12.14 Random sequence generation - randsequence

Parser generators, such as yacc, use a Backus-Naur Form (BNF) or similar notation to describe the grammar of the language to be parsed. The grammar is thus used to generate a program that is able to check if a stream of tokens represents a syntactically correct utterance in that language. SystemVerilog's sequence generator reverses this process: it uses the grammar to randomly create a correct utterance (i.e., a stream of tokens) of the language described by the grammar. The random sequence generator is useful for randomly generating structured sequences of stimulus such as instructions or network traffic patterns.

The sequence generator uses a set of rules and productions within a **randsequence** block. The syntax of the **randsequence** block is:

```
statement ::= [ block_identifier : ] statement_item                    // from Annex A.6.4

statement_item ::=
                ...
              | { attribute_instance } randsequence

randsequence ::=  randsequence  ( [ production_ identifier ] )         // from Annex A.6.12
                        { rs_type_declaration }
                        production { production }
                        endsequence

rs_type_declaration ::= type_identifier  production_ identifier { , production_ identifier } ;

production ::= production_name [ ( tf_port_list ) ] : rs_rule { | rs_rule } ;

rs_rule ::= rs_production_list [ :=  expression ] [ rs_code_block ]

rs_production_list ::=
                 rs_prod { rs_prod }
               | rand join [ (expression) ] production_item  production_item { production_item }

rs_ code_block ::= {  { block_data_declaration } { statement_or_null } }

rs_prod ::=
                 production_item
               | rs_code_block
               | rs_if_else
               | rs_repeat
               | rs_case

production_item ::= production_identifier [ ( list_of_arguments ) ]

rs_if_else ::= if  ( expression ) production_item [ else production_item ]

rs_repeat ::= repeat ( expression ) production_item

rs_case ::= case ( expression )  rs_case_item { rs_case_item } endcase

rs_case_item ::=
                    expression { , expression } : production_item
                  | default [:]   production_item
```

Syntax 12-9 —Randsequence syntax (excerpt from Annex A)

A **randsequence** grammar is composed of one or more productions. Each production contains a name and a list of production items. Production items are further classified into terminals and non-terminals. Non-terminals are defined in terms of terminals and other non-terminals. A terminal is an indivisible item that needs no further definition than its associated code block. Ultimately, every non-terminal is decomposed into its terminals. A production list contains a succession of production items, indicating that the items must be streamed in sequence. A single production may contain multiple production lists separated by the | symbol. Production lists separated by a | imply a set of choices, which the generator will make at random.

A simple example illustrates the basic concepts:

```
randsequence( main )
    main    : first second done ;
    first   : add | dec ;
    second  : pop | push ;
    done    : { $display("done"); } ;
    add     : { $display("add");  } ;
    dec     : { $display("dec");  } ;
    pop     : { $display("pop");  } ;
    push    : { $display("push"); } ;
endsequence
```

The production *main* is defined in terms of three non-terminals: *first, second*, and *done*. When *main* is chosen, it generates the sequence, *first, second*, and *done*. When *first* is generated, it is decomposed into its productions, which specifies a random choice between *add* and *dec*. Similarly, the *second* production specifies a choice between *pop* and *push*. All other productions are terminals; they are completely specified by their code block, which in the example displays the production name. Thus, the grammar leads to the following possible outcomes:

```
add pop done
add push done
dec pop done
dec push done
```

When the **randsequence** statement is executed, it generates a gramar-driven stream of random productions. As each production is generated, the side effects of executing its associated code blocks produce the desired stimulus. In addition to the basic grammar, the sequence generator provides for random weights, interleaving and other control mechanisms.

The **randsequence** statement creates a scope. All production identifiers are local to the scope. In addition, each code block within the **randsequence** block creates an anonymous scope. The **randsequence** keyword may be followed by an optional production name (inside the parenthesis) that designates the name of the top-level production. If unspecified, the first production becomes the top-level production.

12.14.1 Random production weights

The probability that a production list is generated can be changed by assigning weights to production lists. The probability that a particular production list is generated is proportional to its specified weight.

```
production ::= production_name [ ( tf_port_list ) ] : rs_rule { | rs_rule } ;

rs_rule ::= rs_production_list [ := expression ] [ rs_code_block ]
```

The `:=` operator assigns the weight specified by the expression to its production list. Weight expression must evaluate to integral non-negative values. A weight is only meanigful when assigned to alternative productions, that is, production list separated by a |. Weight expressions are evaluated when their enclosing production is selected, thus allowing weights to change dynamically. For example, the *first* production of the previous example can be re-written as:

```
first   : add := 3
        | dec := 2
        ;
```

This defines the production *first* in terms of two weighted production lists *add* and *dec*. The production *add* will be generated with 60% probability and the production *dec* will be generated with 40% probability.

If no weight is specified, a production shall use a weight of 1. If only some weights are specified, the unspecified weights shall use a weight of 1.

12.14.2 If..else production statements

A production can be made conditionally by means of an **if..else** production statement. The syntax of the **if..else** production statement is:

rs_if_else ::= **if (** expression **)** production_item [**else** production_item]

The expression can be any expression that evaluates to a boolean value. If the expression evaluates to true, the production following the expression is generated, otherwise the production following the optional **else** statement is generated. For example:

```
randsequence()
    ...
    PP_PO : if ( depth < 2 ) PUSH else POP ;
    PUSH  : { ++depth; do_push(); };
    POP   : { --depth; do_pop(); };
endsequence
```

This example defines the production PP_OP. If the variable depth is less than 2 then production PUSH is generated, otherwise production POP is generated. The variable depth is updated by the code blocks of both the PUSH and POP productions.

12.14.3 Case production statements

A production can be selected from a set of alternatives using a **case** production statement. The syntax of the **case** production statement is:

rs_case ::= **case (** expression **)** rs_case_item { rs_case_item } **endcase**

rs_case_item ::=
expression { **,** expression } **:** production_item
| **default** [**:**] production_item

The case expression is evaluated, and its value is compared against the value of each case-item expression, which are evaluated and compared in the order in which they are given. The production associated with the first case-item expression that matches the case expression is generated. If no matching case-item expression is found then the production associated with the optional **default** item is generated, or nothing if there no **default** item. Case-item expressions separated by commas allow multiple expressions to share the production. For example:

```
randsequence()
    SELECT : case ( device & 7 )
               0       : NETWORK
               1, 2    : DISK
               default : MEMORY
             endcase ;
    ...
endsequence
```

This example defines the production SELECT with a case statement. The case expression (`device & 7`) is evaluated and compared against the two case-item expressions. If the expression matches 0, the production NETWORK is generated, and if it matches 1 or 2 the production DISK is generated. Otherwise the production MEMORY is generated.

12.14.4 Repeat production statements

The **repeat** production statement is used to iterate over a production a specified number of times. The syntax of the **repeat** production statement is:

```
rs_repeat ::= repeat ( expression ) production_item
```

The repeat expression must evaluate to a positive integral value. That value specifies the number of times that the corresponding production is generated. For example:

```
randsequence()
    ...
    PUSH_OPER : repeat( $urandom_range( 2, 6 ) ) PUSH ;
    PUSH  : ...
endsequence
```

In this example the PUSH_OPER production specifies that the PUSH production be repeated a random number of times (between 2 and 6) depending on by the value returned by $urandom_range().

12.14.5 Interleaving productions – rand join

The **rand join** production control is used to randomly interleave two or more production sequences while maintaining the order of each sequence. The syntax of the **rand join** production control is:

```
rs_production_list ::=
            rs_prod { rs_prod }
          | rand join [ (expression) ] production_item  production_item { production_item }
```

For example:

```
randsequence( TOP )
    TOP : rand join S1 S2 ;
    S1  : A B ;
    S2  : C D ;
endsequence
```

The generator will randomly produce the following sequences:

```
A B C D
A C B D
A C D B
C D A B
C A B D
C A D B
```

The optional expression following the **rand join** keywords must be a real number in the range 0.0 to 1.0. The value of this expression represents the degree to which the length of the sequences to be interleaved affects the probability of selecting a sequence. A sequence's length is the number of productions not yet interleaved at a given time. If the expression is 0.0, the shortest sequences are given higher priority. If the expression is 1.0, the longest sequences are given priority. For instance, using the previous example:

```
TOP : rand join (0.0) S1 S2 ;
```

Gives higher priority to the sequences: `A B C D  C D A B`

```
TOP : rand join (1.0) S1 S2 ;
```

Gives higher priority to the sequences: `A C B D` `A C D B` `C A D B` `C A D B`

If unspecified, the generator used the default value of 0.5, which does not prioritize any sequence length.

12.14.6 Aborting productions - break and return

Two procedural statements can be used to terminate a production prematurely: **break** and **return**. These two statements can appear in any code block; they differ in what they consider the scope from which to exit.

The **break** statement terminates the sequence generation. When a **break** statement is executed from within a production code block, it forces a jump out of the **randsequence** block. For example:

```
randsequence()
    WRITE   : SETUP DATA ;
    SETUP   : { if( fifo_length >= max_length ) break; } COMMAND ;
    DATA    : ...
endsequence
next_statement : ...
```

When the example above executes the **break** statement within the SETUP production, the COMMAND production is not generated, and execution continues on the line labeled next_statement.

The **return** statement aborts the generation of the current production. When a **return** statement is executed from within a production code block, the current production is aborted. Sequence generation continues with the next production following the aborted production. For example:

```
randsequence()
    TOP : P1 P2 ;
    P1  : A B C ;
    P2  : A { if( flag == 1 ) return; } B C ;
    A   : { $display( "A" ); } ;
    B   : { if( flag == 2 ) return; $display( "B" ); } ;
    C   : { $display( "C" ); } ;
endsequence
```

Depending on the value of variable flag, the example above displays the following:

```
flag == 0   ==>  A B C A B C
flag == 1   ==>  A B C A
flag == 2   ==>  A C A C
```

When flag == 1, production P2 is aborted in the middle, after generating A. When flag == 2, production B is aborted twice (once as part of P1 and once as part of P2), but each time, generation continues with the next production, C.

12.14.7 Value passing between productions

Data can be passed down to a production about to be generated, and generated productions can return data to the non-terminals that triggered their generation. Passing data to a production is similar to a task call, and uses the same syntax. Returning data from a production requires that a type be declared for the production, which uses the same syntax as a variable declaration.

Productions that accept data include a formal argument list. The syntax for declaring the arguments to a production is similar to a task prototype; the syntax for passing data to the production is the same as a task call.

production ::= production_name [**(** task_port_list **)**] **:** rs_rule { | rs_rule } **;**

production_item ::= production_identifier [(list_of_arguments)]

For example, the first example above could be written as:

```
randsequence( main )
    main                        : first second gen ;
    first                       : add | dec ;
    second                      : popf | pushf ;
    add                         : gen("add") ;
    dec                         : gen("dec") ;
    popf                        : gen("popf") ;
    pushf                       : gen("pushf") ;
    gen( string s = "done" ) : { $display( s ); } ;
endsequence
```

In this example, the production gen accepts a string argument whose default is "done". Five other productions generate this production, each with a different argument (the one in main uses the default).

A production creates a scope, which encompasses all its rules and code blocks. Thus, arguments passed down to a production are available throughout the production.

Productions that return data require a type declaration. The syntax to declare a production's return type is:

rs_type_declaration ::= type_identifier production_ identifier { , production_ identifier } ;

A value is returned from a production by using the **return** with an expression. When the **return** statement is used with a production that returns a value, it must specify an expression of the correct type, just like non-void functions. The **return** statement assigns the given expression to the corresponding production. The return value can be read in the code blocks of the production that triggered the generation of the production returning a value. Within these code blocks, return values are accessed using the production name plus an optional indexing expression. Within each production, a variable of the same name is implicitly declared for each production that returns a value. If the same production appears multiple times then a one-dimension array that starts at 1 is implicitly declared. For example:

```
randsequence( bin_op )
    bit [7:0] value;        // production return type declarations
    string operator;

    bin_op   : value operator value
               { $display( "%s %b %b", operator, value[1], value[2] ); };
    value    :  { return $urandom };
    operator : add  := 5 { return "+" ; }
             | dec  := 2 { return "-" ; }
             | mult := 1 { return "*" ; } ;
endsequence
```

In the example above, the *operator* and *value* productions return a string and an 8-bit value, respectively. The production *bin_op* includes these two value-returning productions. Therefore, the code block associated with production *bin_op* has access to the following implicit variable declarations:

```
bit [7:0] value [1:2];
string operator;
```

Accessing these implicit variables yields the values returned from the corresponding productions. When executed, the example above displays a simple three-item random sequence: an operator followed by two 8-bit values. The operators +, -, and *are chosen with a distribution of 5/8, 2/8, and 1/8, respectively.

Only the return values of productions already generated (i.e., to the left of the code block accessing them) can be retrieved. Attempting to read the return value of a production that has not been generated results in an undefined value. For example:

```
X : A {int y = B;} B ;                    // invalid use of B
```

```
X : A {int y = A[2];} B A ;                 // invalid use of A[2]
X : A {int y = A;} B {int j = A + B;} ;     // valid
```

The sequences produced by **randsequence** can be driven directly into a system, as a side effect of production generation, or the entire sequence can be generated for future processing. For example, the following function generates and returns a queue of random numbers in the range given by its arguments. The first and last queue item correspond to the lower and upper bounds, respectively. Also, the size of the queue is randomly selected based on the production weights.

```
function int[$] GenQueue(int low, int high);
    int[$] q;

    randsequence()
        int ITEM;

        TOP : BOUND(low) LIST BOUND(high) ;
        LIST : LIST ITEM  := 8   { q = { q, ITEM }; }
             | ITEM       := 2   { q = { q, ITEM ); }
             ;
        ITEM : { return $urandom_range( low, high ); } ;

        BOUND(int b) : { q = { q, b }; } ;
    endsequence
    GenQueue = q;
endfunction
```

When the **randsequence** in function GenQueue is executes, it generates the TOP production, which causes three productions to be generated: BOUND with argument low, LIST, and BOUND with argument high. The BOUND production simply appends its argument to the queue. The LIST production consists of a weighted LIST ITEM production and an ITEM production. The LIST ITEM production is generated with 80% probability, which causes the LIST production to be generated recursively, thereby postponing the generation of the ITEM production. The selection between LIST ITEM and ITEM is repeated until the ITEM production is selected, which terminates the LIST production. Each time the ITEM production is generated, it produces a random number in the indicated range, which is later appended to the queue.

The following example uses a **randsequence** block to produce random traffic for a DSL packet network.

```
class DSL; ... endclass     // class that creates valid DSL packets

randsequence (STREAM)
    DSL PACKET;

    STREAM : GAP DATA := 80
           | DATA     := 20 ;


    DATA : PACKET(0)  := 94 { transmit( PACKET ); }
         | PACKET(1)  := 6  { transmit( PACKET ); } ;

    PACKET(bit bad) : { DSL d = new;
                        if( bad ) d.crc ^= 23;  // mangle crc
                        return d;
                      } ;

    GAP: { ## $urandom_range( 1, 20 ); };
endsequence
```

In this example, the traffic consists of a stream of (good and bad) data packets and gaps. The first production, STREAM, specifies that 80% of the time the traffic consists of a GAP followed by some DATA, and 20% of the time it consists of just DATA (no GAP). The second production, DATA, specifies that 94% of all data packets are *good* packets, and the remaining 6% are *bad* packets. The PACKET production implements the

DSL packet creation; if the production argument is 1 then a bad packet is produced by mangling the crc of a valid DSL packet. Finally, the GAP production implements the transmission gaps by waiting a random number of cycles between 1 and 20.