *should* only need to concern itself with ISR register field leaf classes as the full ISR relationship tree should already be in place and should not change during simulation.

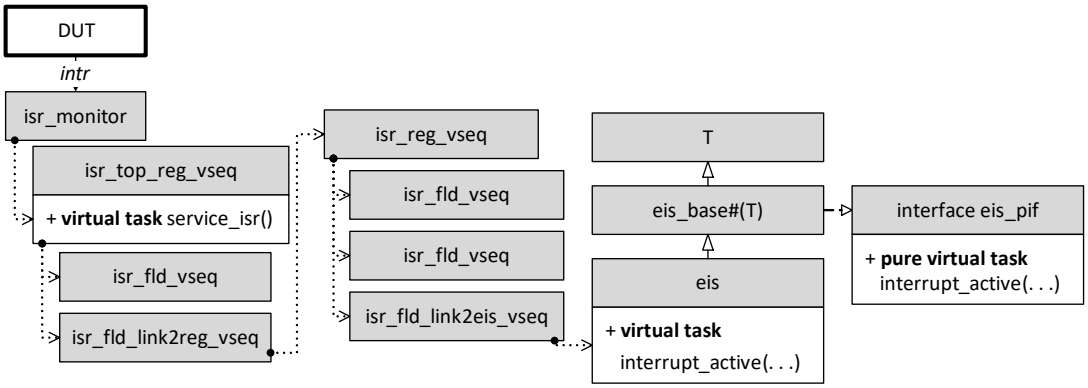

Figure 17: Error Injection Services (EIS) interrupt active path. DUT interrupt triggers the ISR which, eventually, passes control to the eis object via its interrupt_active() task. Dotted arrows indicate execution call hierarchy to service_isr() unless specified.

Figure 17 presents the execution hierarchy upon DUT interrupt output assertion for a completed interrupt service routine within this framework. The isr_monitor uses its reference to isr_top_reg_vseq to start the ISR sequence chain. Exactly one isr_top_reg_vseq may exist per DUT interrupt output (i.e., this framework is extendable to multiple interrupt output signals). Each remaining ISR virtual sequence handler is determined dynamically by table lookup in the virtual sequencer. Each dotted-arrow in the figure represents a call to a virtual sequence handler via the `uvm_send() macro. The body of each virtual sequence handler is implemented in the service_isr()

task. Each instance of isr_fld_vseq represents a *leaf* register field with no dependencies; otherwise a relationship exists between the field and another register (isr_fld_link2reg_vseq) or another field (isr_fld_link2fld_vseq). When the register field is asserted, the isr_fld_vseq clears the field via uvm_reg_field set() and flags an error or note, as desired, then returns.

The EIS object may override, during error injection instrumentation, any isr_fld_vseq with an instance of the isr_fld_link2eis_vseq handler. This class performs the same two functions as its parent class, see class hierarchy in Figure 16, but also executes the EIS object's interrupt_active() task before returning. As the interrupt on this register field is expected, however, no error is reported by the handler. The callback to the EIS object provides an opportunity for the error injection to react to the interrupt by cleaning-up error injection and setup to recover normal simulation. One feasible recovery is to reset the DUT. A suitable reset request mechanism must be implemented to properly jump UVM phase, if required.

5. EIS CRC Implementation

We present an example of the EIS CRC error injection implementation as result of the presentation in this paper. We inject the error onto a packet on the DUT's receive path. As such, we use a Synopsys PCIe VIP callback [6]. The following code is all that is required to setup for and inject an error on the VIP transmit path and expect it on the DUT receive path. The start_injection() task registers itself with the VIP to callback at the next TLP ready for transmit. In the callback function, the EIS object registers its expected interrupt and expected error message. The error message will automatically clear expectation once received. The interrupt must be cleared manually.

```
1   class eis_crc extends eis_base#(svt_pcie_target_app_callback);
2     `uvm_object_utils(eis_crc)
3
4     virtual task start_injection(uvm_phase phase);
5       // Setup for callback from VIP; we will setup expectations then
6       uvm_callbacks#(svt_pcie_target_app,svt_pcie_target_app_callback)::
7           add(null, this);
8     endtask // FYI: Not blocking the current phase
9
10    virtual task interrupt_active(isr_vseq int_vseq,
11                                  output int handled_code);
12      handled_code = 1;
13      clear_expect_interrupts(); // only 1 interrupt required, restore prev
14    endtask
15
16    virtual function void pre_tx_tlp_put(svt_pcie_target_app target_app,
17                                         svt_pcie_tlp transaction,
18                                         ref bit drop)
19      svt_pcie_tlp_exception_list err_list;        // from [6]
20      svt_pcie_tlp_transaction_exception err_item; // from [6]
21
22      // Setup for CRC interrupt
23      isr_fld_link2eis_vseq vseq = new("ecrc_int_vseq", this);
24      set_expect_interrupt(vseq); // calls interrupt_active
25
26      // Setup 1 expected error report; auto-clears expectation when seen
27      set_expect_error("UVC_MON_CRC_ERROR", 1);
28
29
```

```
30      // set the CORRUPT_ECRC exception on the provided transaction from [6]
31      err_list = new("err_list");
32      err_item = new("ecrc_err");
33      err_item.error_kind= svt_pcie_tlp_transaction_exception::CORRUPT_ECRC;
34      err_item.corrupted_data = 32'hffff_ffff; // XOR with ECRC
35      err_list.add_exception(err_item);
36      $cast(transaction.exception_list, err_list.`SVT_DATA_COPY());
37
38      // only 1 required, disable callback
39      uvm_callbacks#(svt_pcie_target_app,svt_pcie_target_app_callback)::
40          delete(null, this);
41    endfunction
42  endclass
```

6. UVM Test Implementation

In our verification environment we have opted to classify each error injection as an enumerated type to allow for easy command-line manipulation. We implement the error injection type selection via an lvm_rand instantiation in our project-specific base test [7]. Some special handling is required to actually instantiate the EIS object leaf class, but then the base test may simply maintain the EIS object as an eis_pif reference.

```
1   typedef enum {
2     ERR_NONE = 0, …, ERR_CRC = 42, …, ERR_PKT_TYPE = 44, …
3   } err_type_t;
4
5   class test_base extends uvm_test;
6     lvm_rand#(int) num_err;
7     lvm_rand#(int) sel_err;
8     err_pif errs[$];
9     `uvm_component_utils(test_base)
10
11    function new(string name = "test_base", uvm_component parent = null);
12      super.new(name, parent);
13      num_errs = new("NUM_ERRS", this);
14      num_errs.push("1");
15      sel_err = new("SEL_ERR", this);
16      sel_err.push($sformatf("%0d", ERR_NONE));
17    endfunction
18
19    virtual task main_phase(uvm_phase phase);
20      int inj_errs = num_errs.next();
21      for(int i = 0; i < inj_errs; i++) begin
22        err_pif = create_eis(sel_err.next()); // create class and return pif
23        if(err_pif != null) begin
24          errs.push_back(err_pif);
25          err_pif.start_injection(phase):
26        end
27      end
28    endtask
29  endclass
30
```

With the above test base class, we can specify the error types available and the number of concurrent errors. We have omitted starting new error injections once these complete, but that implementation is straightforward. For the example test bench starting at Figure 1 and continuing throughout this presentation we may select the CRC error injection by the following command-line (substitute your simulation command):

```
runsim +NUM_ERRS=1 +SEL_ERR=42.
```

However, with our lvm_rand random variable usage, we may also specify a range of error selections as a string directly on the command-line (assuming they are available) [7]:

```
runsim +NUM_ERRS='inside[1:2]' +SEL_ERR='dist{42 := 6, 44 := 2, 0 := 2}'.
```

This command indicates that we should select, with uniform probability, one or two simultaneous error injections and those may be selected from the list of 0 (no error) selected 20% of time, 42 (CRC corruption) selected 60% of time, or 44 (packet type field corruption) selected 20% of time. The two lvm_rand classes instantiated in lines 6-7, above, parse a constraint specified as a string and dynamically apply that constraint at simulation time [7]. Therefore, while the code indicates a default of no errors (good test) we may enable them on the command-line (good test with errors).

As alluded to in section 4. , the EIS object must be take care during error recovery. The example test_base class, above, creates and starts the injection in UVM main_phase. If, following error injection, the appropriate recovery is to perform a DUT hard reset in UVM reset_phase then it is possible to kill the EIS object interrupt_active() virtual task before completion. When jumping out of phase all threads and automatic objects therein are destroyed. Therefore, care must be taken to schedule and prepare for a drastic simulation event while ensuring the EIS object and ISR sequence returns from all method executions prior [8].

7. Conclusions

We have often seen that error testing is regulated to directed UVM leaf tests. Even in a full constrained random UVM test bench, with callbacks, analysis ports, and phasing often it is the job of a single test instance to verify a single error scenario. If that test is not executed in regression then that error scenario is not tested. Furthermore, this directed approach cripples functional cross coverage, especially between testing scenarios. The reality is that we do not know exactly when or how an error will occur in the field. As such, covering both the constrained approach to error injection and randomizing error scenarios with all good tests can cover areas we do not consider and will not enumerate as individual tests.

8. References

[1] Accellera, Universal Verification Methodology 1.1d, 2013.

[2] IEEE Computer Society, SystemVerilog, 2012.

[3] J. Bromley and A. Winkelmann, "SystemVerilog, Batteries Included: A Programmer's Utility Library for SystemVerilog," in *Design and Verification Conference*, San Jose, 2014.

[4] PCI-SIG, PCI Express Base Specification Revision 3.0, 2010.

[5] S. Sutherland and T. Fitzpatrick, "Keeping Up with Chip — the Proposed SystemVerilog 2012 Standard," in *Design and Verification Conference*, San Jose, 2012.

[6] Synopsys, Inc., "VC Verification IP PCI Express UVM User Guide," 2016.

[7] J. Ridgeway, "Interchangeable SystemVerilog Random Constraints," in *Synopsys User Group (SNUG)*, San Jose, 2014.

[8] J. Ridgeway and D. Mehta, "Global Broadcast with UVM Custom Phasing," in *Design and Verification Conference*, Bangalore, 2014.