Agile and dynamic functional coverage using SQL on the cloud

Filip Dojcinovic, Veriest Solutions, Belgrade, Serbia, (filipd@veriestS.com)
Mihailo Ivanovic, Veriest Solutions, Belgrade, Serbia, (mihailoi@veriestS.com)

The authors would like to thank Mr. Avidan Efody for his contribution to this paper
Introduction

• Functional coverage a key metric in most verification project
• Used often to “drive” the verification process
• decoupled and abstracted from the design
• suffers a few major shortcomings
  – Hardly portable to anything outside SV running on a hardware simulator
  – Can’t be changed in light of the results
  – No adding new cover points after running
  – way too static, platform limited, and costly to implement
Functional Coverage

• Coverage collection vs coverage visualization
• Verification plan document linked to coverage results very convincing
• UCIS standard - interoperability of verification coverage data across multiple tools
• Proposed solution addresses coverage collection shortcomings
  – by using log files as the raw data
  – allow coverage to be collected from any language/platform combination
  – by using a standard SQL to process the data
  – enabling exploration, refinement, and even queries that combine data and sequences of events
  – leveraging UCIS can be integrated with any other sources of coverage
Traditional vs Proposed flow

- In a traditional flow -> right hand side,
  - starts from a list of requirements or spec,
  - thing about exact situations to cover
  - coverage model including cover groups and assertions is derived.
  - Regressions are run, and results are then visualized
- holes would require debugging, patching and rerunning
Proposed Flow

– while coding testbench come up with is a list of interesting points to watch
– add logging code into those places
– Run the regression to get a database
– At the end use requirements
– Look at the requirements and link them to queries
SQL as a coverage tool

- SQL can be fine-tuned, focused and extended without re-running the sim
  - High level of SQL queries enough - Many new possibilities, also few limitations
- assume that there is an already parsed transaction log file collected on an AXI interface
- placed the transactions in an SQL table called `axi_if_1`
SQL as a coverage tool 2

• On AXI common to cross burst type and direction

```sql
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'
```

Resulting table:

<table>
<thead>
<tr>
<th>burst</th>
<th>direction</th>
<th>hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCR</td>
<td>RD</td>
<td>TRUE</td>
</tr>
<tr>
<td>FIXED</td>
<td>RD</td>
<td>TRUE</td>
</tr>
<tr>
<td>FIXED</td>
<td>WR</td>
<td>TRUE</td>
</tr>
<tr>
<td>INCR</td>
<td>WR</td>
<td>TRUE</td>
</tr>
<tr>
<td>WRAP</td>
<td>RD</td>
<td>FALSE</td>
</tr>
<tr>
<td>WRAP</td>
<td>WR</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

• All possible expected values
• match those expected values to the actual values, what is needed is a 'LEFT JOIN' with the AXI transactions with matching lines that have both burst and rd/wr equal
• ignoring one of the combinations with ‘WHERE’
• Find which combinations were hit
Coverage percentage

- One of the most important part in functional coverage

Example: Coverage numbers across burst type, direction and memory segment.

```sql
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
```

Average function can be introduced

TRUE/FALSE column replaced with binary

the address range split in buckets and generated a new list of expected buckets

coverage_number
29.1666
Getting the data on the cloud

• Cloud service – any available service can be used
• Data manipulation glue code – python used
• Steps
  – Print transactions and data types into log
  – 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 (legacy SV, formal, SVA).
Printing

• Only one step in verification environment – print statements
  – transactions
  – simulation points and
  – all possible enumerated data types …

• The example of one way how to do it is shown below:

```verbatim
// initialization section: print type information for the fields in our log
$display("# Transaction meta: %s, %d, %s, %d, %s", $typename(tr.dir),
    $size(tr.addr), $typename(tr.burst), $size(tr.len), $size(tr.id),
    $typename(tr.lock));

// run section: print the interesting parts of each transaction into the log
$display("# Time: %t, dir: %s, addr: %d, burst: %s, len: %d, id: %d, lock: %s",$time(), tr.dir.name, tr.addr, tr.burst.name, tr.len, tr.id, tr.lock.name);
```
Additional data

• To get cross coverage including holes the type information needed
• $display statement translate into a format that can be read into a database.
• Easily done with python

```json
types.json
{"enum_type_name": "axi_vip::dir_t", "enum_string": "RD", "enum_int": "0"}
{"enum_type_name": "axi_vip::dir_t", "enum_string": "WR", "enum_int": "1"}
{"enum_type_name": "axi_vip::burst_t", "enum_string": "FIXED", "enum_int": "0"}
{"enum_type_name": "axi_vip::burst_t", "enum_string": "INCR", "enum_int": "1"}
{"enum_type_name": "axi_vip::burst_t", "enum_string": "WRAP", "enum_int": "2"}
{"enum_type_name": "axi_vip::lock_t", "enum_string": "NORMAL", "enum_int": "0"}
{"enum_type_name": "axi_vip::lock_t", "enum_string": "EXCLUSIVE", "enum_int": "1"}
{"enum_type_name": "axi_vip::lock_t", "enum_string": "LOCKED", "enum_int": "2"}
```

```csv
columns.csv
dir, axi_vip::dir_t, 0
addr, int, 32
burst, axi_vip::burst_t, 0
len, int, 4
id, int, 4
lock, axi_vip::lock_t, 0
```
Upload to the cloud

- Directory structure on the cloud
  - For a single cover group
  - Sampled at the same time

```
simple_tb/
├── test1
│   ├── axi_master_1
│   │   ├── columns
│   │   │   └── columns.csv
│   │   └── log
│   │       └── transactions.log
│   └── types_info
│       └── types.json
```
SQL tables

• Cloud platforms have different services that employ SQL queries on the data
• Used example platform – service that directly creates DB and tables from files
  • "json" and "csv" files are natively translated
  • Log files can be translated with user defined format
SQL tables

- Main table – log
- Others: Meta data for all types, Enum type specific table

```sql
CREATE EXTERNAL TABLE demo.axi_if1_trans (
  `time` bigint,
  `dir` string,
  `addr` bigint,
  `burst` string,
  `len` smallint,
  `id` smallint,
  `lock` string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  'input.regex'='# Time: *([^ ^,]*)\s*, dir: *([^ ^,]*)\s*, addr: *([^ ^,]*)\s*,
  burst: *([^ ^,]*)\s*, len: *([^ ^,]*)\s*, id: *([^ ^,]*)\s*, lock: *([^ ^,]*)',
)
LOCATION 's3://db-name/simple_tb/test1/axi_master_1/log/
TBLPROPERTIES ('has_encrypted_data'='false');
```
Last step – query & visualize

- Use queries to get interesting points

Advanced example:
- interrupted exclusive read/write pairs

- group the results by the time of the exclusive-read
- ask for the minimum on the interfering write and the exclusive-write
- To remove the false paths

order the results by addr and write_time and look for max(read_time)

<table>
<thead>
<tr>
<th>addr</th>
<th>read_time</th>
<th>interrupted_at</th>
<th>write_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1490070710</td>
<td>0</td>
<td>26</td>
<td>181</td>
</tr>
<tr>
<td>1490070710</td>
<td>207</td>
<td>290</td>
<td>300</td>
</tr>
</tbody>
</table>

```
select first_tr.addr as addr, first_tr.time as read_time, min(middle_tr.time) as interrupted_at, min(second_tr.time) as write_time
from (select row_number() over () as num,
       inner1.time,inner1.addr,inner1.dir,inner1.lock
       from axi_if1_transactions inner1
       where inner1.dir = 'WR' or inner1.lock = 'EXCLUSIVE'
       order by inner1.addr, inner1.time) as first_tr,
from (select row_number() over () as num,
       inner1.time,inner1.addr,inner1.dir,inner1.lock
       from axi_if1_transactions inner1
       where inner1.dir = 'WR' or inner1.lock = 'EXCLUSIVE'
       order by inner1.addr, inner1.time) as second_tr,
from (select row_number() over () as num,
       inner1.time,inner1.addr,inner1.dir,inner1.lock
       from axi_if1_transactions inner1
       where inner1.dir = 'WR' or inner1.lock = 'EXCLUSIVE'
       order by inner1.addr, inner1.time) as middle_tr
where first_tr.addr = second_tr.addr and
      second_tr.addr = middle_tr.addr and
      first_tr.lock = 'EXCLUSIVE' and second_tr.lock = 'EXCLUSIVE' and
      first_tr.dir = 'RD' and second_tr.dir = 'WR' and middle_tr.dir = 'WR' and
      first_tr.num < middle_tr.num and middle_tr.num < second_tr.num
group by 1,2;
```
Conclusions

- At a high level SQL can replace almost all aspects of System Verilog coverage
- Using queries
  - 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
  - Can do much more with the data at hand
Questions?