2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Generating Bus Traffic Patterns

Jacob Sander Andersen, CTO, SyoSil ApS, Taastrup, Denmark (*jacob@syosil.com*)

Lars Viklund, Expert Engineer, Axis Communications AB, Lund, Sweden (*lars.viklund@axis.com*)

Kenneth Branth, Senior Consultant, SyoSil ApS, Taastrup, Denmark (*kenneth@syosil.com*)

***Abstract*— During the block level verification of the modules in an ASIC using SystemVerilog with the UVM methodology [1] we were required to generate specific traffic patterns for a bus protocol as requested by the architect (design engineers, system architect, *etc*.). Particularly the specification of the traffic pattern was problematic, as it was ambiguous and could be interpreted differently by the architect and the verification engineers. For improved reusability across various verification environments and platforms the traffic patterns could instead be accurately expressed by using a domain specific language (DSL) [2]. The DSL could then easily be converted into executable code for producing the expected traffic patterns.**

Keywords—UVM;domain specific language; SystemVerilog; traffic pattern; bus-based protocol

I. INTRODUCTION

The work presented here is related to the work recently presented by the Accellera Portable Test and Stimuli (PSS) working group [8] as both works deal with portable stimuli. However, this paper focuses on very fine-grained traffic patterns on a single device under test (DUT) interface, targeting elements like:

- The functional specification. Certain patterns must be observed on a given bus. Especially, the specification of performance requirements. Performance standards must be met under certain traffic patterns applied to a given bus.
- Verification closure. Verification metrics (structural, functional coverage etc.) can only be closed if certain traffic patterns are applied to a given bus.

On the other hand, the PSS work is more focused on the specification of system scenarios and coarser traffic patterns. Furthermore, this work is inspired by [3], [4], [5] and [6] as they provide useful information on UVM sequences, scheduling and especially CSP (Communicating Sequential Processes) for inspiration on the syntax.

The traffic patterns can be viewed as several (more than one) transaction producers trying to access the same interface, involving randomization of the individual transactions, and synchronization and load balancing among the producers. We have previously tried two methods for capturing these properties:

1. Informal natural language: Initially, the architects expressed the traffic patterns using informal natural language. This lead typically to different interpretations between the architect and the verification engineer.
2. Transaction diagrams: To improve over the natural language, we switched to using transaction diagrams. However, it was in many cases unclear which aspects of the diagram that could be generalized, if certain elements could be randomized or if the order mattered.

In retrospect, the workload can be significantly reduced if an exact way of expressing these traffic patterns are used by the architect and an underlying, preferably automated, process was supported for generating traffic patterns.

A simple example is a traffic pattern for a bus protocol like the well-known AMBA AXI protocol [9] where a sequence of reads and writes for two different address areas (See Figure 1) are carried out so that:

- Reads are done within a specific separate address area (BUF0) from a start address and with incrementing addresses.
- Writes are done randomly to the same address area as above (BUF0) but with slower frequency so less data is written as is read.
- Writes are done to another address area (BUF1) which is not overlapping with random addresses where the first write must only start after 4 reads have been done.

This can be viewed as three individual producers of transactions:

- Producer A: Reads to BUF0.
- Producer B: Writes to BUF0.
- Producer C: Writes BUF1 with a simple synchronization between them to ensure that producer C starts only after producer A has done 4 transactions.

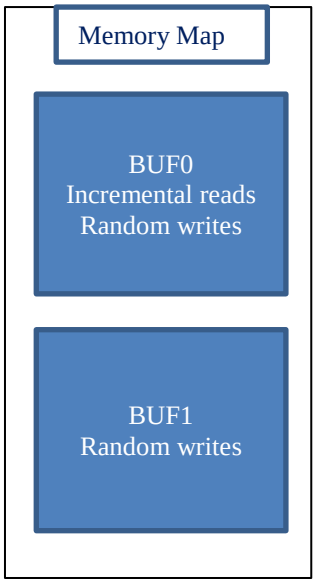

Figure 1: Memory Map

Figure 2 shows a solution to this scheduling problem among the set of legal solutions.

We model the traffic patterns by defining a transaction sequence model for which we define a DSL which captures the set of transaction sequences (lists, parallelism with configurable scheduling, repetition, scheduling, transaction state (start/end), *etc*.) and associated operations in conjunction with multiple producers of transactions (scheduling, synchronization, *etc.*). Section III will provide more details.

The DSL is implemented as an embedded DSL (eDSL[1], [7]). Thus, no parser needs to be implemented and the benefits of a full featured programming language can be exploited directly. This is explained in more detail in section V.

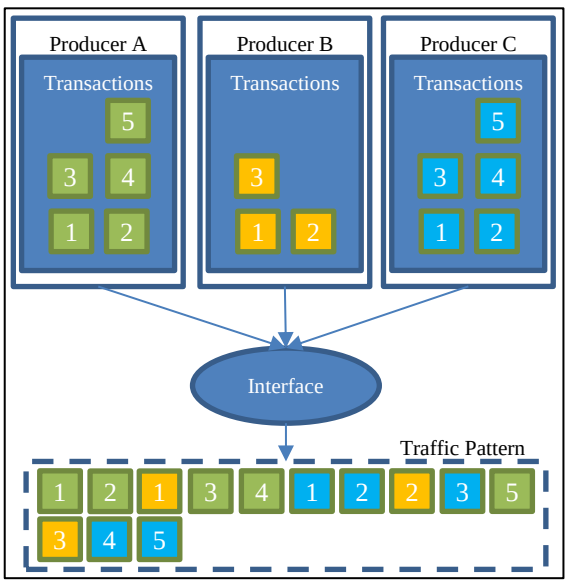

Figure 2: A legal traffic pattern solution to the simple example

[1] An embedded domain-specific language (eDSL) is a domain-specific language that's defined in terms of a more powerful general-purpose "host language"

II. SCOPE DEFINITION

The scope is kept to targeting a single DUT interface throughout the paper for simplicity. However, the concept can be extended to handle the coordination of the production of transactions on multiple DUT interfaces. The bus traffic pattern presented here is generally an abstraction for well-known protocols like AMBA-AXI or similar.

Furthermore, the paper is not targeting how and which transactions the transaction producers are producing and how they are constrained related to each other (Internally for the same transaction producer or across the transaction producers). The work presented here merely expresses a model for the transactions coming from different transaction producers but not the contents of the transactions.

Also, the synchronization and scheduling are based on the number of transactions or if the transactions have started or finished.

III. TRANSACTION SEQUENCE MODEL

The transaction sequence model needs to provide the fundamentals for expressing generic bus patterns with respect to transaction sequences coming from multiple transaction producers in parallel, the synchronization among them and scheduling. Abstractly this can be viewed as a source/sink model with multiple sources and a single sink as depicted in Figure 3. The model then needs to describe how the transactions produced by the sources are executed on the sink being the bus interface. The cloud in Figure 3 represents the potential synchronization, scheduling etc. among the sources.

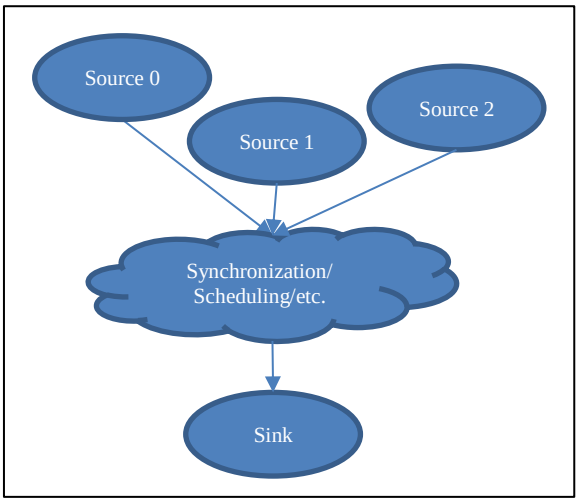

Figure 3: Source/Sink Model

Having the source/sink model fresh in memory we can now refine this model into a more fine-grained model in which the sources are defined as Transaction Producers (**TP**) and the cloud object is refined into a Transaction Sequence (**TS**) Graph. This model allows us to have a separation of concerns:

- The **TS** graph is created once by a tool and instantiated inside the verification environment
- The **TP**s generate the transactions when requested
- The bus traffic pattern at the sink is then the consequence of generating transactions by traversing the **TS** graph and selecting transactions produced by the transaction producers
- The chosen transaction is then applied to the bus interface sequentially, e.g. by an UVM agent which also could allow multiple outstanding transactions.

Ultimately, the transaction sequence model can is defined as consisting of the following components:

- **TP:** Capable of producing a specified number (N) of randomized transactions. A **TP** may be referenced by multiple Transaction Sequence Elements (**TSE**, see below for definition). Furthermore, the **TP** instance has the following state and functions:
  - A state consisting of:
    - **started_trans:** The global number of started transactions for the **TP**. Incremented whenever a transaction is started. Never decremented.
    - **ended_trans:** The global number of ended transactions for **TP**. Incremented whenever a transaction is ended. Never decremented.
  - A set of functions:

  - **started(<*N*>):** Returns: False if the producer has not started N transactions, True if it has started N transactions
  - **ended(<*N*>):** Returns: False if the producer has not ended N transactions, True if it has ended N transactions

- **Transaction Sequence (TS)**: A graph expressing the relation between the defined TPs and optional scheduler strategies using the nodes known as **Transaction Sequence Elements (TSE)**.

- **Schedulers:** Capable of specifying different scheduling strategies such as "Weighted" (Scheduling based e.g. on the number of transactions from each producer related to each other with a weight), "Round Robin", etc. Attributes for each type of scheduling strategy can also be specified, e.g. weights for the weighted strategy. A scheduler instance can be reused in multiple parallel constructs. Hence. State for each reference must be maintained by the scheduler. Additionally, the termination criteria must also be specified.

- **Transaction Sequence Elements (TSE):** The **TSE**s defines the nodes of the **TS** graph. There are different **TSE** node types. Each node type has also a state and a set of functions which are like the **TP** state and functions. They are needed as waits should be done both on **TSE**s and **TP**s.
  - **TSE** types:
    - **TST:** Produces a transaction from a given producer.
    - **TSS:** Sequence node. Allows **TSE**s to be ordered sequentially.
    - **TSP:** Parallel node. Allows scheduling among any number of **TSE**s with a specified scheduling strategy and a termination criterion
    - **TSR:** Repetition node. Repeat a given **TSE** N number of times.
    - **TSW:** Waiting node. The node waits until a Boolean condition is met. The **TSE** and **TP** functions can be used in the Boolean expression.
    - **TSC:** Conditional node. Based on a Boolean condition it either evaluates the true branch **TSE** or the false branch **TSE**.
  - **TSE** state:
    - **started_trans:** The number of started transactions as a sum of all producers which the TSE spans. Reset whenever the **TSE** is started. Always incremented while the **TSE** is running.
    - **ended_trans**: The number of ended transactions as a sum of all producers which the **TSE** spans. Reset whenever the **TSE** is started. Always incremented while the **TSE** is running.
  - **TSE** functions
    - **started(<*N*>):** Returns: False if the **TSE** has not started N transactions, True if it has started N transactions
    - **ended(<*N*>):** Returns: False if the **TSE** has not ended N transactions, True if it has ended N transactions
    - **terminated():** Returns: False if the **TSE** is active and True if it is terminated.
    - **terminate():** Terminates the **TSE**

With this refined model we can now create a refined version of the example in Figure 2. This is shown in Figure 4.

IV. DSL AND ABSTRACT SYNTAX TREE

To formalize this even further then the transaction sequence model allows us to define a DSL for the transaction sequence model. An example of this is given in Figure 5 which shows the **TSR** repetition transaction as BNF and the semantics of the related **TSE** functions.

However, to avoid the overhead of having to define a scanner and parser for the DSL then we deliberately choose to specify the transaction sequence model directly as an eDSL by defining an API for the nodes in the abstract syntax tree (**AST**). Hence, the **AST** can be represented directly by classes in Python. In the following **AST** definition, Python is used for the signature definition of the nodes (For simplicity only the weighted scheduler is defined):

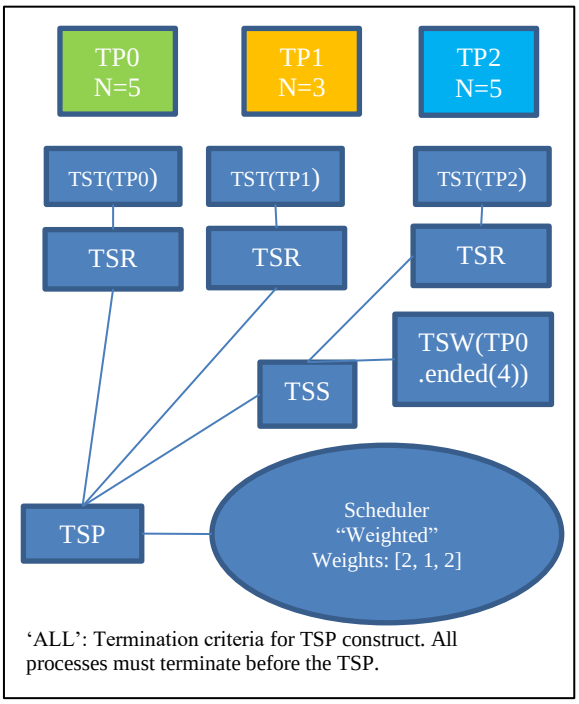

Figure 4: Example of a Transaction Sequence Model

`tp`(*name*, *N*)
Parameters:

- *name* (string) – The name of the transaction producer.
- *N* (integer) – The number of transactions to be produced. 0 indicates infinity.

Return type: Returns a **TP** instance. The **TP** functions described in section III are supported.

`scheduler_weight`(*name, weights*)
Parameters:

- *name* (string) – The name of the scheduler.
- *weights* (integer list) – The list of weights to be used by the scheduler. First element in the list is the weight for the first **TSE** in the *tselist* argument for the **TSP** node etc.

Return type: Returns a weighted scheduler instance.

`tst`(*producer*)
Parameters:

- *producer* (string) – The name of the transaction producer

Return type: Returns a **TST** node which produces a transaction from the specified producer. The **TSE** functions described in section III are supported.

`tss`(*tselist*)
Parameters:

- *tselist* (**TSE** node list) – the list of **TSE** nodes which shall be evaluated in sequence

Return type: Returns a **TSS** node. The **TSE** functions described in section III are supported.

`tsp`(*scheduler, tselist, termination_criteria*)
Parameters:

- *scheduler* (object) – The scheduler used to evaluate the parallel node.

- *tselist* (List of **TSE** nodes) – The **TSE** nodes which shall be evaluated in parallel.
- *termination_criteria* (Boolean expression) – The TSP terminates when *termination_criteria* evaluates to true.

Return type: Returns a **TSP** node. The **TSE** functions described in section III are supported.

`tsr`(*tse, N*)
Parameters:

- *tse* (**TSE** node) – Node to be evaluated N number of times.
- *N* (integer) – The number of iterations. 0 indicates infinity.

Return type: Returns a **TSR** node. The **TSE** functions described in section III are supported.

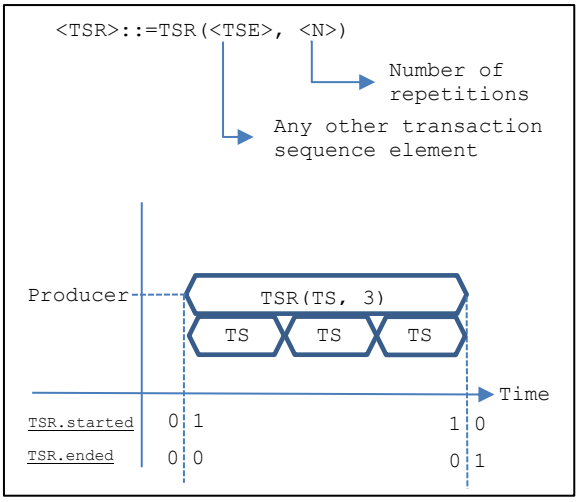

Figure 5: DSL for TSR repetition transaction

`tsw`(*wait_criteria*)
Parameters:

- *wait_criteria* (Boolean expression) – Boolean expression to be evaluated. *tse* is evaluated when *wait_criteria* is true.

Return type: Returns a **TSW** node. The **TSE** functions described in section III are supported.

`tsc`(*tset, tsef, conditional*)
Parameters:

- *tset* (**TSE** node) – Node to be evaluated when *conditional* is true.
- *tsef* (**TSE** node) – Node to be evaluated when *conditional* is false.
- *conditional* (Boolean expression) – Boolean expression to be evaluated. *tset* is evaluated when *conditional* is true and *tsef* is evaluated *conditional* is false.

Return type: Returns a **TSC** node. The **TSE** functions described in section III are supported.

V. Implementation Details

The implementation becomes simpler as we have no need for a scanner and parser as we can express the transaction sequence model directly as an **AST**. The consequence is that the implementation builds up the abstract syntax tree for the wanted bus traffic pattern using the **AST** nodes and then generates code from that instance directly. Thus, a Python representation is needed for each node in the AST. See Figure 6 for an example of the TP node.

```
class Base():
    def __init__(self, t, name = None):
        self.node_type      = t
        self.refs           = []
        self.name       = name

    def get_name(self):
        return self.name;

    def get_type(self):
        return self.node_type;

    def get_refs(self):
        return self.refs;

class tp(Base):

    def __init__(self, name, n = 0):
        Base.__init__(self, 'tp', name)
        self.max_iter = n

    def show(self, parent, label = None):
        s = '{}'.format(self.name)
        print(s);
        self.tool.add_node(self, parent, label)
```

Figure 6: Python Representation of the TP node

Our prototype generator in Python, targeting SystemVerilog/UVM, constructs the specified **AST** and then generates the following elements by using Python templates that contains the static parts of the SystemVerilog code:

- A SystemVerilog/UVM representation of the **AST** instance. The set of **AST** classes generated in SystemVerilog/UVM is **AST** instance specific.
- A synchronization object for synchronizing among transaction sequences.

Additionally, some reusable run-time SystemVerilog components are also needed, such as:

- Base classes for the **AST** nodes
- UVM virtual sequence starting the bus traffic pattern

- Interpretation algorithm

The **AST** is evaluated by traversing the nodes and getting the overall tree status which can be one of three things only:

- **WAITING**: Cannot produce a transaction now.
- **READY**: The next transaction can be produced and applied
- **TERMINATED**: The **AST** finished (Either terminated or it finished)

Thus, once the **AST** has been constructed by the UVM sequence then the evaluation algorithm is started, and it will evaluate the **AST** until it is **TERMINATED**.

This approach is not SystemVerilog/UVM specific. Thus, another back end can be implemented which for instance generates the same output but for SystemC instead and a SystemC based runtime environment can be used to evaluate the **AST**. Another solution could be to let the generator produce UVM sequences directly but since we also have a SystemC based verification platform to consider then the former was a better choice.

VI. EXAMPLE

The transaction sequence model example shown in Figure 4 can now be expressed as an **AST** in Python. This is shown in Figure 7. The Python code is then executed once within the verification flow. The `dsl` library contains the base classes for the **AST** nodes along with the generator implementation. The generator is executed via the `root.create_uvm()` which will generate the appropriate UVM code (Primarily `uvm_sequences` and the **AST** captured in a `uvm_object` data structure). A top uvm_sequence is also generated which need to be started by a `uvm_test`.

Furthermore, the `root.create_graph()` call generates the **AST** as a viewable graph. It needs to be rerun if the Python code or the generator is changed. The

```
"""
AST Example
"""

from dsl import *;

tp0 = tp0('tp0', 5)
tp1 = tp1('tp1', 3)
tp2 = tp2('tp2', 5)

sch    = scheduler_weight('WEIGHT', [2, 1, 2]);

tst0   = TST(tp0)
tst1   = TST(tp1)
tst1   = TST(tp2)

tsr0   = TSR(tst0)
tsr1   = TSR(tst1)
tsr2   = TSR(tst2)

tsw = TSW(tp0.ended() == 4)

tss = TSS([tsw, tsr2])

root = TSP(sch, [tsr0, tsr1, tss],
           tsr0.terminated() &&
           tsr1.terminated() &&
           tss.terminated())

root.create_graph('root');
root.create_uvm('root');
```

Figure 7: AST Example in Python

VII. RESULTS

When compared to PSS, the defined **AST** will not outperform PSS on vendor support, simulation platforms, debug, etc. but it allows the architect to express the traffic patterns required when targeting fine-grained transaction sequences on a single interface. Debugging is quite straight forward as the result will just be a series of transactions being applied to the bus interface. Also, because the implementation is an eDSL in Python, the maintenance of the software bundle is much easier.

Furthermore, the formalization of the problem has led to a tool which now can be used internally for expressing advanced traffic patterns on different buses.

The **AST** will also make it possible to generate other elements in addition to the actual traffic pattern, e.g. a graph visualizing the traffic pattern. It should be noted that the generated stimuli are portable as they will be related to a certain bus protocol and not to a specific Device Under Test (DUT).

VIII. CONCLUSIONS

Overall, the paper demonstrates that the ability to express the traffic patterns accurately directly improves the functional verification productivity and quality for RTL blocks significantly. It achieves this by providing architects and verification engineers with a portable and executable specification of the required traffic patterns.

Additionally, the framework can quite straight forward be extended to handle the limitations defined in section 3. Handling multiple sinks can be done my allowing multiple roots and/or by defining multiple **AST**s which can share **TSE** nodes. The scheduling can also be extended to handle other meta data than counting transactions. For instance, if scheduling is needed based on the amount of data then the **TSE** state and functions just needs to be extended with a state capturing this and functions for accessing the new information. The functions can then be used in the Boolean expressions.

Being able to handle constraints among transactions coming from the same transaction producer is more difficult as this would require support for a constraint language in the transaction sequence model. For now, this has been pushed to the evaluating layer. One way to solve this could of course be to allow the constraints to be written in clear text as SystemVerilog constraints and then add directly into the code. However, the solution would not be directly cross platform usable. This would require a constraint language which was generic.

Finally, releasing the generator to the open source community can be one way of allowing other verification engineers access to tool which would allow them to generate complex traffic patterns.

REFERENCES

[1] IEEE Standard 1800.2-2017, “IEEE Standard for Universal Verification Methodology Language Reference Manual”, 2017.

[2] M. Fowler, “Domain-specific Languages”, Pearson Education, 2010.

[3] C. A. R Hoare, “Communicating Sequential Processes”, Prentice Hall International, 1985.

[4] S. Jamil Quirem and P. Krishna Saravu, “Fake CPU: A Flexible and Simulation Cost-Effective UVC for Testing Shared Caches”, In Microprocessor and SOC Test and Verification (MTV), 2016 17th International Workshop on (pp. 1-6). IEEE, 2016.

[5] R. Edelman and R. Ardeishar, “Sequence, Sequence on the Wall – Who's the Fairest of Them All?”, Design and Verification Conference, 2013.

[6] A. K. Parekh and R. G. Gallager, “A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case”, IEEE/ACM Transactions on Networking. **1**(3), 344-357, 1993.

[7] A. van Dursen, P. Klint and J. Visser, “Domain Specific Languages: An Annotated Bibliography”, ACM SIGPLAN Notices. **45**(1), 26-36, 2000.

[8] Portable Test and Stimulus Standard, “PSS Early Adopter II (PSS EA II)”, Accellera Draft Standard, 2018.

[9] Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) protocol, ARM Ltd, 1996