DYNAMIC DATA STRUCTURES IN ASSERTIONS

BEN COHEN SYSTEMVERILOG.US

Introduction
The integration of dynamic data structures such as queues and associative arrays into SystemVerilog assertions has been a topic of ongoing discussion in the hardware verification community. While these constructs offer powerful capabilities for complex assertion writing, their implementation and support across various Electronic Design Automation (EDA) tools have been inconsistent. This paper addresses the challenges arising from the lack of uniform support for dynamic structures in SVA among tool vendors.

This paper examines the current landscape of assertion support in major EDA tools, highlighting the disparities in handling queues and associative arrays within assertions. This inconsistency raises important questions about the necessity and practicality of using such complex dynamic structures in assertions. Are these constructs truly indispensable for effective assertion-based verification, or do they introduce unnecessary complexity?

Despite these concerns, the potential benefits of using queues and associative arrays in certain verification scenarios cannot be ignored. Therefore, this paper explores various methods to support these data structures in assertions, even when direct tool support is lacking. I investigate techniques to implement efficient support logic, allowing verification engineers to leverage the power of queues and associative arrays in their assertions without sacrificing portability or performance. I demonstrate these techniques with a simulateable example using a queue and an associative array.

Furthermore, I explore the use of synthesizable logic as an alternative approach, which is particularly well-suited for formal verification environments. This method offers a more robust and tool-independent solution, potentially bridging the gap between simulation-based and formal verification methodologies.

1.1 ARE DYNAMIC DATA STRUCTURES USEFUL FOR SVA VERIFICATION?

In the verification of real-world designs using SVA, are there cases where dynamic data structures are truly necessary? There are indeed real design and system-level scenarios where using dynamic arrays, associative arrays, and/or queues in assertions can be valuable. For example:

1. Protocol Verification: In complex protocols with variable-length packets or out-of-order transactions, queues, or dynamic arrays, or associative arrays can help track and verify sequences of events.

2. Cache Coherency Checks: Associative arrays can model cache lines, allowing assertions to verify cache coherency protocols in multicore systems.

3. Resource Allocation Verification: Associative arrays can model resource pools (e.g., memory buffers, communication channels) to assert proper allocation and deallocation.

4. Scoreboarding: In testbenches, queues can efficiently implement scoreboards for tracking and checking multiple outstanding transactions. Queues are most versatile there due to many built-in methods.

Note that the aforementioned list of potential applications is more related to system-level performance analysis than to RTL design verification. However, one could argue that complex RTL design verification environments need these features. There are important considerations when using these dynamic structures, including:

1. Tool Support: Not all simulation tools fully support these structures in assertions, potentially limiting portability (*See 1.2.3).*

2. Performance Impact: The complexity is related to the processing required to obtain the sampled values from these complex data structures for use in assertions. This may significantly reduce simulation speed, especially for large designs.

3. Complexity vs. Clarity: Using complex data structures might make concurrent assertions harder to understand, debug, and maintain. However, smaller immediate assertions throughout scoreboards are easier to understand and maintain.

4. Alternative Approaches: In many cases, simpler assertions combined with auxiliary logic in the testbench might be more efficient and clearer.

1.2 QUEUES AND ASSOCIATIVE ARRAYS IN ASSERTIONS

1.2.1 What is a queue?

A queue is a variable-size unpacked array that supports constant-time insertion and removal at both the beginning and end of the array. Constant-time operations take the same amount of time regardless of the size of the data structure. Additionally, queues provide constant-time access to all their elements. Like classes, queues in SystemVerilog are statically declared in terms of their type and identifier. However, unlike class instances which require dynamic instantiation with the `new` keyword, queues do not need dynamic allocation. Queues are static in declaration but dynamic in size, meaning they can grow or shrink at runtime based on the elements added or removed. Their sizes can be expanded or reduced through method calls or assignments as defined in 1800’2023:7.10. For example:

```
module m;
 bit valid, clk, a;
 int q[$], data;
 initial begin
  $timeformat(-9, 2, " ns", 10);  // <---------
  q= { 2, 4, 8 };  // queue is initialized
  data=0;
  $display("%t q[0] 0x%0h, q[$] 0x%0h", $realtime, q[0], q[$]);
  //  0.00 ns  q[0] 0x2, q[$] 0x8  // first element, last element
  q.push_front(data+1); // Adds an element to the beginning of the queue
  #1  $display("%t q[0] 0x%0h, q[$] 0x%0h", $realtime, q[0], q[$]);
  //  1.00 ns q[0] 0x1, q[$] 0x8  // q[0] updated with the q.push_front(data+1
  @(posedge clk) data = q.pop_back(); // Removes and returns the last element of the queue.
  // data==8
  $display("%t q[0] 0x%0h, q[$] 0x%0h", $realtime, q[0], q[$]);
  //  10.00 ns q[0] 0x1, q[$] 0x4   // The 2nd element is now the last item in the queue
  #1 q[0] = data+2; // q[0]==8+2
  $display("%t q[0] 0x%0h, q[$] 0x%0h", $realtime, q[0], q[$]);
  //  11.00 ns q[0] 0xa, q[$] 0x4
  #10 $finish();
 end
 initial forever #10 clk=!clk;
endmodule
```

```
// $timeformat(units, precision, suffix_string, min_field_width)
// units = -9 means time in nanoseconds
// precision = 2 means two decimal places
// suffix_string = " ns" is appended after the time
// min_field_width = 10 specifies minimum width of time display
```

1.2.2 What is an associative array?

An associative array implements a lookup table of the elements of its declared type. The data type to be used as an index serves as the lookup key. An associative array is statically declared (unlike a class element with a `new`), but it is dynamically expanded or reduced in size through method calls or assignments as defined in 1800’2023:7.8. For example:

```
module memory_data_integrity_check ( input bit write, bit read,  bit[31:0] wdata,
                                              bit [31:0] rddata,  bit[31:0]  addr,  bit reset_n, clk);
  default clocking cb_clk @ (posedge clk);  endclocking
  int mem_aarray[int]; // associative array (AA) with typed-index to be used by property
  bit [31:0] raadata;  // data read from AA
  always@ (posedge clk) begin
    if (reset_n==1'b0) mem_aarray.delete; // Clears AA elements
    else if (write) mem_aarray[addr]  = wdata; // store data
  end

  //  Using an associative array to store the last data written
  // If a read and data exists at that address, then input rddata==expected_data (in the array)
  property p_read_after_write;
    (read && mem_aarray.exists(addr) |=> // @read and data there at same written addr
         rddata==mem_aarray[addr]; // read data from IO is same as what was written
  endproperty : p_read_after_write
endmodule : memory_data_integrity_check
```

1.2.3 What does IEEE 1800 say about dynamic structures in SVA?

16.6 Boolean expressions: “Elements of dynamic arrays, queues, and associative arrays that are sampled for assertion expression evaluation may get removed from the array or the array may get resized before the assertion expression is evaluated. These specific array elements sampled for assertion expression evaluation shall continue to exist within the scope of the assertion until the assertion expression evaluation completes.”

What does that mean?

This statement describes an important behavior of SystemVerilog assertions when dealing with dynamic data structures like dynamic arrays, queues, and associative arrays. Though most/all of the following points apply to simpler structures (e.g., bit, int) they represent a higher level of complexity unique to dynamic structures.

1. Sampling: When an assertion involves elements from dynamic data structures, the simulator takes a "snapshot" of the relevant elements at the sampling point (typically at a clock edge).
2. Persistence: Even if the original data structure is modified after sampling (e.g., elements are removed or the array is resized), the original sampled values are preserved for the assertion evaluation.
3. Scope: These sampled values are kept in a separate, temporary scope that exists only for the duration of the assertion evaluation.
4. Evaluation integrity: This mechanism ensures that the assertion is evaluated based on the state of the data structures at the time of sampling, regardless of any changes that occur between sampling and evaluation.

The simulator doesn't necessarily make a full copy of the entire data structure. Instead, it likely maintains references or copies of only the specific elements needed for the assertion. This approach allows for accurate and consistent assertion checking, even when dealing with dynamic data structures that might change during simulation. This behavior is crucial for maintaining the correctness and predictability of assertions, especially in complex designs where data structures may be modified frequently or asynchronously with respect to the assertion evaluation.

Creating snapshots to obtain sampled values from dynamic data structures demonstrates the complexities inherent in simulation tools. This process not only highlights the sophisticated mechanisms required but also raises concerns about performance impacts, especially when dealing with large data structures.

Many vendors were slow to adopt this IEEE 1800 requirement. Several imposed restrictions on the use of queues and associative arrays within assertions, and some prohibited them entirely, likely due to performance concerns. These dynamic structures are fully adopted for SystemVerilog (non-SVA) applications, but nonblocking assignment is restricted by some tools. Types of error messages observed when they are used is assertions include:

int q[$] = { 2, 4, 8 }, data=0; // queue
int mem_aarray[int]; // associative array (AA) to be used by property *(See note 3)*.

- $rose(a) |-> **q[$]**==8'h08;
  ERROR: *this context is not supported.*
- ($rose(a), v = b) |-> ##[3:10] **q[v];**
  ERROR: *Expressions involving real, string, event, tagged union and dynamic SystemVerilog types are not allowed in Boolean expressions in properties and sequences.*
- ($rose(a) |-> (q.**pop_back** == 8'h08)
  ERROR: *The method 'pop_back' is not allowed in assertions as it has side effects*
  Expressions involving real, string, event, tagged union and dynamic
  ERROR**:** *SystemVerilog types are not allowed in Boolean expressions in properties and sequences.*
- Initial q[0] <= 12;
  ERROR**:** *Non-blocking assignment to elements of dynamic arrays is not currently supported*
- mem_aarray*[10] <= 32'hFFFF0000;*
  ERROR: *Non-blocking assignment to elements of associative arrays is not currently supported.*
- ((valid, v_err=mem_aarray.**exists**(addr))
  ERROR: *Expressions involving real, realtime, string, event and dynamic SystemVerilog types including virtual interface references are not allowed in local variable assignments in properties and sequences*
- **assign** mem_aarray_exists = mem_aarray.**exists**(addr) ; // should be legal
  ERROR: *Associative array may not be used in non-procedural context.*
  Note: the tool accepted the following instead
  always_comb mem_aarray_exists = mem_aarray.exists(addr); // OK too

1.2.4 Static structures versus dynamic structures in assertions

As of today, formal tools do not support dynamic storage such as dynamic arrays, associative arrays and queues. Typically, these structures are heavily used for simulation in class-based testbenches where concurrent assertions are illegal. In simulation, the use of dynamic structures in assertions is not without challenges:

1. A significant issue, as mentioned earlier, is the lack of uniform support across simulation tools. This inconsistency potentially limits code portability and may introduce compatibility issues in multi-vendor verification environments. To mitigate these issues and reduce the likelihood of bugs caused by tool inconsistencies, users should consider adopting a "*Build Your Own Linter*" (BYOL) approach. Open-source initiatives can lead the way in this area, potentially offering more flexible and customizable solutions than commercial EDA tools. This approach allows verification teams to create tailored lint rules that enforce consistent coding practices across different tools, thereby improving code portability and reducing the risk of tool-specific errors.

2. The use of dynamic structures can impact simulation performance, especially when dealing with large-scale designs or extensive test suites.

Given these considerations, two alternative approaches warrant exploration:

1.2.4.1 Static Structures Replacing the Dynamic Structures:

For assertions, instead of relying on dynamic structures within assertions, would it be more effective to utilize static structures? Thus, instead of a dynamic array, use a static fixed-sized memory. Also, instead of a queue, use a fixed-sized FIFO. This method would involve constraining the model to an acceptable set of parameters, carefully chosen to represent the most critical scenarios without compromising verification integrity. This method also creates a synthesizable environment suitable for formal verification. Key considerations include:

- Determining the optimal size for static structures to balance coverage and performance
- Developing strategies to ensure that the constrained model adequately represents the full range of system behaviors.
- Assessing the impact on verification completeness and the potential need for additional test cases.

1.2.4.2 Support Logic with Static Variables Replacing Function Calls

Alternatively, we could still use the dynamic structures within the testbench, but we would implement dedicated support logic to address the issue of using the needed elements of those structures in the assertions. This approach involves creating a mechanism to copy the results from functions that access dynamic structures into static variables. These static variables would then serve as proxies for the dynamic data in our assertions. The process would work as follows:

1. Define static variables that correspond to the key data points we wish to use in the assertions.
2. In support logic, use the dynamic structures' functions to update the static variables accordingly.
3. Use the static variables in assertions instead of directly referencing the dynamic structures.

For example,

SUPPORT LOGIC	**EXAMPLES**

logic[31:0] raadata; // data read from AA
bit aa_exists; // exists at specified address
logic[31:0] q_size;
logic[31:0] q_front, q_back;
 	
always @(addr) begin
aa_exists = aa.exists(addr);
if (aa_exists) begin
raadata = aa[addr];
end
end
	// Instead of
	aa.exists(addr) |-> aa[addr]==data;
	
	// USE
	aa_exists |-> raadata==data;

always @(posedge clk) begin
if(pop && !push) begin
q_back=q.pop_back();
q_size = q.size();
end
.....
 	
end
	int q_size, q_data; *// Updated with support logic*
	// instead of
	$rose(a) \|-> ##1 q.size() != 0 && q[$]==data;
	// USE
	$rose(a) |-> ##1 q_size != 0 && q_back==data;

By employing these approaches, we can circumvent tool issues related to having dynamic structures within the assertions while still maintaining the ability to verify their behavior in both simulation and formal verification.

These alternative approaches aim to address the challenges posed by dynamic structures while maintaining robust verification capabilities. By exploring these options, we can potentially develop more portable, efficient, and widely compatible assertion-based verification strategies.

1.3 AVOIDING QUEUES AND ASSOCIATIVE ARRAYS IN ASSERTIONS

To limit the size and scope of the testbenches, several alternatives to dynamic structures are used. They are presented here because they offer effective techniques for addressing the challenges. It's a different way of looking at and processing the verification environment.

1. Using Arrays Instead of Associative Arrays:

Section 1.2.4.1 addressed the case of using fixed-size arrays with a known index range. This is synthesizable and can be used in formal verification. For simulation, dynamic structures are preferred

2. Using Encoders/Decoders:

Encoders and decoders can be used for verification in the following ways:

1. Testbench Components:

You can create encoder and decoder modules as part of your testbench to generate stimulus and check responses. For example, you could use an encoder to convert test vectors into a more compact form for input to your DUT, and a decoder to expand the DUT output for easier comparison with expected results.

2. Self-Checking Testbenches:

Implement complementary encoder-decoder pairs in your testbench. Use an encoder to generate inputs for your DUT, then use a decoder on the DUT output and compare it with the original input to verify correct functionality.

Here is an example to map certain select integer values to encoded numbers.

```
module string_to_value_mapper(
  input logic [7:0], int addr_key // need to select up to 256 addr values
               // for the TB instead of all 2^31 possible values
  output logic [3:0] value);
  always_comb begin
    case (addr_key)
      32’h0000_0000: value = 4'd0;
      32’h1111_1111: value = 4'd1;
      32’hF0F0_F0F0: value=4’d2;
      32’h0F0F_oF0F: value=4’d3;
      // ... more mappings ...
      default: value = 4'd15;  // Unknown key
    endcase
  end
endmodule
```

3. Using Look-Up Tables (LUTs):

For small mappings, you can use LUTs implemented as ROM.

```
module lut_mapper(  input logic [3:0] key,
  output logic [7:0] value);
  logic [7:0] lut[16];
  initial begin
    lut[0] = 8'h12;
    lut[1] = 8'h34;
    // ... initialize other values ...
    lut[15] = 8'hFF;
  end
  assign value = lut[key];
endmodule
```

These approaches have some limitations compared to associative arrays:

- Fixed size: You need to know the maximum size in advance.
- Less flexible: Adding new mappings may require modifying the RTL.
- Potentially larger hardware: For sparse mappings, you might use more resources.

However, they offer the advantage of being synthesizable and usable in formal verification environments. This method offers a more robust and tool-independent solution, potentially bridging the gap between simulation-based and formal verification methodologies. The choice between these methods and associative arrays depends on your specific verification needs, the size of the data set, and whether synthesizability is a requirement.

1.4 Associative Array Application EXAMPLE

The purpose of this example is twofold:

1. To show how associative arrays can be used in assertions using native features. Note that these features may not be supported by all vendors.

2. To demonstrate an alternative approach using support logic that is compatible with all vendors because it uses support logic to replace the dynamic structures' built-in features. In Figure 1, we present a transaction model for a memory device. This model incorporates the following elements:

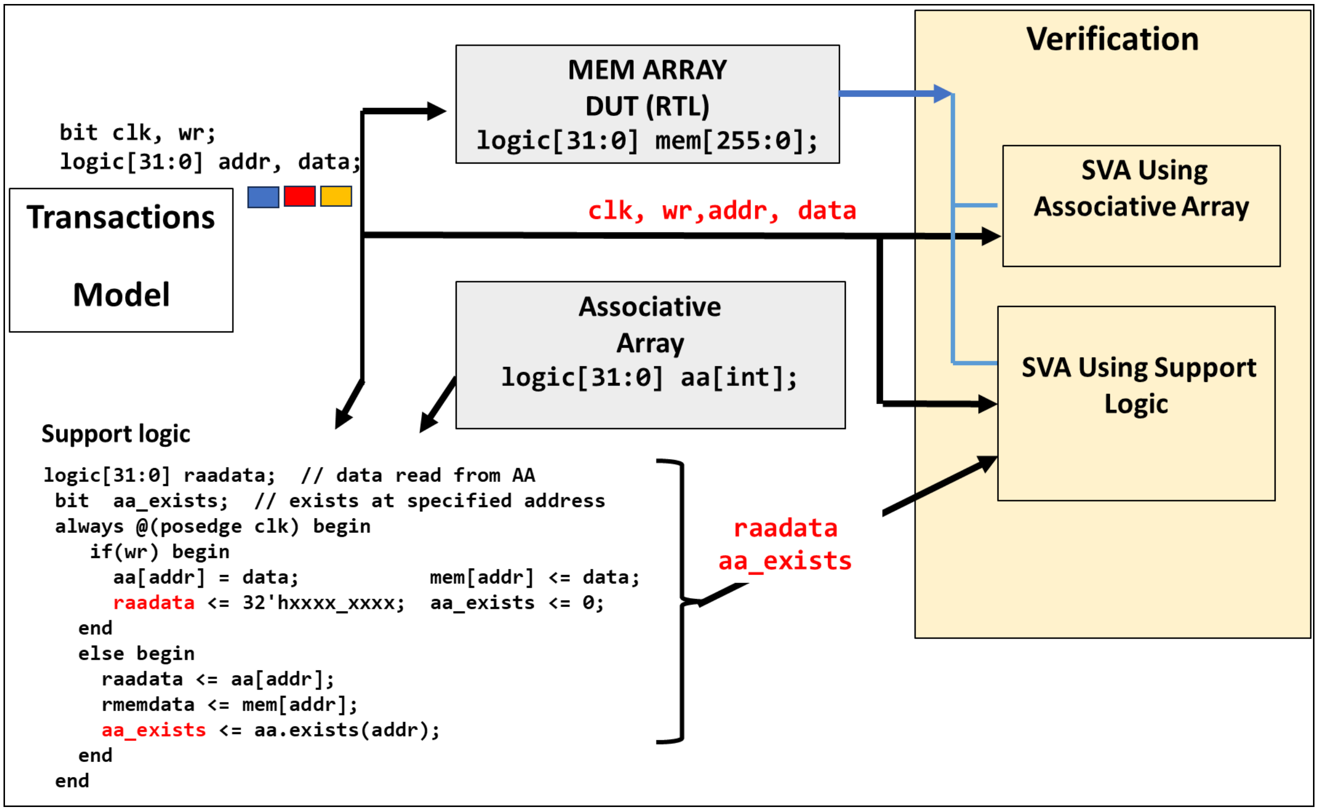
A write command (`wr`), a 32-bit data (`data`) to be written, and a 32-bit address (`addr`). The data is stored in an RTL memory within a Device Under Test (DUT). The DUT memory is smaller in size compared to the associative array used for verification.

The verification process includes two optional assertions:
1. An assertion utilizing built-in associative array functions.
2. An assertion employing support logic to store necessary associative function data in module variables.
The second option enhances portability by directly using these variables in the assertions. The requirements for this assertion include:

1. Write-Read Consistency: If data is written to an address, any subsequent read from that address should return the same data.

2. Multiple Write Handling: If a second write occurs to the same address before a read operation, the most recent write should be considered for evaluation.

This example demonstrates how support logic can be used to replace the built-in features of associative arrays. By storing the required information in module variables, the assertions become more portable and can be easily adapted to different verification environments.



```
bit clk, wr;
logic[31:0] addr, data;
```

```
logic[31:0] raadata;  // data read from AA
 bit  aa_exists;  // exists at specified address
 always @(posedge clk) begin
    if(wr) begin
      aa[addr] = data;            mem[addr] <= data;
      raadata <= 32'hxxxx_xxxx;   aa_exists <= 0;
   end
   else begin
     raadata <= aa[addr];
     rmemdata <= mem[addr];
     aa_exists <= aa.exists(addr);
   end
 end
```

Figure 1. Associative Array Model for SVA Testing

1. **Assertion using associative array information**:

```
property p_aawr;
  int vaddr;
  @ (posedge clk)  (wr, vaddr=addr)  ##1
     (!(addr==vaddr)[*0:$] ##1 (!wr && addr==vaddr))   |->
           ##1 aa.exists(vaddr) && aa[addr]==mem[addr];
endproperty
```

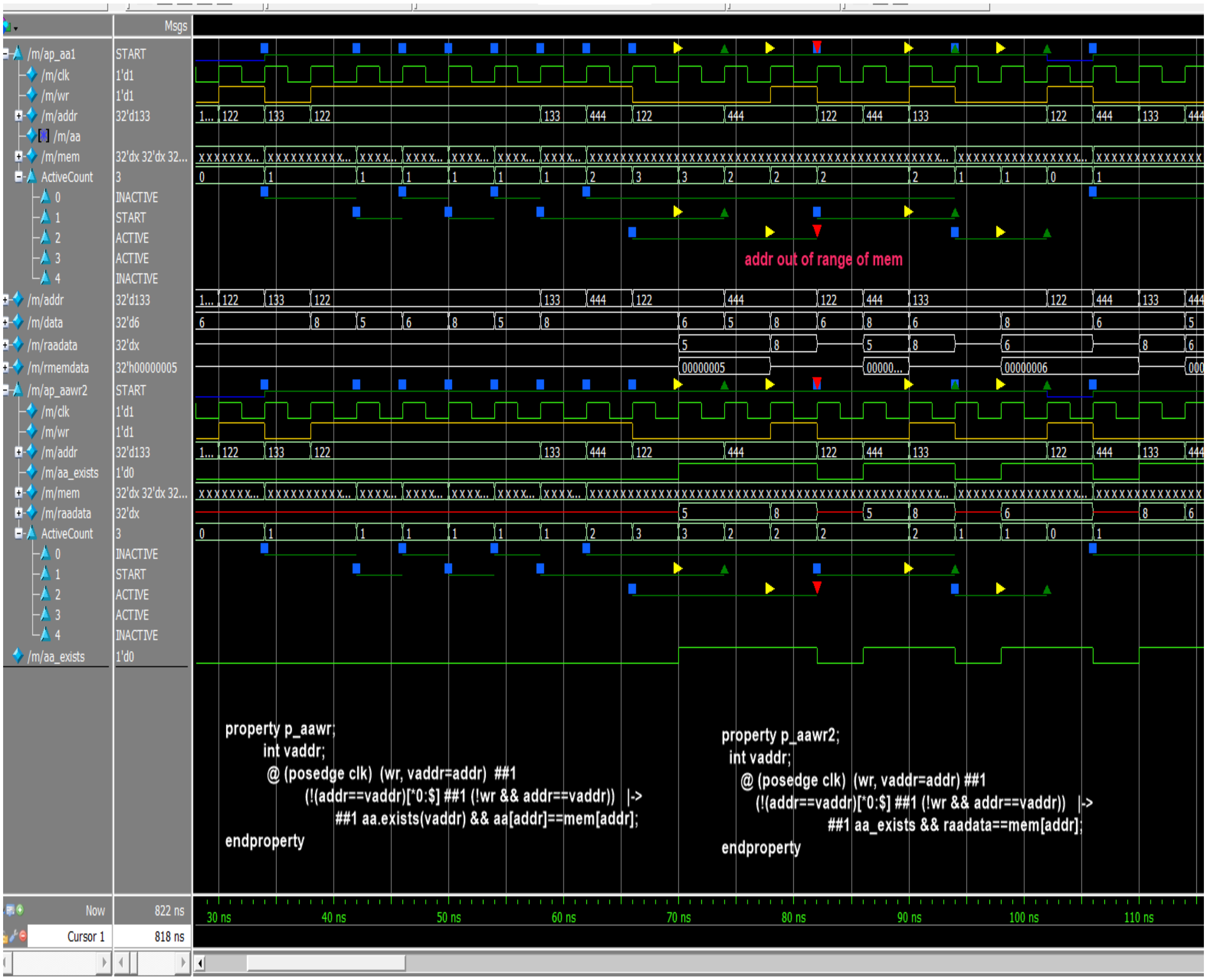
The property states:

1. If `wr`, store the address into local variable `vaddr`
2. If that same address occurs then it is a read or a write.
   a. If it is a write then this property is vacuous. Another thread will be initiated for that 2nd write.
   b. If it is a read, the associative array for that written item exists and its content is what is expected.

2. **Using support logic to extract the associative array information into the module variables.**

```
 property p_aawr2;
   int vaddr;
   @ (posedge clk)  (wr, vaddr=addr) ##1
      (!(addr==vaddr)[*0:$] ##1 (!wr && addr==vaddr))   |->
            ##1 aa_exists && raadata==mem[addr];
endproperty
```

Figure 2. SVA Code



	Msgs
/m/ap_aa1	START
/m/clk	1'd1
/m/wr	1'd1
/m/addr	32'd133
/m/aa	
/m/mem	32'dx 32'dx 32...
ActiveCount	3
0	INACTIVE
1	START
2	ACTIVE
3	ACTIVE
4	INACTIVE
/m/addr	32'd133
/m/data	32'd6
/m/raadata	32'dx
/m/rmemdata	32'h00000005
/m/ap_aawr2	START
/m/clk	1'd1
/m/wr	1'd1
/m/addr	32'd133
/m/aa_exists	1'd0
/m/mem	32'dx 32'dx 32...
/m/raadata	32'dx
ActiveCount	3
0	INACTIVE
1	START
2	ACTIVE
3	ACTIVE
4	INACTIVE
/m/aa_exists	1'd0
Now	822 ns
Cursor 1	818 ns

addr out of range of mem

```
property p_aawr;
        int vaddr;
         @ (posedge clk)  (wr, vaddr=addr)  ##1
                (!(addr==vaddr)[*0:$] ##1 (!wr && addr==vaddr))   |->
                      ##1 aa.exists(vaddr) && aa[addr]==mem[addr];
endproperty
```

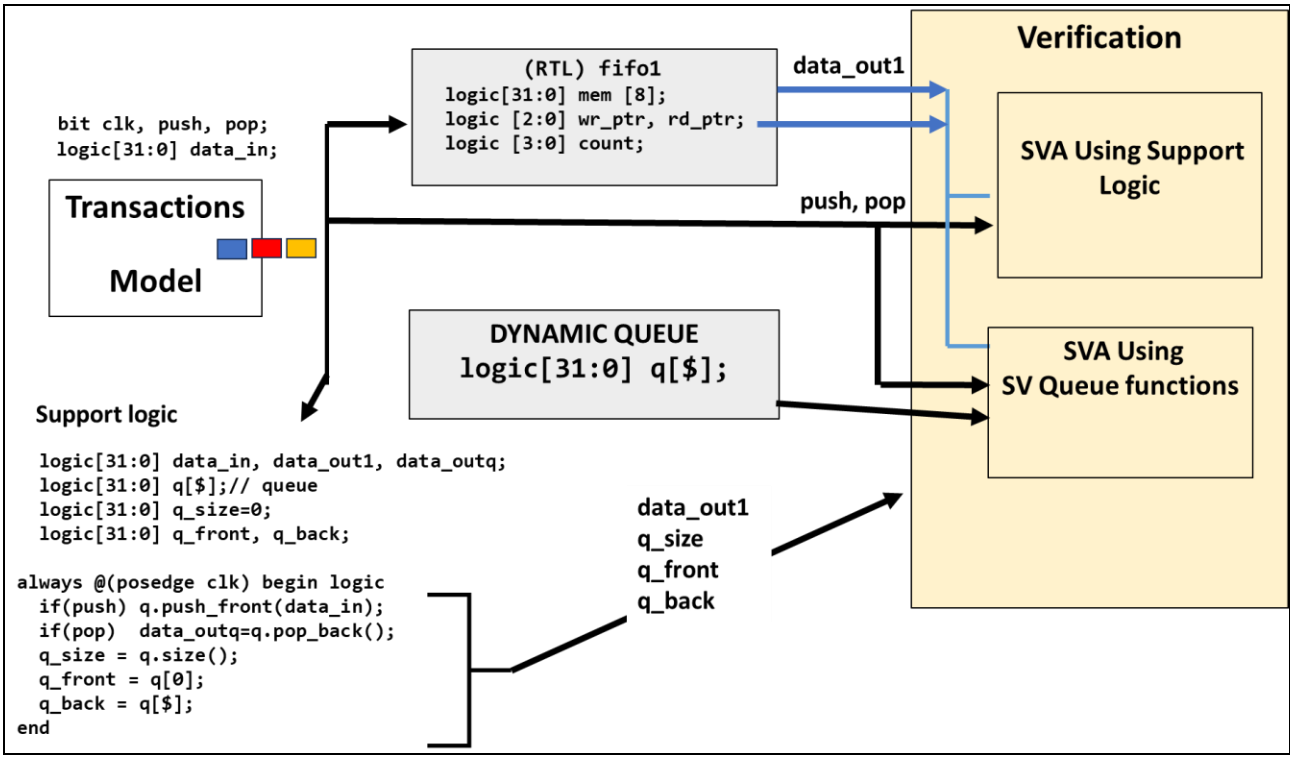
```
property p_aawr2;
  int vaddr;
     @ (posedge clk)  (wr, vaddr=addr) ##1
        (!(addr==vaddr)[*0:$] ##1 (!wr && addr==vaddr))   |->
                    ##1 aa_exists && raadata==mem[addr];
endproperty
```

Figure 3. Simulation Results for Associative Array Application

1.5 QUEUE APPLICATION EXAMPLE

This example demonstrates the applications of queues within assertions. Since not all tools support these dynamic structures, supporting logic demonstrates available options. In this example, we have a FIFO RTL driven by an external source with push and pop commands. The assertions use a queue to emulate the FIFO.

Requirements: If the RTL FIFO receives a `push` and no `pop`, or a `pop` and no `push`, then the RTL FIFO is operating correctly and is in sync with a pseudo-random mix of pushes and pops. These requirements are limited in this paper because the goal is to demonstrate concepts in utilization and not the verification of the FIFO.



```
bit clk, push, pop;
logic[31:0] data_in;
```

```
(RTL) fifo1
logic[31:0] mem [8];
logic [2:0] wr_ptr, rd_ptr;
logic [3:0] count;
```

```
DYNAMIC QUEUE
logic[31:0] q[$];
```

```
logic[31:0] data_in, data_out1, data_outq;
logic[31:0] q[$];// queue
logic[31:0] q_size=0;
logic[31:0] q_front, q_back;

always @(posedge clk) begin logic
  if(push) q.push_front(data_in);
  if(pop)  data_outq=q.pop_back();
  q_size = q.size();
  q_front = q[0];
  q_back = q[$];
end
```

Figure 4. Queue Model for SVA Testing

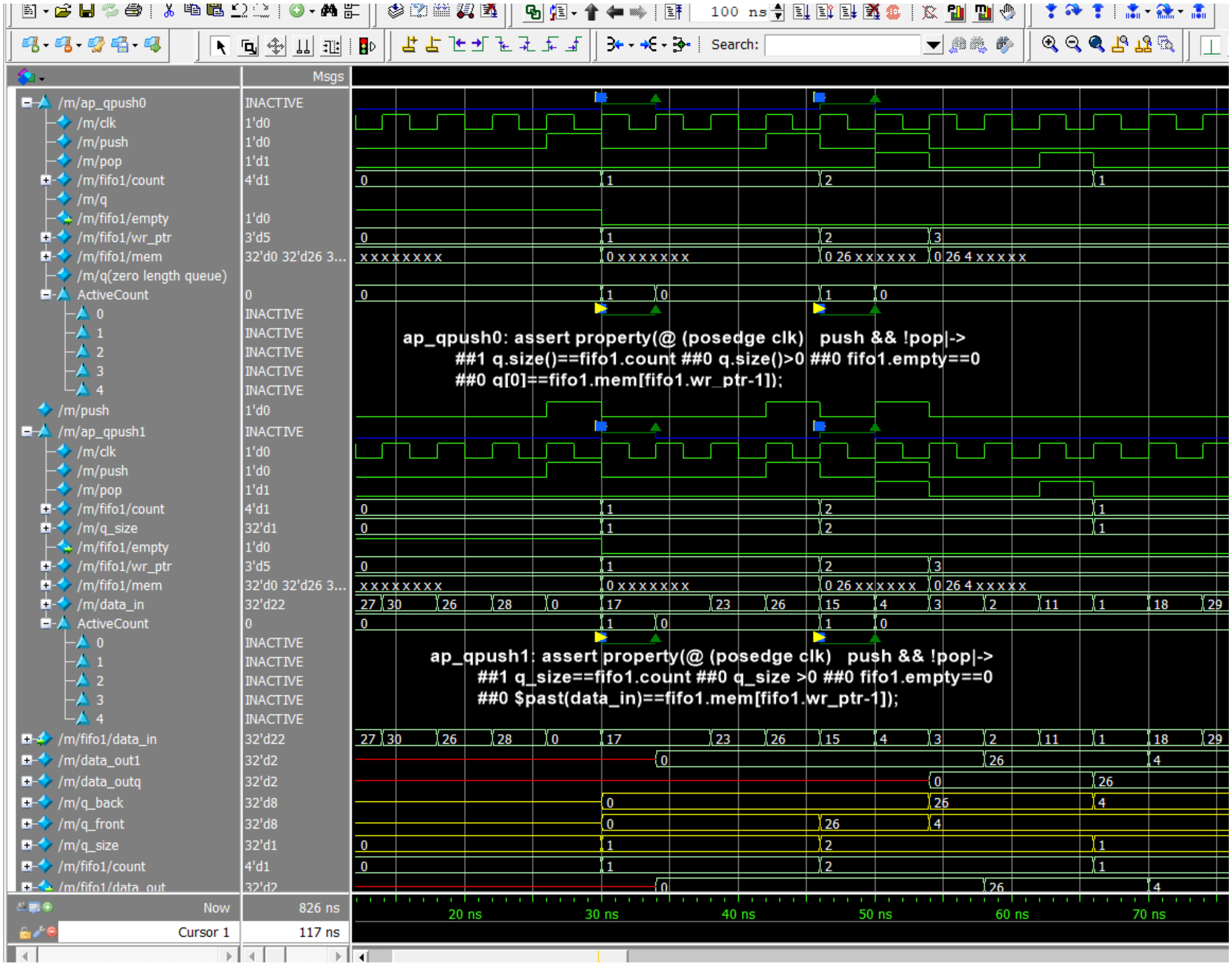
```
 // requirements: If  a push then at the next cycle q[0]== pushed rtl pushed value
// Also, the RTL fifo size is operational as expected, i.e., if push and no pop then
// its size increases by 1.  If a pop and no push then its size decreases by 1.
    ap_qpush0: assert property(@ (posedge clk)   push && !pop|->
          ##1 q.size()==fifo1.count ##0 q.size()>0 ##0 fifo1.empty==0
          ##0 q[0]==fifo1.mem[fifo1.wr_ptr-1]);

    ap_qpush1: assert property(@ (posedge clk)   push && !pop|->
          ##1 q_size==fifo1.count ##0 q_size >0 ##0 fifo1.empty==0
          ##0 $past(data_in)==fifo1.mem[fifo1.wr_ptr-1]);

    ap_qpop0: assert property(@ (posedge clk)   !push && pop|->
          ##1 q.size()==fifo1.count ##0 $past(q.size()) > 0
          ##0 $past(q[$])==data_out1 );

    ap_qpop1: assert property(@ (posedge clk)   !push && pop|->
          ##1 q_size==fifo1.count ##0 $past(q_size) > 0
          ##0 data_outq==(data_out1));
```

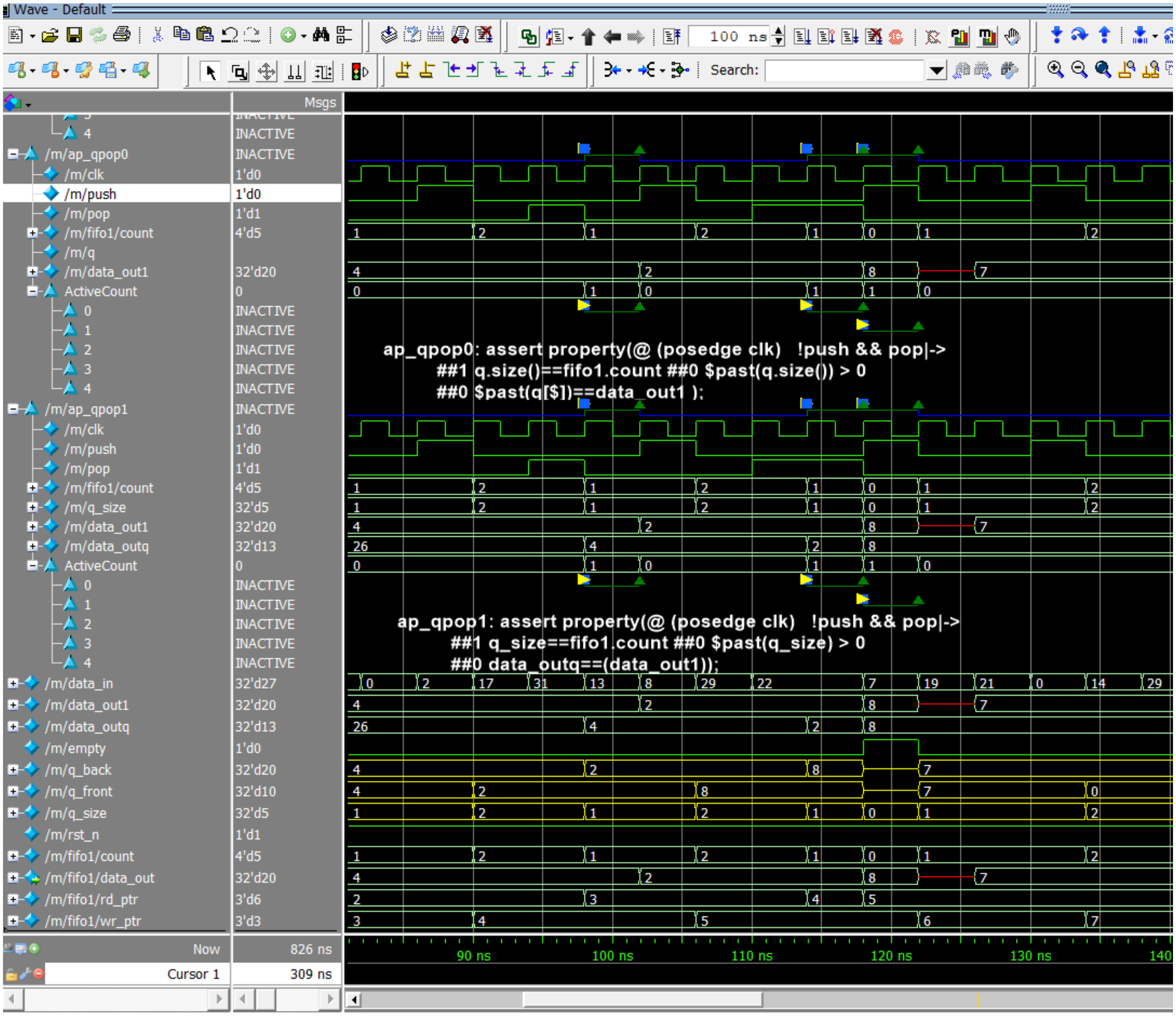
Figure 4. SVA Code



```
ap_qpush0: assert property(@ (posedge clk)  push && !pop|->
        ##1 q.size()==fifo1.count ##0 q.size()>0 ##0 fifo1.empty==0
        ##0 q[0]==fifo1.mem[fifo1.wr_ptr-1]);
```

```
ap_qpush1: assert property(@ (posedge clk)  push && !pop|->
        ##1 q_size==fifo1.count ##0 q_size >0 ##0 fifo1.empty==0
        ##0 $past(data_in)==fifo1.mem[fifo1.wr_ptr-1]);
```

Figure 5. Simulation Results for Queue push



```
ap_qpop0: assert property(@ (posedge clk)   !push && pop|->
    ##1 q.size()==fifo1.count ##0 $past(q.size()) > 0
    ##0 $past(q[$])==data_out1 );
```

```
ap_qpop1: assert property(@ (posedge clk)   !push && pop|->
    ##1 q_size==fifo1.count ##0 $past(q_size) > 0
    ##0 data_outq==(data_out1));
```

Figure 6. Simulation Results for Queue pop

1.6 Conclusion

While there are legitimate use cases for dynamic data structures in assertions, their use should be carefully considered. The decision should balance the need for complex checking against factors like performance, tool support, and maintainability. In many cases, a combination of simpler assertions and well-designed testbench logic may provide a more robust and efficient solution. Ultimately, the goal is to create effective, efficient, and maintainable verification environments. Dynamic structures in assertions can be powerful tools when used judiciously, but they should not be employed without careful consideration of alternatives and potential drawbacks.

Notes, links, and afterthoughts

Code:

```
https://systemverilog.us/vf/queue_1027b.sv
https://systemverilog.us/vf/aa_test1022c.sv
```

Acknowledgments:

I thank Ajeetha Kumari for providing valuable comments about this topic. Ajeetha is a co-authors of my book *SystemVerilog Assertions Handbook Revised* $^{4\ th}$ *edition 2023.* Ajeetha is currently driving SysetmVerilog testbench linting at AsFigo AI (https://www.linkedin.com/in/asfigo-ai-307900263/ https://www.linkedin.com/in/ajeetha/)

I also thank Ed Cerny providing feedback (https://www.linkedin.com/in/eduard-cerny-a40901/). Ed is a co-author of the book *SVA: The Power of Assertions in SystemVerilog* ^{2nd} *Edition* and was a member of the IEEE committee, representing Synopsys, for the definition of SystemVerilog Assertions.

I used Perplexity AI to assist with research, idea expansion, and code examples for this paper. The critical analysis, discussion of practical implications, and proposed solutions are my original contributions. This approach combines AI-assisted research efficiency with human insight and domain expertise, resulting in a thorough examination of queues and associative arrays in SystemVerilog Assertions.

Afterthoughts

Industry can benefit by doing custom Lint around such issues. AsFigo AI is will soon have SVALint in limited form available as opensource. Sites of interest:

https://asfigo.com/products
https://asfigo.blogspot.com/2023/10/dont-go-wild-with-associative-arrays-in.html
https://github.com/AsFigo/SVALint/issues

Good examples of assertion usage using Qs.
https://github.com/lowRISC/opentitan/blob/master/hw/vendor/lowrisc_ibex/dv/uvm/icache/dv/env/ibex_icache_scoreboard.sv
https://github.com/lowRISC/opentitan/blob/10f1ee5d5ad80de5eeb9f7c37dd58b6b1e2a600a/hw/vendor/lowrisc_ibex/dv/uvm/icache/dv/env/ibex_icache_scoreboard.sv#L162

The OVL library of assertion checkers has useful information. It is intended to be used by design, integration, and verification engineers to check for good/bad behavior in simulation, emulation, and formal verification. The OVL standard includes the *OVL V2 Library Reference Manual*.
https://www.accellera.org/downloads/standards/ovl

You may also be interested in papers and books that I wrote, many are now donated. These address design methodologies in the field of documentation and planning, VMM, and SVA understanding and utilization.
http://systemverilog.us/vf/Cohen_Links_to_papers_books.pdf