Agile and dynamic functional coverage using SQL on the cloud

DVcon Europe 2019 – Paper number 278-XD146

Filip Dojcinovic, Veriest Solutions, Belgrade, Serbia (*filipd@veriests.com*)

Mihailo Ivanovic, Veriest Solutions, Belgrade, Serbia (*mihailoi@veriests.com*)[1]

***Abstract*— This paper we will present a functional coverage solution that is based on log files, and leverage generic database technologies to parse, ingest and query the data. Such tools are easily available from AWS, Google and other public cloud providers.**

Keywords—functional; SQL; coverage; cloud;

I. Introduction

Functional coverage is a key metric in most verification projects and is used by many teams to “drive” the verification process, to determine which areas are sufficiently verified and which ones need further testing. Unlike metrics such as number of bugs found, and code coverage, it is decoupled and abstracted from the design, and can therefore be used not only by engineers, but also by managers and project leads. Unlike metrics such as number of tests written, it is based on data coming from the actual DUT via monitors and is therefore more trustworthy and provides a better measure of progress. Unfortunately, it suffers a few major shortcomings that make it a much less powerful tool than it could have been.

In this paper we will present a functional coverage solution that is based on log files, and leverage generic database technologies to parse, ingest and query the data. Such tools are easily available from AWS, Google and other public cloud providers.

The following paragraphs show how this flow and these technologies not only address all the shortcomings mentioned above, but also allow for much more thorough analysis of data at hand, and deep-dive where needed, without being required to rerun lengthy regressions.

II. Functional Coverage

Functional coverage is a key metric in verification projects used to determine which areas are sufficiently verified, and which ones need further testing, thus allowing the verification manager to direct the team's effort. It is called “functional”, because unlike code coverage, it is usually derived from documentation describing the intended behavior of a design, rather than from implementation. Since the intended behavior is often distributed across multiple components of the design, functional coverage usually relies on combinations of events and data collected during testing, rather than on testing going through a given line of code.

To give an example, a design might be required to complete packet processing in less than a given time, in the presence of an interrupt. Functional coverage for this requirement will look for a situation where an interrupt is emitted simultaneously with an incoming packet and log it. In conjunction with functional coverage, automatic checkers will verify that processing time never exceeds a given number. For such a requirement, it is usually very hard to pinpoint a few lines of code implementing it. Therefore, functional coverage is the state-of-the-art metric in IC design, while code coverage is usually considered less credible.

[1] The authors would like to thank Mr. Avidan Efody for his contribution to this paper

III. Traditional Flow

The traditional flow for obtaining functional coverage is described on the left-hand side of diagram #1 below. Based on the assumption that a complete specification is at hand, the verification team will try to identify indicators that a specific situation has been tested, and then implement coverage that monitors these indicators and logs them in the coverage database. Since coverage is implemented as part of the testbench, the verification team must run some tests or an entire regression before it can look at the coverage results to check what situations have been covered.

IV. Traditional Flow Shortcomings

The traditional flow suffers a few major shortcomings that make it inflexible, partial, platform specific and overall limiting. Because coverage is implemented as part of the testbench, and a regression must be run to obtain the results, iteration time for any bugs within the coverage collection code is extremely long. Since coverage is often implemented towards the end of a project, and since some of the coverage will only be hit by very few tests (coverage for interesting corner-cases is a typical example), it is not uncommon for users to run very long regressions, only to find that a bug prevented relevant coverage from being collected.

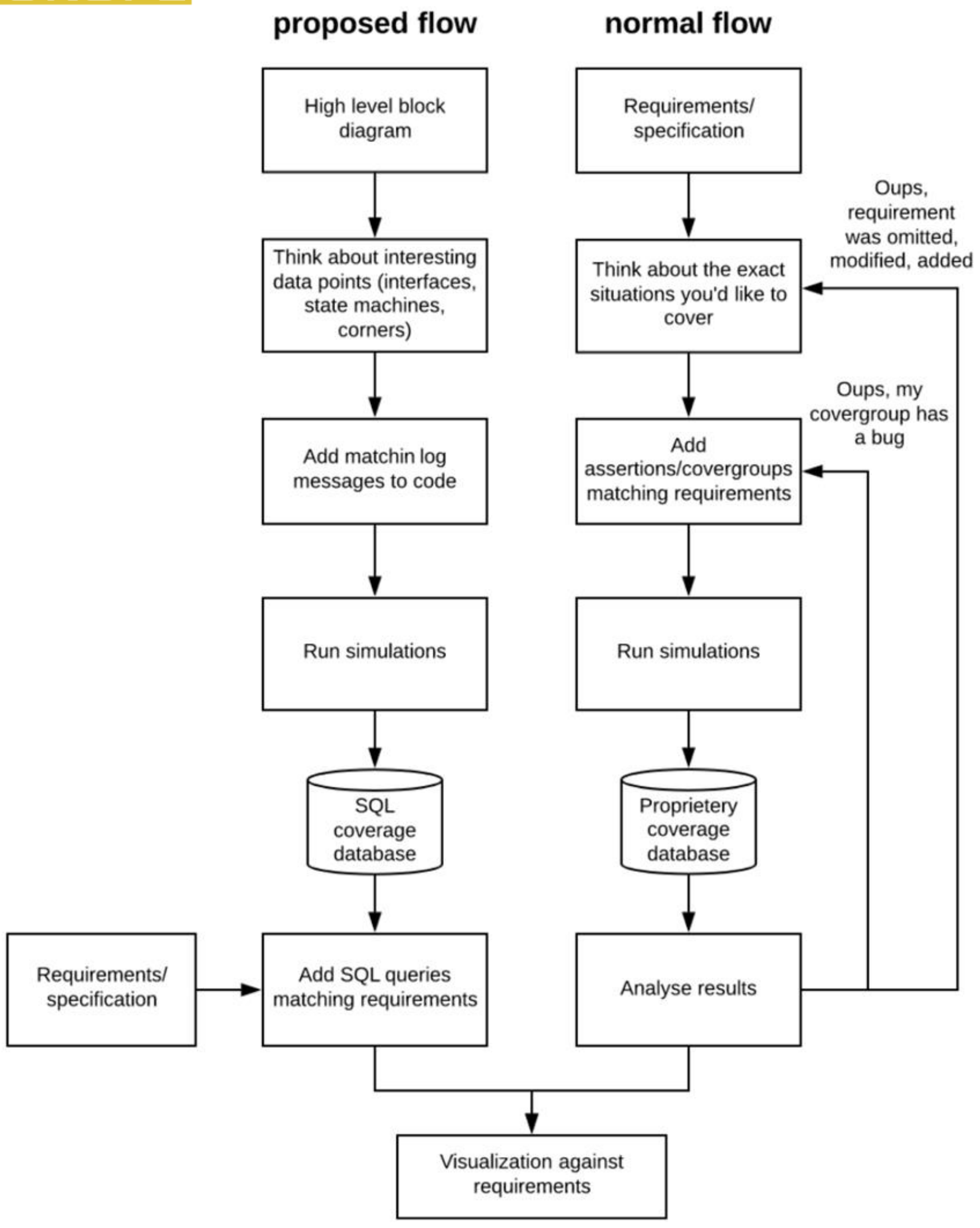
Worse, at a higher level, coverage flow deviates from a normal analysis flow, in that it forces the users to ask all the questions upfront, before they have seen the data collected. Quite often, analysis is an iterative process, where one question about the data leads to another. The most interesting questions are not the first ones asked. With the traditional coverage flow, users are limited to the coverage they have defined before seeing the data. If coverage results trigger some more questions, another long iteration has to be done to get answers. Most often, the effort involved makes users simply not ask the questions.

Finally, the traditional flow is hard to integrate with any platform/language outside RTL simulation in SystemVerilog or *e*. Users looking to merge data coming from sources such as C/C++ FW/SW, SystemC, Matlab, VHDL testbenches with their SystemVerilog/*e* coverage can usually not do it without writing custom code. With the exception of emulation, where some vendors offer partial and limited support of coverage, no platform – SystemC, FPGA prototyping or real silicon – offers any support for the coverage constructs supported by SystemVerilog/e.

V. Proposed Flow

The flow proposed here is described on the right-hand side of diagram #1. In this flow, the user simply logs the data that moves across strategic points in the design or the events emitted by key elements. Examples of strategic points and key elements might be interfaces, state machines, configuration and status registers, etc. Following the test these logs are uploaded to a cloud provider storage such as AWS, Google or others, and are then parsed and placed in an SQL database. Coverage information is obtained by querying this database post testing. Note that many cloud providers provide automatic parsing and ingestion of data placed in their storage solution into a variety of databases, both SQL and non-SQL. Such flow (i.e. upload logs, then parse/ingest them into a database) is extremely common and is an integral part of a wide variety of applications from web-servers to cloud managed production. It should also be noted that such flow can often run serverless, without any requirement from the user to manage either storage or database servers.

The proposed flow solves most of the shortcomings of the traditional flow. By moving coverage extraction to post-testing it allows fast iteration, debug, and iterative incremental investigation of the data at hand. By using log files instead of specific syntax, it allows coverage information to be collected from any language and platform. Furthermore, as the paper will show, using SQL for extracting coverage data, doesn't only cover almost everything possible with SystemVerilog/*e* cover groups/assertions, but also enables queries that usually require a lot of buggy glue logic in testbench environments. Last but not least – the flow is simple and straightforward.



proposed flow
normal flow
High level block diagram
Requirements/ specification
Think about interesting data points (interfaces, state machines, corners)
Think about the exact situations you'd like to cover
Oups, requirement was omitted, modified, added
Add matchin log messages to code
Add assertions/covergroups matching requirements
Oups, my covergroup has a bug
Run simulations
Run simulations
SQL coverage database
Proprietery coverage database
Requirements/ specification
Add SQL queries matching requirements
Analyse results
Visualization against requirements

Diagram #1

VI. SQL AS A COVERAGE TOOL

To show how SQL can deliver all forms of coverage available as a part of System Verilog coverage syntax and much mode, we will demonstrate how SQL can be fine-tuned, focused and extended without re-running any simulations.

The important point is that there is no need to become a SQL expert, but rather, a high-level understanding of queries is sufficient. As we will see, there are many new possibilities brought by SQL queries, but also a few limitations.

For example, assume that there is an already parsed transaction log file collected on an AXI interface and placed the transactions in an SQL table called **axi_if_1**. The first few lines and columns of the table **axi_if_1** are shown below:

```
time rd_wr addr       burst len

150  RD    165377426  INCR  12

250  RD    2310710676 FIXED 13

350  WR    2328599037 FIXED 15

360  WR    2921785595 INCR  6

500  RD    1490070710 INCR  0

550  RD    3668650794 FIXED 8

1000 WR    1314868187 INCR  13

1100 RD    3114753989 FIXED 10

1110 RD    2032547025 FIXED 14

1120 WR    2834194867 INCR  4

1490 RD    4294967295 FIXED 8
```

The following query would give us all distinct values in the burst type column:

```
SELECT DISTINCT burst FROM axi_if_1

burst

INCR

FIXED
```

First thing to note is that there is a coverage hole in the table result of the select query, as the WRAP value does not appear in any transactions. Since SQL can look into the transactions that happened, without additional information about the expected coverage, it cannot tell us anything about coverage holes. This behavior, although obvious, is somewhat surprising if you’re used to SV coverage, since SV does provide information about coverage holes, without any input about the expected coverage space from the user. For example, if you have the following:

```
typedef enum burst_type_t {FIXED = 0, INCR = 1, WRAP =2};

//…
burst_type_t burst;


covergroup axi_tr;

   burst : coverpoint burst;

endgroup
```

SV will automatically show a coverage hole for the WRAP value, because it uses the type definition as a coverage space description. To provide it with the information that SV obtains via type definitions, using SQL approach will have to create tables like the ones below for all enumerated types. For the example, the assumption is that this info is also represented in the log file as shown below:

```
name value

RD   0

WR   1

name  value

FIXED 0

INCR  1

WRAP  2
```

With the type information, the query is modified to contain following steps:

- Select the name column from the **burst_type** table shown above. This column contains our expected values
- Match each **name** value to all transactions with a corresponding **burst** field, from axi_if_1. The only value that will be left unmatched by any transaction is **WRAP**
- Collapse all lines that share an identical **burst_type.name** into one, leaving it with the 3 entries that were expected.
- Improve it using an ***if*** function, to present True/False values

```
SELECT DISTINCT # (3)
    t.name AS expected, # (1)
    IF(axi.burst IS NOT NULL, 'TRUE', 'FALSE') AS hit #(4)
FROM burst_type t
LEFT JOIN axi_if_1 axi
    ON t.name=axi.burst #(2)
```

Query result:

```
expected hit

INCR     TRUE
```

```
FIXED     TRUE

WRAP      FALSE
```

VII. Cross Coverage

When showing the example of using SQL for cross-coverage, one can appreciate the full potential of this approach. For extracting the cross-coverage of burst type and rd/wr transactions, using the only two type definitions shown above, all one need to do is to run the following simple query.

```
SELECT
    t1.name AS burst,
    t2.name AS direction
FROM burst_type t1
CROSS JOIN rdwr_type t2
```

The result is a table showing all possible expected values:

```
burst direction

FIXED RD

FIXED WR

INCR  RD

INCR  WR

WRAP  RD

WRAP  WR
```

To match those expected values to the actual values, what is needed is a left join with the AXI transactions table, but this time with matching lines that have both burst and rd/wr equal.

```
SELECT DISTINCT
    t1.name AS burst,
    t2.name AS rd_wr,
    IF(axi.burst IS NOT NULL, 'TRUE', 'FALSE') AS hit
FROM burst_type t1
CROSS JOIN rdwr_type t2
LEFT JOIN axi_if_1 axi
    ON t1.name=axi.burst AND t2.name=axi.rd_wr
```

The resulting coverage results table:

```
burst direction hit

INCR  RD        TRUE

FIXED RD        TRUE

FIXED WR        TRUE

INCR  WR        TRUE

WRAP  RD        FALSE

WRAP  WR        FALSE
```

On AXI it is common to cross direction, and memory segment, to make sure all segments were accessed. Using SQL variables, the address range can be split in 4 and generated a new list of expected buckets with the following query. The *SELECT ctr etc.* part in the brackets, is simply SQL's way of generating a column with numbers 1,2,3,4.

```
SET @buckets = 4

SELECT
    t1.name AS direction,
    t2.ctr * 1000000000 AS segment
CROSS JOIN rdwr_type t1
CROSS JOIN (SELECT ctr FROM ctr_to_100 WHERE ctr < @buckets) t2
```

The resulting cross-coverage expected table of direction vs. memory segment:

```
name segment

RD   0

WR   0

RD   1000000000

WR   1000000000

RD   2000000000

WR   2000000000

RD   3000000000

WR   3000000000
```

Matching it to the actual results only requires one additional *AND* in the condition that does the matching. As an addition *ORDER BY* clause is added at the end so that combinations are easy to follow

```
SELECT DISTINCT
    t1.name AS direction,
    t2.ctr * 1000000000 AS segment,
    if(axi.addr is not null, 'TRUE', 'FALSE') AS hit
CROSS JOIN rdwr_type t1
CROSS JOIN (SELECT ctr FROM ctr_to_100 WHERE ctr < @buckets) t2
LEFT JOIN axi_if_1 axi ON
    t1.name=axi.rd_wr AND
    t2.ctr=floor(axi.addr/1000000000)
ORDER BY direction,segment
```

Coverage results table:

```
name segment    hit

RD   0          TRUE

RD   1000000000 TRUE

RD   2000000000 TRUE

RD   3000000000 TRUE

WR   0          FALSE

WR   1000000000 TRUE

WR   2000000000 TRUE

WR   3000000000 FALSE
```

For ignoring one of the combinations in the cross-coverage results an additional WHERE command can be added, as shown below.

```
SELECT DISTINCT
    t1.name AS burst,
    t2.name AS rd_wr,
    IF(axi.burst IS NOT NULL, 'TRUE', 'FALSE') AS hit
FROM burst_type t1 CROSS JOIN rdwr_type t2
LEFT JOIN axi_if_2 axi ON
    t1.name=axi.burst AND
    t2.name=axi.rd_wr
WHERE
    t1.name <> 'INCR' OR
    t2.name <> 'RD'
```

```
burst direction hit

FIXED RD        TRUE

FIXED WR        TRUE

INCR  WR        TRUE

WRAP  RD        FALSE
WRAP   WR          FALSE
```

VIII. Coverage percentage

One of the most important parts of collecting coverage in functional verification, is the coverage percentage number indicated at the end. To finish the SQL example a TRUE/FALSE resulting column can be replaced with binary 1/0 notation and an averaging function can be introduced. Below is an example showing how to get coverage numbers across burst type, direction and memory segment.

```
SELECT AVG(hit)*100 AS coverage_number FROM (
    SELECT DISTINCT
       t1.name AS burst,
       t2.name AS direction,
       t3.ctr * 1000000000 AS segment,
       IF(axi.addr IS NOT NULL, 1, 0) AS hit
FROM burst_type t1
CROSS JOIN rdwr_type t2
CROSS JOIN (SELECT ctr FROM ctr_to_100 WHERE ctr < @buckets) t3
LEFT JOIN axi_if_2 axi ON
    t1.name=axi.burst AND
    t2.name=axi.rd_wr AND
    t3.ctr=floor(axi.addr/1000000000)) t4
```

Resulting number for the tables above:

```
coverage_number

29.1666
```

IX. Conclusion

At a high level, practically anything that is available for System Verilog coverage can be done with SQL. To improve the usability of this method, all SQL commands can be wrapped into a scripting language to create coverage functions that can mimic SV API and also keep the SQL hidden upfront.

To prove the possible improvement of this approach and show that it is not all the same as System Verilog coverage engine, all queries shown:
- Are dynamic and platform independent
- Can be done long after the simulation has ended
- Can be modified and debugged on-the-fly
- Give the same information in a more convenient way

In the proposed solution the SystemVerilog's role is reduced from analyzing the data using cover groups, to merely printing out the information into a log file to be post processed later. The steps needed to setup the cloud-based coverage include:
- Having the logs and type information files uploaded to the cloud
- Turn these logs in to SQL tables (using available cloud services)
- Query the tables for coverage as described
- As a last step visualize the tables linked to a test plan, possibly alongside other forms of coverage (formal, SVA).

The SQL language is a natural way of working with data, so much more possibilities can be taken from it and used for improving the way functional coverage is implemented and collected for any verification platform.