

25. Sequences 1

25.1 Overview 5

Sequences provide a uniform way to define stream-data items and compose them into verification scenarios of growing complexity. *Sequence items* are typically driven into a DUT, but the details of the interaction with the DUT are decoupled from the data stream creation and hidden behind a standard interface.

For defining sequences, it is also necessary to define standard interfacing entities between the sequence and the DUT. Therefore, the *sequence model* has three main entities. 10

- a) Item—a *struct* that represents the main input to the DUT (e.g., packet, transaction, or instruction).
- b) Sequence—a *struct* that defines a stream of items representing a high-level stimuli scenario. This is done by generating items one after the other, according to some specific rules. The sequence struct has a set of predefined fields and methods. The user can also extend this struct. 15
- c) *Sequence driver*—a unit that serves as the mediator between the sequences and a verification environment. The generated items are passed from the sequence to the sequence driver; the sequence driver acts upon them one-by-one, typically passing them to some kind of bus functional model (BFM). 20
 - 1) The sequence driver and the BFM work as a pair, where the sequence driver serves as the interface upwards towards the sequences so the sequences can always see a standard interface to the DUT. The BFM serves as the interface downwards to the DUT, allowing the user to write sequences as appropriate. The importance of maintaining the separation between the sequence driver and the BFM becomes clear when implementing virtual sequences (see 25.1.2). 25
 - 2) To complete the picture:
 - i) A TCM does the actual driving of items into a specific DUT channel.
 - ii) The TCM resides in a BFM unit.
 - iii) For the purpose of driving data into the DUT, the sequence driver interacts only with the BFM. 30

From the point of view of a programming language, a sequence is best described as the instantiation of a *design pattern*. The same pattern needs to be instantiated separately for every type of data item that constitutes verification scenarios for the DUT. The basic implementation of the pattern is given to a user-defined sequence through *like* inheritance, and through type-parameterized code (a template), where the parameter is the data-item type. 35

25.1.1 Object model

The actual sequence object model consists of six struct types and their relationships, as show in Figure 15. 40

45

50

55

1
5
10
15
20
25
30
35
40
45
50
55

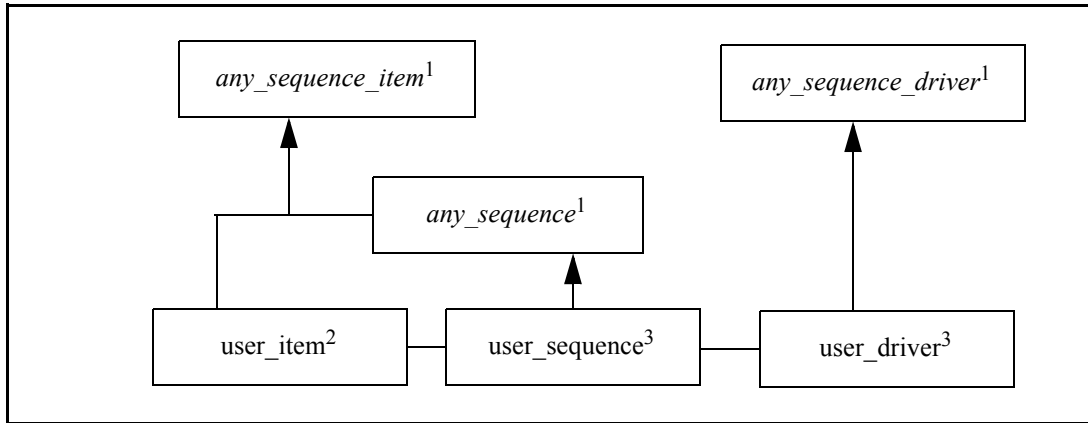


Figure 15—Sequence object model types

- 1) Built-in struct type
- 2) Declared by user
- 3) Declared by a *sequence* statement

25.1.2 Virtual sequences

BFM sequences are tightly connected to their own type and items; BFM sequences cannot **do** sequences created by other sequence statements. *Virtual sequences*, however, are not tightly connected and can **do** sequences of other types (but not items) Hence, they can be used to drive more than one agent, model a generic driver, or to synchronize and dispatch BFM sequences to several BFM drivers.

A virtual sequence is driven by a *virtual sequence driver*, which typically has references to the individual BFM sequence drivers; however, a virtual sequence driver is not connected to a specific BFM. Therefore, it lacks the logic and functionality of a BFM driver, e.g., a virtual sequence driver does not schedule items—it only drives sequences. As much of a driver’s functionality is aimed at controlling and manipulating the scheduling of items, any method that controls this functionality cannot be called for a virtual sequence. For the full list of driver interface methods that cannot be used for virtual sequences, see Table 50.

NOTE—To activate single items from a virtual sequence, use the SIMPLE sequence. To pass any parameters to the item, define the parameters as fields in the SIMPLE sequence and propagate them by constraining the item.

25.2 sequence

Purpose	Define the sequence struct, the sequence driver unit, and the sequence kind enumerated type; and extend the item <i>struct</i> for collaboration in the sequence pattern
Category	Statement
Syntax	sequence <i>sequence_type_name</i> [using <i>sequence_option</i> , ...];
Parameters	<i>sequence_type_name</i> The name of the new sequence.
	<p><i>sequence_option</i> <i>sequence_option</i> is one of the following:</p> <ul style="list-style-type: none"> a) item = <i>item_type</i>—the item to use in the sequence. This struct type shall already be defined and inherit from any_sequence_item. The item <i>struct</i> is extended by the sequence statement; the sequence is a <i>BFM sequence</i>. If this option is not used, this sequence is presumed to be a <i>virtual sequence</i>. b) created_kind = <i>kind_type_name</i>—the associated kind enumerated type to create; the default is <i>sequence_type_name_kind</i>. c) created_driver = <i>driver_type_name</i>—the associated sequence driver to create; the default is <i>sequence_type_name_driver</i>. d) sequence_type = <i>base_sequence_type</i>—the sequence struct used for inheritance. This <i>struct</i> shall inherit from <i>any_sequence</i>; the default is <i>any_sequence</i>. e) sequence_driver_type = <i>base_sequence_driver_type</i>—the sequence driver unit used for inheritance. This <i>**struct??</i> shall inherit from <i>any_sequence_driver</i>; the default is <i>any_sequence_driver</i>. <p>The definition of a sequence can be empty, containing no members.</p>

The **sequence** statement creates a new sequence struct, which inherits from the predefined **any_sequence**, which in turn inherits from **any_sequence_item**. It also creates a new sequence driver unit, which inherits from **any_sequence_driver**. Finally, it extends the (user-defined) item struct. For more details on all the resulting struct type members, see 25.4. For further details about the resulting predefined sequence kinds, see 25.2.1.

Syntax example:

```
sequence ex_atm_sequence using item=ex_atm_cell;
```

25.2.1 Predefined sequence kinds

25.2.1.1 MAIN

This sequence subtype is defined directly under the sequence driver and is started by default. It is used as the root of the whole sequence tree.

```
extend MAIN sequence_name {
    count: uint;
    !sequence: sequence_name;

    keep soft count > 0;
    keep soft count <= MAX_RANDOM_COUNT;
    keep sequence.kind not in [RANDOM, MAIN];
```

```

1         body() @driver.clock is only {
           for i from 1 to count do {
             do sequence;
           };
5        };

```

25.2.1.2 RANDOM

This sequence subtype is used for creating random scenarios based on `SIMPLE` and user-defined sequence subtypes.

```

15      extend RANDOM sequence_name {
          count: uint;
          !sequence: sequence_name;

          keep soft count > 0;
          keep soft count <= MAX_RANDOM_COUNT;
          keep sequence.kind not in [RANDOM, MAIN];
20      keep depth_from_driver >= driver.max_random_depth =>
          sequence.kind == SIMPLE;

          body() @driver.clock is only {
            for i from 1 to count do {
              do sequence;
            };
25         };
        };

```

25.2.1.3 SIMPLE

This sequence subtype generates and executes a single item.

```

35      extend SIMPLE sequence_name {
          !seq_item: item;
          body() @driver.clock is only {
            do seq_item;
          };
        };

```

25.2.2 Examples

Example 1

This example defines a BFM sequence for ATM cells.

```

45      sequence ex_atm_sequence using item=ex_atm_cell;

```

It assumes the `ex_atm_cell` *struct* already exists and establishes the following definitions.

```

50      struct          ex_atm_sequence (inherits from any_sequence)
                    type          ex_atm_sequence_kind
                    unit          ex_atm_sequence_driver (inherits from any_sequence_driver)
55

```

Example 2

This example defines a virtual sequence for an SOC environment; it defines the `soc_sequence_struct`, the `soc_sequence_kind` type, and the `soc_sequence_driver` unit.

```
sequence soc_sequence;
```

25.3 Sequence actions**25.3.1 do**

Purpose	Generate a field on a basic item or subsequence	
Category	Action	
Syntax	do <i>field_name</i> [keeping { <i>constraint</i> [; <i>constraint</i>] ...}]	
Parameters	<i>field_name</i>	A field in the current <i>struct</i> . It shall be an ungenerated field, as indicated by a leading exclamation mark (!), and also needs to be a basic item or a sequence.
	<i>constraint</i>	Any generation constraints on <i>field_name</i> .

The **do** action performs the following steps (see also: Tables 49-50).

- a) On a subsequence:
 - 1) Generates the field, considering the constraints, if any.
 - 2) Calls its **body()** TCM.
 The **do** action finishes when the subsequence **body()** returns.
- b) On an item:
 - 1) Waits until the driver is ready to perform the **do** action.
 - 2) Generates the field, considering the constraints, if any.
 The item is returned by **get_next_item()**.
 The **do** action finishes when it emits the event `driver.item_done`.

The following considerations also apply.

- **do** is a time-consuming, atomic action—as observed from the thread executing the **do** action—that activates an item or sequence.
- The **do** action can only be activated inside sequences.
- BFM sequences cannot **do** sequences created by other sequence statements.
- When **do**-ing an item, emit the event `driver.item_done` to let the sequence complete the **do** action and inform the driver the item was processed (typically, after the transmission of the item via the BFM.) Otherwise, the sequence cannot continue and the driver cannot drive more items.
- For items, waiting for the sequence driver to be ready is performed before generation to ensure generation is done as close to the actual driving as possible. Thus, if the constraints depend on the current status of the DUT/environment, that status is as accurate as possible.
- The sequence driver decides when the item is ready by managing a FIFO that also considers any **grab/ungrab** actions done by the various sequences and the **is_relevant()** sequence value. If no **grab** is done and **is_relevant()** returns `TRUE` for all sequences, the order of doing the items is determined by the order of the **do** actions in the various sequences that refer to the sequence driver, regardless of their depth or origin. All sequences and items can also be done in parallel, using the **all of** and **first of** actions. (See also: **grab()** and **is_relevant()** in Table 49.)

1 Syntax example:

```

    extend FOO ex_atm_sequence {
      // Parameters
5      i: int;
      b: bool;

      // Items/subsequences
10     !cell: ex_atm_cell;
      !seq: bar ex_atm_sequence;

      // The body() method
      body() @driver.clock is {
15         do cell keeping {.len == 4};
         do cell;
         for i = 1 to 20 do {
             do cell keeping {.address == i};
         };
         do seq keeping {.f == 2};
20     };
};

```

25.3.2 do_and_grab

25	Purpose	Execute a do action and a grab() method simultaneously
	Category	Action
	Syntax	do_and_grab <i>field_name</i> [keeping { <i>constraint</i> ; ...}]
30	Parameters	<i>field_name</i> A non-generatable item in the current <i>struct</i> , i.e., the field represents a sequence item and not a sequence.
		<i>constraint</i> Any constraints on <i>field_name</i> .

35 The **do_and_grab** action differs from a normal **do** action (see 25.3.1) in that it also grabs the **do**-ing sequence's driver. It also differs from a regular **grab()** as follows (see also: Table 49).

- When a regular **grab()** occurs, the attempt to lock the driver is immediate.
- 40 — With **do_and_grab**, no grabbing is attempted until the driver starts handling the item.

NOTE—If **get_next_item()** terminates before returning the item done by **do_and_grab**, the grab still takes effect.

Syntax example:

```

45     ...
    do_and_grab my_item;
    do my_item keeping {.x==BLUE;};
    do another_item;
50     ungrab(driver);
    ...

```

55

25.4 Sequence struct types and members

This sub clause describes the entities the sequence statement extends (the sequence-item structs) or creates (the sequence and sequence-driver structs).

25.4.1 Item structs

The **sequence** statement does not create the item struct, but it extends it. The user needs to create the item struct, which inherits from **any_sequence_item**. The main members that are added to the item struct are shown in Table 48.

Table 48—Sequence-item structs

Struct member ^a	Description	RO/RW ^b
do_location() : string	Returns a string describing the source location of the do action that generated the current item. Not relevant for sequence_items not created via the do action.	RO
<i>driver</i> : driver_type	Driver for the item, soft-constrained to its parent sequence's driver. Never use is only on the pre_generate() or post_generate() of items or sequences. The parent_sequence and <i>driver</i> of fields are assigned in the pre_generate() of any_sequence_item . The <i>driver</i> field is not defined in any_sequence_item ; to do so, use the item in a sequence statement.	RW
get_depth() : int	Depth from the sequence driver, valid from pre-generation.	RO
get_driver() : driver_type	Returns the driver for an item. The get_driver() method is declared as undefined in any_sequence_item ; to define it, use the item in a sequence statement.	RO
nice_string() : <i>string</i> is empty	A short string representing the item for tracing. It is used by trace sequence , wave sequence , and show sequence . The default implementation returns the value of to_string() .	RW
!parent_sequence : any_sequence;	Back-pointer to the sequence in which an item was created. Assigned automatically in the pre_generate() of the item. Never use is only on the pre_generate() or post_generate() of items or sequences. The parent_sequence and <i>driver</i> of fields are assigned in the pre_generate() of any_sequence_item .	RO

^aThe information in this table also applies to sequences.

^bRO designates a read-only member (which can only be read or invoked); RW designates a read/write member (which can also be set, constrained, or implemented).

25.4.2 Sequence structs

The **sequence** statement creates a new sequence struct, which inherits from the predefined **any_sequence**, which in turn inherits from **any_sequence_item**. The main members of the created sequence struct are shown in Table 49.

Table 49—Sequence structs

Struct member ^a	Description	RO/RW ^b
body() @ <i>driver.clock</i> is empty	Main method called by do of parent sequence after it generates the current sequence.	RW
<i>driver</i> : driver_type	Driver for the sequence, soft-constrained to its parent sequence's driver.	RW
event ended	Emitted immediately after body() is finished.	RW
event started	Emitted just before body() is called.	RW
get_depth() : int	Depth from the sequence driver, valid from pre-generation.	RO
get_driver() : driver_type	Returns the driver for an sequence.	RO
get_index() : int	Starts at zero (0) and gets incremented after every do . Provides a declarative style.	RO
grab (<i>driver</i> : any_sequence_driver) @ sys.any is undefined	Grabs the sequence driver for exclusive access and returns exclusive access has been granted.	RO
is_blocked() : bool	Indicates whether the sequence is blocked.	RO
is_relevant() : bool	Apply a condition for performing a do item action, so that the do action is not be scheduled until is_relevant() returns TRUE.	RO
<i>kind</i> : kind_type	The kind field that determines which sequence it is (within its when family).	RW
mid_do (<i>s</i> : any_sequence_item) is empty ;	A hook method called in the middle of do , just after item <i>s</i> is generated and before it is executed by calling the body() TCM.	RW
nice_string() : <i>string</i> is empty	A short string representing the sequence for tracing. It is used by trace sequence , wave sequence , and show sequence . The default implementation returns the <i>kind</i> , followed by the instance name (e.g., RED atm_sequence-@2).	RW
!parent_sequence : any_sequence;	Back-pointer to the sequence in which this sequence was created. Assigned automatically in the pre_generate() of the sequence if such a parent exists.	RO
post_body() @ sys.any is empty;	A hook method called after body() when sequence is started using the start_sequence() method.	RW
post_do (<i>s</i> : any_sequence_item) is empty ;	A hook method called at the end of a do , just after the execution of <i>s.body()</i> .	RW
post_do_tcm (<i>s</i> : any_sequence_item) @ sys.any is empty;	A hook TCM called after post_do() that extends the life of a do after its item_done event is emitted. The sequence driver, freed by the item_done event, no longer manages the current item so it can now handle other items.	RW
post_trace()	Called for every sequence just after a trace message about it is printed (useful for breakpoints).	RW
pre_body() @ sys.any is empty;	A hook method called before body() when a sequence is started using the start_sequence() method.	RW
pre_do (<i>is_item</i> : bool) @ sys.any is empty;	A hook TCM called at start of a do performed by the sequence. <i>is_item</i> specifies whether the context is do -ing an item or a sequence.	RW

Table 49—Sequence structs (Continued)

Struct member ^a	Description	RO/RW ^b
start_sequence()	Starts sequence activity by starting body() . Call this method instead of starting body() directly.	RO
stop()	Terminates body() , if it exists.	RO
ungrab(driver: any_sequence_driver) is undefined	Releases the grab on a sequence driver and returns immediately.	RO

^aSome of these methods are inherited from the **any_sequence_item** interface shown in Table 48.

So, do the same table footnotes also apply here (e.g., for **get_driver())??

^bRO designates a read-only member (which can only be read or invoked); RW designates a read/write member (which can also be set, constrained, or implemented).

25.4.3 Sequence-driver structs

The **sequence** statement creates a new sequence driver unit, which inherits from **any_sequence_driver**. The main members of the created driver unit are shown in Table 50.

Table 50—Sequence-driver structs

Struct member ^a	Description	RO/RW ^b	BFM only ^c
<i>bfm_interaction_mode:</i> bfm_interaction_mode_t	Specifies the way the driver and the BFM interact with each other. Possible options are PULL_MODE (the default) and PUSH_MODE . It can be constrained.	RW	Yes
branch_terminated()	Enables resumption of normal operation in the current cycle after the enclosing first of completes.	RO	No
check_is_relevant()	Forces a driver to recheck the relevance (value of is_relevant()) for each sequence that has items in the driver's item queue. Useful when something has changed in the BFM that affects the relevance of some sequences.	RW	Yes
current_grabber(): any_sequence	Indicates which sequence (if any) has exclusive control over a sequence driver.	RO	Yes
delay_clock() @sys.any	Emits the driver's clock with some inter-cycle delay to let the BFM export its state before activation of sequences in a specific cycle. Use this TCM instead of <i>connecting</i> the clock to another event.	RW	No
event clock	The main clock. The user needs to tie this to a temporal expression during the sequence driver hook-up.	RW	No
event item_done	Synchronization event for the do action in PULL_MODE . Emit this event to complete the do item and let the driver get more items using get_next_item() .	RW	Yes
<i>gen_and_start_main:</i> bool	Enables or disables automatic generation and launch of the MAIN sequence upon run() . The default is TRUE (the MAIN sequence is generated and started).	RW	No

Table 50—Sequence-driver structs (Continued)

Struct member ^a	Description	RO/ RW ^b	BFM only ^c
<code>get_current_item():</code> any_sequence_item	Returns the item currently being sent. (NULL if the BFM is currently idle.)	RO	Yes
<code>get_index():</code> int	Returns the index of this sequence driver from the all-driver list.	RO	No
<code>get_item_trace_list():</code> list of any_sequence_item	Returns a list of the items handled (sent) by the sequence driver (the item sublist of the trace log). The list is populated only if the trace sequence command was activated in On or Log Only mode.	RO	Yes
<code>get_next_item():</code> item_type @clock	Call this TCM in PULL_MODE to receive the next item from the BFM driver. This TCM is blocked until there is an item to do in the driver.	RO	Yes
<code>get_num_items_sent():</code> int	Returns the count of items sent (excluding current_item, if any).	RO	Yes
<code>get_sequence_trace_list()</code> : list of any_sequence	Returns a list of the sequences handled by the sequence driver (the sequence sublist of the trace log). The list is populated only if the trace sequence command was activated in On or Log Only mode.	RO	Yes
<code>get_sub_drivers():</code> list of any_sequence_driver is empty	For virtual sequence drivers, the writer of the specific sequence driver needs to fill in this method. It needs to return the list of subdrivers of the sequence driver. For a BFM sequence driver, this would be unchanged, i.e., it returns an empty list.	RW	Virtual only ^d
<code>get_trace_list():</code> list of any_sequence_item	Returns a list of all sequences and items handled by the sequence driver (the full trace log). The list is populated only if the trace sequence command was activated in On or Log Only mode.	RO	Yes
<code>has_do_available():</code> bool	Returns TRUE if the driver can execute a do immediately. This can only happen when one of the following conditions are met: the driver is not already busy handling a do , the driver has at least one relevant do in its queue, or the relevant do is not blocked. Even when <code>has_do_available()</code> returns TRUE, an item might not be returned immediately by <code>get_next_item()</code> ; it depends on the <code>pre_do()</code> TCM.	RO	No
<code>is_grabbed():</code> bool	Indicates the grab status of the sequence driver.	RO	Yes
<code>last(index):</code> any_sequence_item	Enables access to previously sent items in the sequence driver.	RO	Yes
<code>max_random_count:</code> int	Sets the maximum number of subsequences in a RANDOM or MAIN sequence, e.g., <pre>keep soft max_random_count == MAX_RANDOM_COUNT; // Defined to be 10</pre>	RW	No
<code>max_random_depth:</code> int	Sets the maximum depth inside a RANDOM sequence. (Beyond that depth, RANDOM creates only SIMPLE sequences.), e.g., <pre>keep soft max_random_depth == MAX_RANDOM_DEPTH; // Defined to be 4</pre>	RW	No

Table 50—Sequence-driver structs (Continued)

Struct member ^a	Description	RO/ RW ^b	BFM only ^c
<i>num_of_last_items</i> : int	Sets the length of the history of previously sent items. The default is 1.	RW	Yes
read (<i>address</i> : list of bit): list of bit @clock is undefined;	For implementing a DUT-independent interface.	RW	No
regenerate_data () is empty	Regenerates driver's data upon rerun (). Never use is only on the run () or rerun () of drivers. Some important initializations are performed in those methods.	RW	No
send_to_bfm (<i>seq_item</i> : <i>item_name</i>) @clock is empty	When working in PUSH_MODE , sends the item to the corresponding BFM. The user needs to implement this as part of achieving the hook-up.	RW	Yes
try_next_item (): <i>item_type</i> @clock	Call this TCM in PULL_MODE when the BFM has to receive an item or perform some default behavior. Unlike get_next_item (), when there is no available item waiting to be done in the current cycle, this TCM returns NULL. try_next_item () returns in the same cycle if there is no pending do action. However, if a do action started execution, then try_next_item () might take longer than a cycle, e.g., if the pre_do () TCM has been extended to take longer than a cycle.	RO	Yes
wait_for_sequences () @sys.any	Call this TCM to delay the return of try_next_item () and let sequences create items. It can also be called in other locations to help propagation of activity (e.g., among several layers of sequences). This TCM can also be overridden to implement a scheme other than the default one.	RW	Yes
write (<i>address</i> : list of bit, <i>data</i> : list of bit) @clock is undefined;	For implementing a DUT-independent interface.	RW	No

^aNever use **is only** on the **pre_generate**() or **post_generate**() of drivers. The list of previously sent items of the driver is initialized in the **post_generate**() of the drivers. ****Not sure where this footnote actually applies in this table****

^bRO designates a read-only member (which can only be read or invoked); RW designates a read/write member (which can also be set, constrained, or implemented).

^cSome of the driver members are relevant only for BFM drivers.

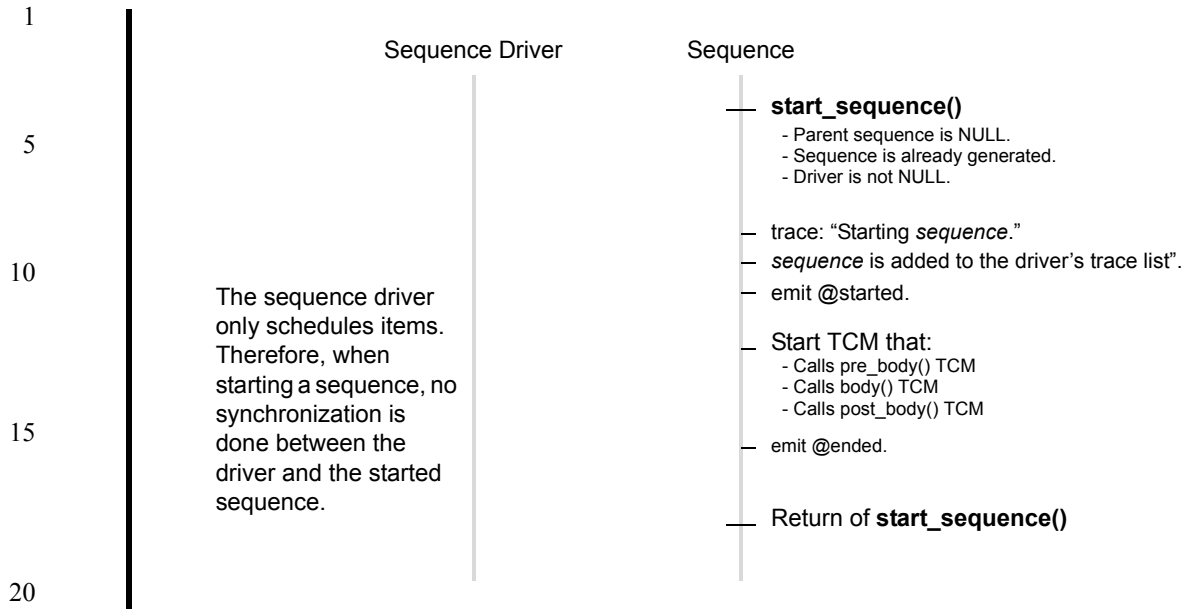
^dThis method can only be used for virtual sequences.

25.5 BFM-driver-sequence flow diagrams

This sub clause shows how the BFM, driver, and sequences interact with each other.

25.5.1 sequence.start_sequence() flow

[Figure 16](#) describes the flow for starting a sequence using the **start_sequence**() method. This flow does not depend on the *driver.bfm_interaction_mode*.

Figure 16—`sequence.start_sequence()` flow

For more information on the `start_sequence()` method, see [Table 49](#).

25.5.2 do subsequence flow

[Figure 17](#) describes the flow for **do**-ing a subsequence. This flow does not depend on the `driver.bfm_interaction_mode`.

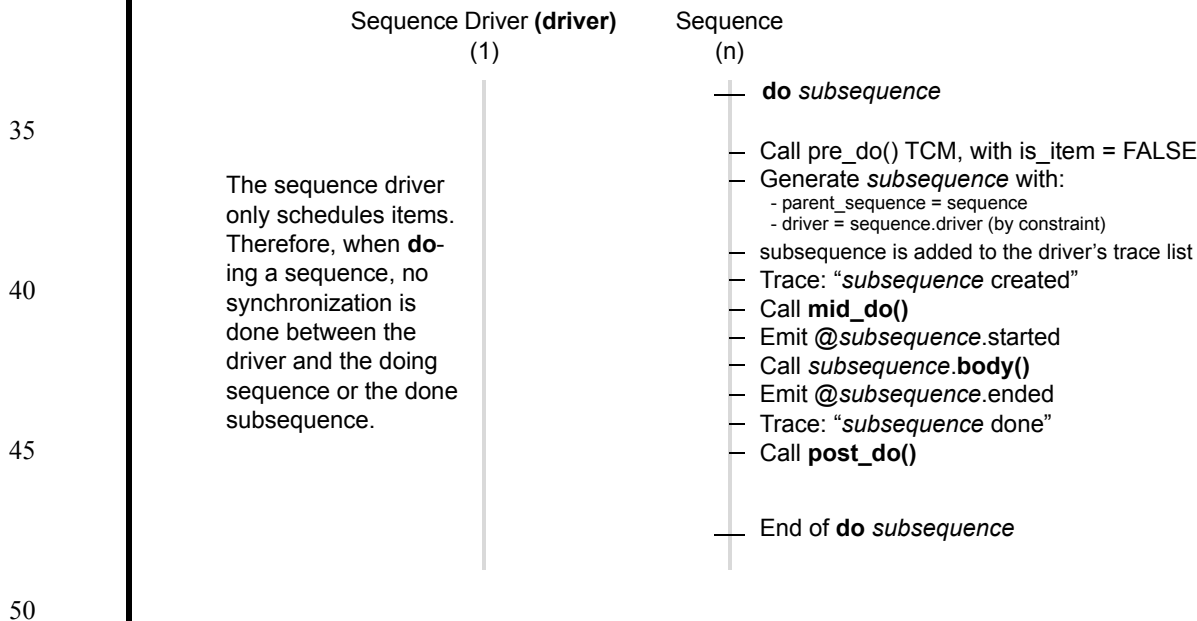


Figure 17—do subsequence flow

For more information on the **do** action, see [25.3](#).

25.5.3 do item flow in push mode

Figure 18 describes the flow for **do**-ing an item when *driver.bfm_interaction_mode* (see Table 49) is set to PUSH_MODE.

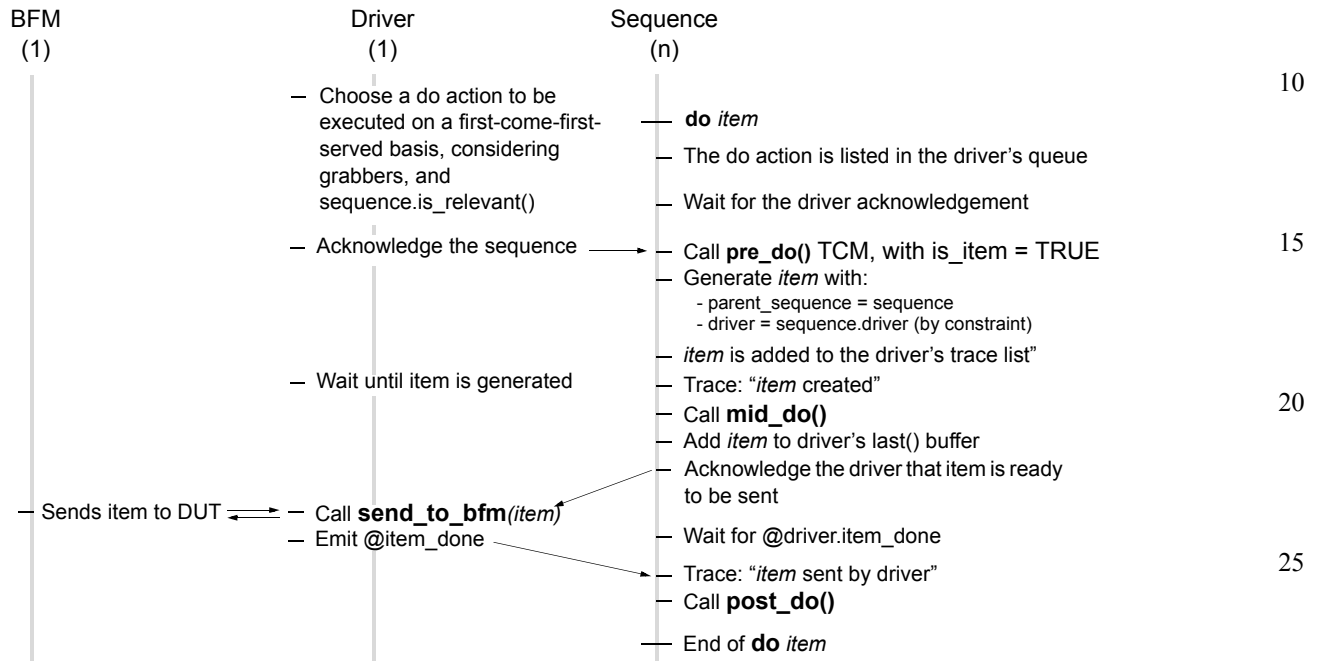


Figure 18—do item flow in push mode

For more information on the **do** action, see 25.3.

25.5.4 do item flow in pull mode using get_next_item()

Figure 19 describes the flow for **do**-ing an item when *driver.bfm_interaction_mode* (see Table 49) is set to PULL_MODE and *driver.get_next_item()* (see Table 50) is used to receive items from the driver.

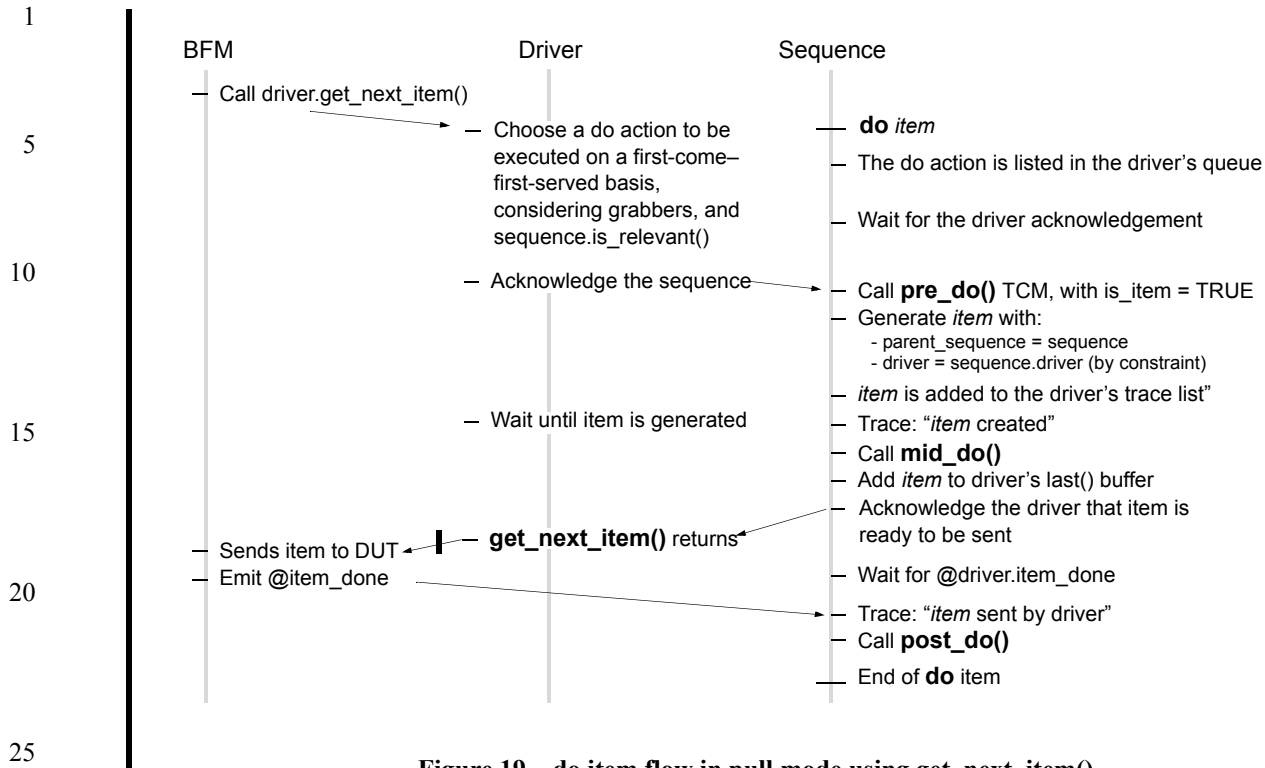


Figure 19—do item flow in pull mode using `get_next_item()`

For more information on the `do` action, see [25.3](#).

25.5.5 do item flow in pull mode using `try_next_item()`

[Figure 20](#) describes the flow for `do`-ing an item when `driver.bfm_interaction_mode` (see [Table 49](#)) is set to `PULL_MODE` and `driver.try_next_item()` (see [Table 50](#)) is used to receive items from the driver. When a `do` is chosen and `pre_do()` takes more than a cycle, `driver.try_next_item()` might also take more than a cycle.

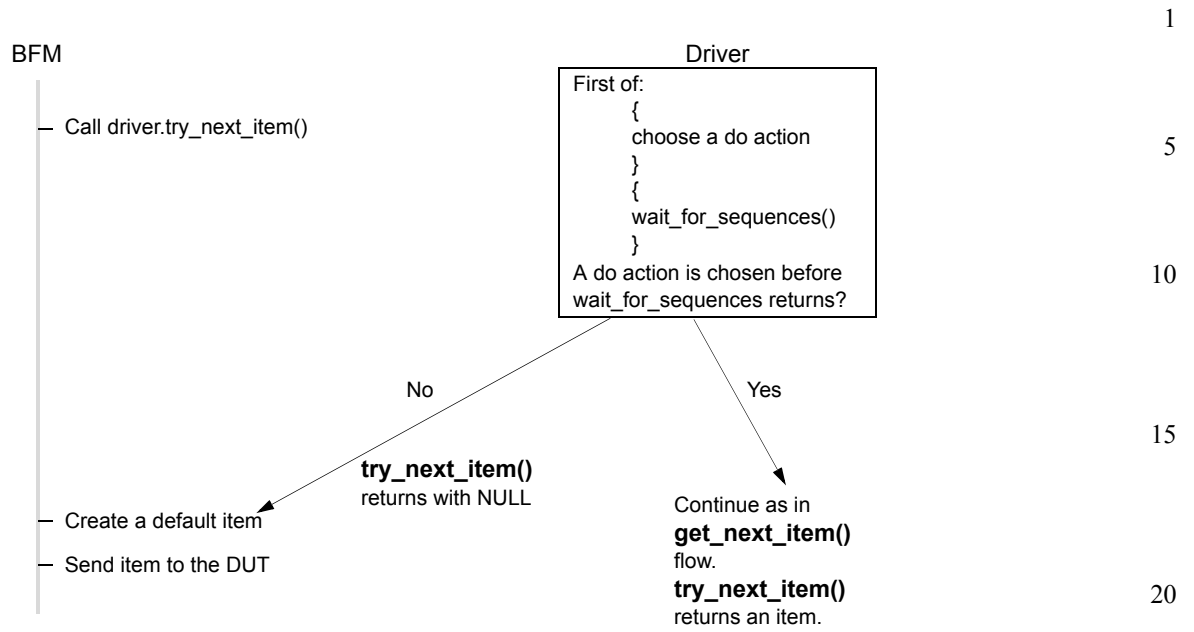


Figure 20—do item flow in pull mode using try_next_item()

For more information on the **do** action, see [25.3](#).

1

5

10

15

20

25

30

35

40

45

50

55